



Models to Code

With No Mysterious Gaps

—
Leon Starr
Andrew Mangogna
Stephen Mellor

Apress®

Models to Code

With No Mysterious Gaps



Leon Starr

Andrew Mangogna

Stephen Mellor

Apress®

Models to Code

Leon Starr
San Francisco, California, USA

ISBN-13 (pbk): 978-1-4842-2216-4
DOI 10.1007/978-1-4842-2217-1

Andrew Mangogna
Nipomo, California, USA

ISBN-13 (electronic): 978-1-4842-2217-1

Stephen Mellor
San Francisco, California, USA

Library of Congress Control Number: 2017944371

Copyright © 2017 by Leon Starr, Andrew Mangogna and Stephen Mellor

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image, we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

UML is a registered trademark of the Object Management Group.

Alf is a registered trademark of the Object Management Group.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Editorial Director: Todd Green

Acquisitions Editor: Louise Corrigan

Development Editor: James Markham

Coordinating Editor: Nancy Chen

Copy Editor: Sharon Wilkey

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York,
233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505,
e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California
LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc).
SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484222164. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper.

To Kristina and Marjorie

Contents at a Glance

About the Authors.....	xvii
Acknowledgments	xix
Foreword	xxi
■ Chapter 1: The Modeling Landscape	1
■ Chapter 2: A Simple Executable Model.....	13
■ Chapter 3: Making Translation Decisions	29
■ Chapter 4: Translating the Air Traffic Control Model	39
■ Chapter 5: Model Execution Domain.....	59
■ Chapter 6: An Extended Example	77
■ Chapter 7: Sensor and Actuator Service Domain	111
■ Chapter 8: Integrating the Application and Service Domains	129
■ Chapter 9: Event Polymorphism	161
■ Chapter 10: Pycca and Other Platforms	185
■ Chapter 11: The Translation Landscape	213
■ Appendix A: xUML Summary	237
■ Appendix B: Scall Overview	255
■ Appendix C: Pycca Language Overview.....	265
■ Appendix D: Bibliography	289
Index.....	291

Contents

About the Authors.....	xvii
Acknowledgments	xix
Foreword	xxi
■ Chapter 1: The Modeling Landscape	1
Prerequisites	1
No Magic.....	1
Elaboration: The Easy Path to Failure	2
Elaboration—Gradual Failure	2
Elaboration—Abrupt Failure.....	3
Model Destruction	3
The Value of a Good Model.....	3
A Better Way Forward: Translation	4
Executable Models	4
Platform-Independent Models.....	4
Deriving Code from Models	4
xUML: Same Notation, Different Attitude	5
The x in xUML.....	5
Translation.....	7
Our Target Technology	7
Our Translation Environment	8
A Final Word About UML and Standards.....	9
What's Next?	10

■ Chapter 2: A Simple Executable Model.....	13
An Air Traffic Controller Application.....	13
Step 1: The Class Model	15
Step 2: State Models.....	19
Step 3: Actions.....	22
Executing the Model.....	27
Standard Action Languages	27
Summary.....	28
■ Chapter 3: Making Translation Decisions	29
Reviewing the Target Platform	29
Working with the Class Model.....	31
Data Types	31
Classes and Attributes.....	32
Associations	32
Generalizations	34
Initial Instance Population	35
Describing the State Models	35
States	37
Events, Transitions, and Responses.....	37
Executing State Machines	37
Translating Processing	38
Coding from Models	38
Translating a Model	38
Summary.....	38
■ Chapter 4: Translating the Air Traffic Control Model	39
Overview of Pycca Syntax.....	39
Organization of a Pycca File	40
Translating the Class Model	40
Data Types	40

Class Definitions	42
Initial Instance Population	46
Translating State Models	48
Duty Station State Model	48
Air Traffic Controller State Model	51
Translating Actions	52
Air Traffic Controller State Activities	53
Summary	57
■ Chapter 5: Model Execution Domain.....	59
Role of the Model Execution Domain	59
Overview of ST/MX.....	60
The ST/MX View of a Class Instance	61
Managing Execution.....	63
Event Control Block	65
Signaling an Event.....	66
Catching the Event-in-Flight Error	67
Delayed Signals	67
Event Dispatch.....	69
Tracing Execution	73
Running in a POSIX Environment.....	74
Handling Errors	75
Summary	76
■ Chapter 6: An Extended Example	77
The Automated Lubrication System	78
ALS Domains	79
Lubrication Domain	81
Lubrication Class Model	81
State Models	86
Injector State Model	86
Autocycle Session State Model	89

■ CONTENTS

Reservoir State Model	91
Class Collaboration	91
Class Method and Other Activities	93
State Tables	93
Translating the Lubrication Domain.....	97
Translating Association Classes	97
Navigating Associative Relationships	99
Creation Events.....	101
Asynchronous Instance Deletion	103
Operations	104
Summary.....	109
■ Chapter 7: Sensor and Actuator Service Domain	111
Domain Overview	112
Converting Electrical Signals	113
Modeling Signal Conversion	114
Implementing the Assigner.....	120
Tracing Execution	122
Limitations	123
Value Thresholds	123
Initial Instance Population	126
Summary.....	127
■ Chapter 8: Integrating the Application and Service Domains	129
Summary of Domain Benefits	129
Each Domain Is a Black Box	130
Marking and Mapping	132
Start and Stop Monitoring Pressure	138
Update Pressure	140
Injector Pressure Alerts	142
Implementing Bridges in Pyccca.....	146
Pyccca Facilities for Implementing Bridge Code	146

The Domain Portal	147
Identifying Domain Elements.....	147
Lubrication Domain External Entity Functions	148
Implementing Injection Control Functions	148
Implementing Injector Pressure Monitoring Functions.....	150
SIO Domain External Entity Function.....	151
Updating Injector Pressure Attribute.....	151
Signaling Pressure Alerts	154
How the Portal Works	156
Summary.....	160
■ Chapter 9: Event Polymorphism	161
Generalization and Set Partitioning.....	161
Routing Polymorphic Events.....	162
Routing for each Form of Generalization	162
Torpedo Launch Example	165
Translating Polymorphic Events with Pyccca.....	169
How Polymorphic Events Are Signaled.....	174
How Polymorphic Events Are Dispatched.....	174
Summary.....	182
■ Chapter 10: Pyccca and Other Platforms	185
Design of the Pyccca Program.....	185
Platform Model	185
Domain Specific Language Processing	187
Template-Driven Code Generation.....	188
Pyccca Implementation	188
Pyccca Performance	189
Target Hardware Platform.....	189
Target Software Platform.....	190
ALS Code Size.....	190
Execution Speed	191

■ CONTENTS

Performance Discussion.....	192
Supplying Implementation-Specific Code	192
Considering Other Platforms	195
Mapping Domain Data to Berkeley DB.....	195
Platform-Model Differences	207
Alternate MX Design Discussion.....	210
Summary.....	212
■ Chapter 11: The Translation Landscape	213
A Reference Workflow for xUML Translation	213
Key Challenges.....	215
Identify Domains	216
Build and Document the Models	217
Use the Right Modeling Talent.....	218
Enter and Edit the Models Productively.....	218
Usefully Document the Models.....	219
Specify Domain Mapping	220
Populate the Models.....	221
Populate the Domain Mappings	223
Marking	224
The xUML Metamodel.....	224
The xUML Language.....	226
Action Language.....	226
Desirable Characteristics of an Action Language.....	227
Translation Considerations	231
The Pycca Workflow	232
Summary.....	235
■ Appendix A: xUML Summary	237
xUML	237
Domain	238

Bridge	239
Domain Chart	240
Class	241
Attribute	242
Data Type	243
Identifier	244
Association	245
Association Class	246
Generalization/Specialization	247
Other Activity Types	248
State Model (Instance Lifecycle)	250
Platform Independent Synchronization Rules	251
Events	251
Activities	251
State Model (Assigner)	252
Single Assigner	252
Multiple Assigner	252
Polymorphic Events	253
External Entity	254
Appendix B: Scroll Overview	255
Principles	255
Names	255
Variable Types	255
Instance Set Variable	255
Relation Variable	256
Scalar Variable	256
Data Types	256
System Variables	256
No Literals	257

■ CONTENTS

Boolean Values	257
Enumerated Values.....	257
Attribute References	257
Assignment Operators.....	257
Instance Selection.....	258
Selection with No Criteria.....	258
Selection with Criteria	258
Relationship Navigation.....	259
Signaling	259
Link/Unlink	260
Error Handling	260
Subclass Migration.....	261
Interaction with External Domains	261
Asynchronous: Signal to External Entity	261
Synchronous: Invoke Operation on External Entity	262
Self Reference.....	262
Events to Assigner State Machines	262
Class Method	263
More Commands	263
■ Appendix C: Pycca Language Overview.....	265
Invocation	265
Options	265
Lexical Conventions	266
Comments	266
Whitespace	266
C Variables.....	267
C Code	267
Name	267
Number	267

Keywords.....	267
Other Tokens.....	267
Domain Definition.....	268
domain.....	268
class	268
domain operation.....	268
external operation	269
interface prolog, implementation prolog, interface epilog, implementation epilog	269
instance	270
table.....	270
Class Definition	271
attribute	271
reference	271
subtype	272
machine.....	273
population.....	273
slots.....	273
class operation	273
instance operation.....	274
polymorphic event.....	274
constructor	274
destructor	274
State Model Definition.....	275
state.....	275
transition	275
default transition	275
initial state.....	275
final state.....	276
Activity Macros.....	276
Instance References	276
Events.....	277

■ CONTENTS

Instance Creation and Deletion.....	281
Instance Selection	281
Instance Identifiers.....	284
Navigating Generalizations	285
Appendix D: Bibliography	289
Books	289
Papers	289
Articles	290
Index.....	291

About the Authors



Leon Starr has been developing real-time distributed and embedded software with object-oriented, executable models since 1984. His models have been used in fighter jets, factory material transport control systems, ultrasound diagnostic and cardiac pacing systems, gas chromatography and semiconductor wafer inspection systems, video postproduction systems, and networked military battle simulators. He has taught numerous courses on executable systems and data modeling to systems engineers and software developers worldwide through his company Model Integration, LLC (modelint.com) based in San Francisco, California. He is the author of *How to Build Shlaer-Mellor Object Models*, *How to Build Class Models*, *Executable UML: A Case Study*, and assorted papers at uml.org and modeling-languages.com. He regularly assists project teams who model complex requirements and generate code from those models for challenging hardware and software platforms.



Andrew Mangogna has developed a number of open source tools, including pycca, which we use in this book to specify Executable UML models and translate them into code. He has also worked extensively in the medical device community. For more than 30 years, Andrew has been a hands-on builder of embedded software systems. He has worked in application areas ranging from laboratory instrumentation, remote data collection, and video special effects, to implantable medical devices. Andrew has always had a special interest in applying more formal techniques to the challenge of engineering software to create systems in a cost-effective manner with demonstrable quality. Trained in the basics of modeling by Stephen Mellor himself, he has successfully applied executable modeling techniques and model translation to many projects and has written several tools to help automate the translation process. With a keen interest in technology and a practical realization of the benefits of modeling, he has a mastery of mapping models to appropriate implementation technology to obtain high-quality software systems.

■ ABOUT THE AUTHORS



Stephen Mellor is the primary author of *Executable UML: A Foundation for Model-Driven Architecture*, as well as several other books on model-driven software. He is a frequent speaker and a key contributor to many modeling language standards established by the Object Management Group (OMG).

Acknowledgments

Any substantial project always depends on the generous help of others. Friends, colleagues, and a host of experiences have provided valuable service to *Models to Code*.

We all wish to thank Ed Seidewicz, our technical reviewer, for his many insightful and detailed comments throughout the text. His eye for detail rooted out quite a few potential and devilishly subtle errors. He passed many of our “tests” to see whether he was actually reading this stuff. We especially appreciate his patience with us, as we are well aware that our opinions and experience are often at odds with current practice in the greater UML community.

I want to express my heartfelt thanks to friends and colleagues who expressed early enthusiasm for this project. This includes many of my colleagues at SAAB Aerospace here in Sweden. At the Por Que No? taqueria in Portland, Oregon in March of 2016, Dan George and I had some amazing tacos, and I was encouraged by his infectious comments and enthusiasm. Back here in Linköping, Sweden, my present home base, I enjoyed many beers, laughs, and technical discussions with Nils Paulsson of SAAB at The Bishops Arms pub. He also contributed helpful comments on early drafts (in both the writing and beer sense of the word). Thanks also to my Scandinavian “agent” and good friend, Christer Andersson, for equal measures of encouragement and pestering me to get this project over with so I could get back to doing “real” work. My longtime friend and business partner, Michael M. Lee, has served many a quality margarita on my recurring visits back to the Bay Area in California. I would like to thank the staff at Babettes Kafferie in Linköping for letting me use a corner of their cafe as my second office and keeping me well supplied with coffee, morning buns, and warm conversation during the dark and cold Swedish winter days. Most important, though, I owe a massive debt of gratitude to my wife, Kristina, who set me up with a beautiful home office and forbade me from getting a new Xbox until the book was complete and for putting up with my writing throughout our summer wedding plans. Along with Andrew’s superb project management skills, you can thank her the most for this book making it to the shelves in 2017.

—Leon Starr

The many discussions over coffee and beer of modeling and its foundations with Paul Higham were fundamental to a broader understanding of how all the pieces fit together. There were many people, such as Melanie Gurunathan, Cary Campbell, and Thomas Brennan-Marquez, who actually used pycca to deliver real products and whose feedback was essential. Finally, without the patience and support of my wife, Marjorie Lane-Mangogna, it is hard to envision how this book could ever have happened. Few would have the patience to deal with the blah, blah, blah chatter that surrounds someone immersed in writing on a technical subject.

—Andrew Mangogna

Foreword

I published my first book on requirements analysis and modeling back in 1997. It was based on my experience working with teams to build complex, real-time, model-driven software. Like so many other books on software modeling, I left the process of transforming the models into code as an exercise for the student. Ha!

In fact, on actual projects, we were able to make this transformation, but it certainly wasn't easy. Worthwhile, yes. The models proved themselves again and again as being worth the effort. But the code translation problem has never been trivial. We always required a small team dedicated to the translation task running in parallel with the modeling effort. Note that this was not a waterfall approach, as modeling and design were being accomplished simultaneously.

Over time, the tools and technology to bridge the path from models to code improved. Unfortunately, they were locked up in proprietary solutions and expensive licensed products. This put me in an awkward position when extolling the benefits of modeling to younger engineers. Getting them interested in the benefits of modeling and a more promising style of model execution and code generation was easy. I would then be confronted with the inevitable question, "That's sounds great; how do I get started with this stuff?" I then had to concede that, while you might toy around with the ideas on your own, you really needed to get attached to a project that could fund the required tooling. Consequently, I didn't want to write another book without giving the reader a real path, any path, that took the models into running code without requiring the purchase of any expensive tools.

Andrew and I have worked together on and off on various projects over the years. One of his early translation systems extracted models from a tool called BridgePoint at the time (now xtUML) and transformed those models directly into assembly language. Because of the specialized CPU, no sufficiently optimal C compiler was available. It was a proprietary solution for a particular embedded device company, but it was a compelling proof of concept. Abstract models could be compacted and reorganized directly to the machine instruction level and run on specialized and limited resource hardware to yield the precise behavior specified by the models.

Andrew has since moved on to build a number of open source translation systems of which one, pycca, is used in this book. I eventually realized that these open source tools were the key to explaining how to bridge the gap from models to code while providing an accessible solution to any interested party.

Thus began our latest book project. After coaxing Andrew to take part, and enticing the primary author of *Executable UML* and Agile signatory, Stephen Mellor, to join in, we had the perfect mix of expertise.

As you read along, you may find that this is not an easy subject, and may require some hard work to make it through. The reward is the ability to take executable models that solve real problems through to an efficient implementation without destroying the models in the process. I want you to know that you are not alone in this journey, and we're more than happy to help if you have questions or run into obstacles. Feel free to use our contact page at the end of the book. Happy reading!

—Leon Starr
Linköping, Sweden
March 15, 2017

■ FOREWORD

In conversations with colleagues who share a similar outlook about software development, the topic invariably turns to why the practice of software development using models and translation is not more widespread. Many reasons are offered, but usually the conversation trails off into consternation over an industry that claims to want well-engineered software juxtaposed to the repeated practices that do not deliver.

For my part, I have concluded that modeling and translation are hard because abstraction is hard and requires practice to master. One need only look at the facial contortions of a young teenager when initially exposed to the idea of variables in beginning algebra to understand that we are creatures first of the immediate and concrete. Gone are the comforts of explicit numbers, and suddenly a new set of rules is in place to cope with the idea that an arbitrary symbol, which doesn't look anything like a numeral, can now range over many values. The look is similar to that of a beginning programming student struggling to fix firmly the difference between a variable and the value contained in the variable when the same symbol is used for both and only context provides the distinction. Even though I have successfully produced software systems using modeling and translation for many years, there are times when looking at pages of boxes and arrows that it is hard to see how the gap between symbols on a page and running software will ever be filled. It looks like a jump across the Grand Canyon to be attempted only by an extreme stunt rider equipped with a magic, rocket-propelled vehicle.

The young algebra student, by working through many specific problems, does come to grasp the underlying abstractions. Reasoning, and sometimes wrestling, with the concrete yields up the abstract. In this book, we undertake to illustrate general principles of translating models into code by examining specific examples. We practice a specific way to model requirements and translate to a specific type of target platform using a specific programming language. We hold tightly to our tagline of "no mysterious gaps" in our attempt to show, at each step along the way, exactly how the translated software is produced and runs. We show lots of models and lots of code, confident in the awesome power of an individual to grasp the abstractions embedded in the specifics.

There is a danger that some readers will look at the details of the platform or translation technique and conclude that, because the specifics don't match their interests or preferences, the idea of translating models into programs is itself not generally applicable. In that case, we will have failed as authors to provide the vision of model translation as a tangible engineering approach. But I am confident that there are readers who will realize that symbols on a diagram can be made tangible as a running program (as tangible as software ever is) and that obtaining code by translating models looks more like driving over a bridge than jumping the Colorado river.

—Andrew Mangogna
Nipomo, California
March 15, 2017

CHAPTER 1



The Modeling Landscape

We could begin this book with a litany of the failings and sins of software development. We won't bother with that. If you are a practicing software developer, you know the problems and have lived through them. Instead, we'll get straight to the point.

We believe that modeling requirements and translating those models directly to code can solve many of the failings of software development. Unfortunately, a huge gap exists between creating requirement models and a running program. How to bridge that gap is not obvious. If it were smooth, easy, and productive, we would expect modeling to be everywhere. But it's not. Reports vary, but only about 10 percent of development efforts use models. Clearly, models are not delivering enough value to take the software development world by storm. We believe this is caused primarily by the difficulty in bridging the gap between models and implementation.

In this book, we're going to take you on a detailed journey from a platform-independent model to a very platform-specific file of C code. We focus on a specific modeling language, translation process, implementation language, and class of hardware in an endeavor to demonstrate by example how to turn a model into code. The study of concrete examples is essential to grasping the general principles of obtaining code from models. This means that occasionally you may see concepts introduced that are not fully described until later. It may be necessary—in fact, it is encouraged—to take multiple reading passes through some of the chapters.

Prerequisites

We assume that you have some interest and background in modeling and UML. Although we won't be teaching modeling here, we explain key aspects of the Executable UML (xUML) modeling language we use, provide external references, and expand on our modeling approach as we go.

There will be code. So we assume you have some familiarity with the C programming language. We'll be walking through a lot of it as we proceed. We don't make any deep dives into the dark recesses of C, so a passing knowledge should be sufficient for these purposes. And for whatever we can't squeeze into the book, we'll provide online references.

No Magic

Software modeling tools and techniques have been around for many years, yet the industry seems to be waiting for some future development to set us on the right path and make the whole process work like magic. But this book isn't about the future, unicorns, or fairies. It is about what you can do today, on a real project in a real organization to produce deliverable code.

Elaboration: The Easy Path to Failure

To appreciate our approach, it is helpful to compare it to a completely different and common approach called *elaboration* that has repeatedly failed to deliver results. Elaboration was initially popular because of its intuitive appeal and low bar of admission when it comes to learning anything new about analysis and modeling. Knowledge of notation and skill in object-oriented programming were the only engineering prerequisites. As it turns out, this practice generally leads to disaster. In fact, elaboration has caused many a project to abandon the whole idea of model-driven software, stigmatizing the practice of modeling in the process.

Elaboration has you start out creating a “system” or “architectural” design by building a high-level model. The concept of *high level* has no definition, so it is supposedly intuitive. That fact alone should be a sign that you are headed for trouble. This period is often concerned less with determining how the components of the system will work together than with partitioning the work among the developers on the project.

Over time, classes become less high level and more Java-ish or C++-ish. It all seems so natural. Implementation boundaries cause model elements to be grouped along task, thread, or CPU boundaries. Private and public methods and inheritance, in whatever particular form is supported by the target language, are sculpted in and around the modeled classes.

Sooner or later in the modeling process, you will trip up on some vexing problem. After banging your head a while trying to solve it at your oxygen-deprived modeling altitude, someone says, “Oh, that can be fixed in implementation; don’t worry about it.” Then you smooth over the whole nastiness with an aggregation diamond or whatever and keep going. That is where the lack of a definition of *high level* makes it easy to avoid solving problems while modeling.

More modeling goes along until you realize that it’s hard to know when to stop modeling. The usual guideline is to stop when all that remains is “implementation detail.” Sadly, there is no good definition of implementation detail, either; there are usually lots of discussions and disagreements over what constitutes a detail. This is another warning flag of trouble ahead.

You rinse and repeat until either gradual or abrupt failure sets in.

Elaboration—Gradual Failure

Here the models are elaborated until code results. Attributes are added, states are added, and models are reorganized as necessary. Eventually, each modeled class corresponds directly to a Java or other implementation language class. Each attribute corresponds to a property of an implementation class. Each package corresponds to a thread, task, or other software execution unit. And you get code. As code is produced, there are even tools that will “round trip” the code base with the model diagrams to keep them in sync.

How is this a failure? Two reasons. First, the “high-level” models were destroyed. Adding detail progressively obscures requirements, and the models morph into pictures of the code. These pictures may be useful for navigating the code base, but any statement of the fundamental application requirements is soon obscured.

So what was the point of building models you were going to destroy? They’ll have no life beyond the project. They’ll be of little value moving to other platforms or software architectures. They’re not substantially easier to understand than the code base itself and will be completely foreign to anyone in the organization outside the software development staff.

Second, the modeling process wasn’t of much value because it wasn’t focused on capturing the requirements of the problem. The difficult problems got swept under the rug and then later solved while adding code, when there is no choice but to do something, anything, however right or wrong that may be. So, the preliminaries are rightly perceived, in retrospect, as an additional, time-consuming, and not necessarily helpful phase.

Elaboration–Abrupt Failure

In the abrupt case, pressure to deliver increases to the point where everyone freaks out, casts the models aside, and starts coding. This start of coding is deemed when the “real work” begins, and any further attempts to understand the problem being solved are abandoned. The models may be referenced later somewhat, but attention inexorably shifts to the mounting code base because that’s now the only thing that matters.

Model Destruction

In either failure scenario, the models are rendered useless. In the gradual case, the essential application logic is obscured, and the models become little more than pictures of the code. In the abrupt case, the models may survive for a short time as advisory documentation, but soon lose their relevance. After the project, in either case, the value of having built the models in the first place is rightfully brought into question. With results like these, it is no wonder that modeling never became widely accepted as an essential software development activity.

The Value of a Good Model

Plenty of other books detail the many good reasons to model the requirements of your application. Here we have to assume that you think models have the potential for redeeming value even if that value has appeared tenuous in your experience. Let’s instead try to boil down whether you are getting tangible benefits from your modeling efforts with a hardware engineering analogy.

Consider a typical workflow of our electrical engineering cousins when designing a printed circuit board (PCB). They first capture the *application logic* of the circuits in a schematic diagram. They define the components that are required and the way they are connected. The schematic defines exactly how the board works.

Only after schematic capture is consideration given to PCB layout. Layout is about the physicality of the board: component sizes, trace widths, ground planes, signal distances, and many other elements must be specified. In many organizations, the person doing the layout is different from the one who did the schematic capture, because the skills for the two tasks are that different. When the board is fabricated, the electrical circuits *must* function as dictated by the schematic diagram—layout does not change functionality.

To an electrical engineer, a schematic diagram is a model of the application logic. What is excluded from the schematic diagram is the physical arrangement of the components on a PCB and all the practical considerations of electrical connectivity. That exclusion is then introduced by the board layout.

After the board is fabricated, if testing indicates an error, then not only is the board patched, but the schematic diagram is scrupulously updated to reflect the fix. Amazingly, the entire workflow of schematic diagram, board layout, and schematic updates is never a subject of contention among electrical engineers. The value of a schematic diagram and the absolute necessity that it accurately describes the way a board works is part of the shared outlook for electrical engineers.

If you ask an electrical engineer a question about how a board works, immediately he or she will pull out a thick wad of paper containing the schematic diagram drawings for the board. They flip to the relevant page and proceed to point and trace through the diagram to answer the question. They do this with absolute confidence that the answer obtained from the schematic diagram matches the way the board works on the lab bench.

Now, if someone asks you a question about how a software feature works and your first instinct to answer the question is to browse through code files, to read and execute, in your head, a bunch of code, then whatever models may exist for that software are not providing enough value to be worth producing. On the other hand, your models truly capture the requirements of the system when someone asks, “How does the gronkolator masticator know to turn itself off?” and you reach, without hesitation, for a thick wad of paper (or load up a model file on your giant display) containing the software model diagrams and start pointing and tracing to answer the question.

A Better Way Forward: Translation

We subscribe to an entirely different approach to getting code from models called *translation*.

This translation approach is based on two fundamental tenets:

- The models are *detailed* expressions of application logic. They should capture and formalize requirements of a problem.
- Code is *derived from* the models. The models are not destroyed in the process of translation.

The models are detailed in two key respects: executability and platform independence. The vague concepts of *high* and *low* level are not meaningful or relevant here.

Executable Models

The models are fully executable without code. By *executable*, we mean that there must be a clear, complete, unambiguous set of rules for running the models.

Each set of models is adequately detailed so that it can be executed and tested without the need for any programming language code. This is made possible by a small set of well-defined, platform-independent execution rules and descriptions of processing that operate exclusively on model elements. The execution rules have been designed so that they can be implemented on a broad array of platforms such as a single microcontroller or spread out across a distributed system with many parallel processing units.

If you are curious, you can get a taste of the execution rules by looking at how platform-independent synchronization is managed in xUML by visiting www.modelint.com/MBSE or www.executableuml.org.

Platform-Independent Models

The models must *not* contain implementation technology. This is for two reasons.

First, the logic of an application stands apart from the means and methods of its implementation. We have long been defining logic and processing without using any programming language. Second, as many developers have discovered, implementation features are more flexibly defined in code rather than in the constrained context of model formalisms. Any implementation concepts assumed in the models hampers the choices that can be made downstream. It becomes more difficult to optimize and adjust to platform-specific requirements when implementation choices are forced by the models. It's the age-old principle of using the right tool for the right job. Programming languages are great for devising implementations and not so good at expressing application logic distinctly. Models of the type that we build are great at expressing logic and required data and synchronization, but are a terrible place to design code.

Deriving Code from Models

Assuming that the models are executable and platform independent, we get code by mapping them onto the implementation technology. All implementation decisions and artifacts are folded in downstream from the modeling process. Consequently, all the work and intellectual property that goes into the models is maintained, distinct from any particular implementation.

We never mix code anywhere back into the models themselves. We need a modeling language that supports these features and, to bridge the gap to code, we need a way to take those models as input and add the necessary implementation technology to transform them into running programs on a hardware platform. Let's start with the language.

xUML: Same Notation, Different Attitude

We use Executable UML (xUML) as our modeling language. Like most modeling languages, it has a graphical notation and underlying semantics. Semantics is just a short way of saying, “what the notation actually means.” xUML uses a subset of the UML notation on top of platform-independent execution semantics. This is not the same as the object-oriented semantics employed by the greater UML community. Both *Executable UML: A Foundation for Model-Driven Architecture* by Stephen Mellor and Marc Balcer (Addison-Wesley, 2002) and *Model Driven Architecture with Executable UML* by Chris Raistrick et al. (Cambridge University Press, 2004) provide complete descriptions of xUML modeling.

Let’s take the class symbol, for example. Most UML folks will look at it and imagine a “high level” Java-ish or C++-ish class. This brings with it all of the attendant object-oriented programming concepts such as object references, public/private methods, inheritance, and so forth in whatever object-oriented programming language you assume is most likely targeted.

In xUML, the semantics are entirely different. First of all, there is no implicit presumption that we are targeting an object-oriented programming language. Instead, xUML is built on a set of mathematical formalisms. Now, we do *not* intend to write lots of Greek letters or upside-down Latin characters when we describe a model. Being based on a formalism is not the same as mimicking a formalism. We also don’t want to imply that the modeling formalism is somehow associated with proving the correctness of the resulting program. We’ll leave formal program proofs to the computer scientists and simply strive to be better software engineers. It is not necessary to understand the underlying formalisms to use xUML, because they are expressed as a set of modeling rules. However, understanding the fundamentals will make you a better modeler. The formalisms remove any platform bias, and the mathematical basis provides assurance that we can translate to any target platform required to meet our specific engineering needs, object oriented or otherwise. A good introduction to the mathematics of logic and its use in describing data can be found in *Applied Mathematics for Database Professionals* by Lex de Hann and Toon Koppelaars (Apress, 2011).

The *x* in xUML

As programmers, we read a lot of code. Reading code leads to understanding a program’s behavior, because you can run the code in your head. In this sense, C code is executable because there are well-documented execution rules to accurately predict what the code will do. Consider, for example, the following function, written in C, which computes the y-coordinate of a line, given the x-coordinate, slope, and intercept:

```
int
linearValue (
    int x,
    int m,
    int b)
{
    return (m * x) + b ; /* yes, the parentheses are unnecessary */
}

int y = linearValue(5, 10, -2) ;
```

Any two C programmers can mentally execute this code line by line and conclude that the value contained in the *y* variable will be 48. It is not subject to opinion, mood, or perspective, because each programmer envisions the same execution model that defines how the program runs. (And if the answers diverge, it means that one programmer or both are misunderstanding that model!) Yet we also know that real computers do not behave in strict accordance with the execution model of the C language. The C compiler lays out memory for variables, selects the correct integer instructions, and generates the necessary

instructions to enforce the conventions for passing arguments. But it *must* yield the same result as our execution model. You don't need to envision the compiler's complicated layout scheme to get the correct result if the compiler is doing its job correctly.

The same principle applies to an executable model. You must be able to read a model and accurately predict an outcome, though the model may be implemented entirely differently from the way we execute it in our heads. Nothing is open to subjective interpretation. UML alone does not provide this level of certainty, but the xUML we use in this book does. Figure 1-1 illustrates this principle.

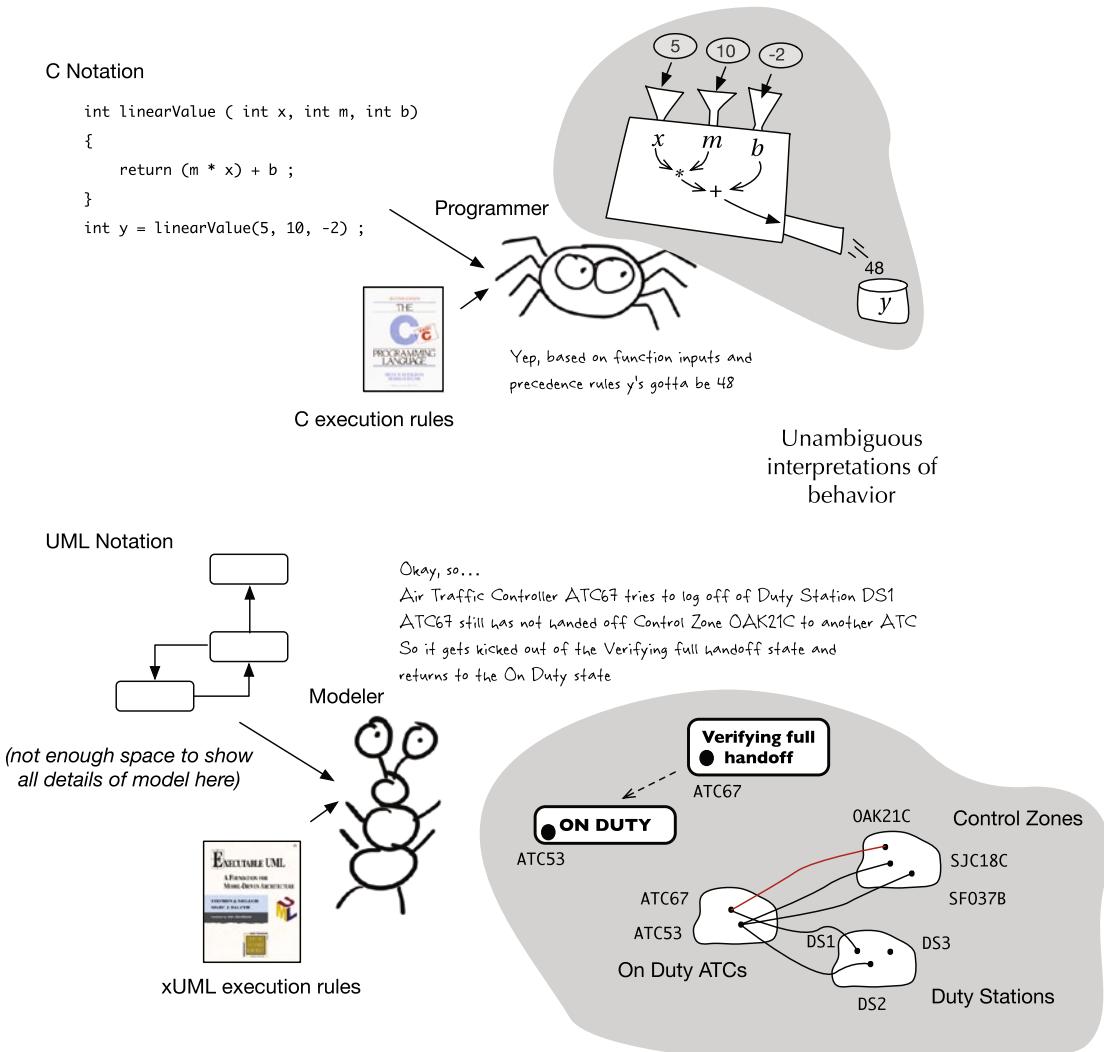


Figure 1-1. Mentally executing code and models

The syntax of any executable language associates keywords and symbols with corresponding executable meaning or semantics. Assembly language, for example, operates with the registers and memory structure of a particular processor. C, on the other hand, lives in the world of variables, structs, arrays, loops, and functions. Our executable modeling language takes the level of abstraction up a few notches and operates in a world of sets, relationships, life cycles, and data flows. This is ideal for capturing application logic without distraction.

With an executable model, you can model applications unambiguously and in sufficient detail that they can be executed and verified just as you commonly do with code, without actually *being* code. This forces you to think about what the application is doing. Decisions must be made. Inadequate subject-matter knowledge must be filled in. You cannot defer difficult problems with statements like, “We’ll sort that out when we write the code.” When we model the requirements for an air traffic control system, for example, we must decide exactly what is to happen for an on-duty controller to go off duty. We must define what data is required, what computations must be performed on it, and how the computations must be sequenced or otherwise synchronized. Model-level executable semantics let us specify all of this without assuming any particular target programming language, design patterns, processor distribution, framework, library, or any other implementation technology.

Translation

Executable models are not the same as code. Executable models describe application logic. To get to code, an executable application model must be translated onto implementation technology.

Our insistence on a separation of logic and technology has created a larger gap to fill than if we had insinuated implementation concepts into our modeling language. If your modeling language looks a lot like an object-oriented programming language, the step to get object-oriented code is not very big. In our case, we must fold in all the required technology. This can run the entire gamut of programming technology such as a target programming language, design patterns, processor distribution, framework, runtime library, and any other mechanisms of implementation we need to create a running program.

It is important to emphasize that translation adds nothing to the logic or behavior of the system. It adds computing technology only. Whereas earlier we ignored implementation technology to capture concisely the application logic, translation shifts the emphasis from application logic to technology. Because we are concerned only with how computing technology implements the application logic, the details of how the application logic meets the requirements of the system can be ignored. When we translate, we don’t try to second-guess the application logic, just as we didn’t try to specify the implementation mechanism when we were modeling.

Our Target Technology

The landscape of modern computing technology is vast, so many decisions need to be made in determining the type of computing technology that should be applied to a given problem. Translation is directed, then, at a particular target technology and environment, and we must be specific about the details of how and where that program will run.

Although the approach we describe can be used in multiple contexts (web applications, software-oriented architecture, batch programs, and distributed systems, to name just a few), it would be distracting and counterproductive to keep switching back and forth between implementation environments. Instead, we pick one and consider the consequences all the way through to the code. This book shows one example of how to apply specific technology to translation. It is not a comprehensive treatise of translation theory. Still, we expect you to come away with a greater understanding of how translation is accomplished.

We demonstrate translation assuming our target computing environment has the following characteristics:

- The environment is based on an embedded microcontroller. These typically have less than 512 KB of memory, and frequently as little as 32 KB. We will not consider the very small end of this scale. Microcontrollers with less than 16 KB of read-only memory or less than 4 KB of read/write memory are outside the design scope of our target environment.
- The implementation language is C. C is common in this realm, and good C compilers are available.
- The class of applications is reactive in nature. The software responds to unsolicited external stimuli and interacts with its environment via a set of hardware peripherals. This is in contrast to applications that are primarily transforming in nature, such as rendering a graphical image.

WHY NOT C++?

Some readers may wonder why C was chosen rather than C++. Despite their syntactic similarities, C and C++ are very different languages. C++ provides powerful features, not all of which are supportable in a microcontroller environment. Some embedded projects use subsets of C++ to avoid these difficulties. But programming language features make it easier for human programmers to accomplish programming tasks. The same features are not as useful in a translation environment. The role of an implementation language in translation is different than when you are directly coding a program. A translation program doesn't care whether a particular concept is not well supported in the implementation language. Finally, several, obscure incompatibilities exist between C and C++, and our translator program needs a construct in C that is illegal in C++.

This target technology would not work well for a web application. But that does not mean the translation approach fails for a web application; it simply means that models for web applications must target a different technology.

Our Translation Environment

There is a broad spectrum of ways to accomplish the translation of a model into code. Any mechanism that faithfully maps the logic of the application onto its code equivalent without destroying the model is a candidate.

At one end of the spectrum, translating a model starts with software tools that capture a graphical representation of the model. Dozens of UML drawing tools are available. The model's meaning is captured in a database by using sophisticated tools. The database is then traversed in potentially complicated ways to produce code. The main advantage of this approach is power and generality. The main disadvantage is that it appears to be magic. This perception is heightened by the number of elements you must understand before you can make anything happen and by the amount of unseen processing that goes on to get between the model and the code.

Our approach starts at the other end of the spectrum. We have defined the details of the target platform and fixed the way model execution rules are implemented within the technology of that target platform. A set of platform-specific rules determine the choices for how model elements are mapped onto the implementation constructs. We encode the mapping of model elements into implementation constructs by using a text-based domain-specific language (DSL). Then, a relatively simple tool, named `pycca`, generates the output C code. The main advantage is that we are assured of obtaining a program that matches our target platform technology. The main disadvantage is the lack of integration to front-end model development.

Here is our basic workflow:

1. Create an xUML model of the subject matter. Use your favorite model-drawing tool. It does not matter which one.
2. Analyze the model to determine the translation characteristics.
3. Using the drawing tool artifacts, encode the structural model elements, such as classes and state models, into a text file by using the pyccca DSL.
4. Translate the actions into C code that is also placed directly into the pyccca file.
5. Generate a code file by running pyccca.
6. Compile and link the resulting code.

What pyccca lacks in front-end integration, it makes up for in these important characteristics: platform specificity, transparency, and availability.

Platform Specificity

Every target platform has its engineering challenges. If you have the luxury of commonly available hardware and software technology to satisfy the needs of your application, you are indeed fortunate and can focus your attention on model logic. However, experience shows that there are many expectations for the characteristics of an implementation. For our example target, small memories and slow processors mean that techniques that would be acceptable on a conventional desktop computer simply do not work in a microcontroller environment. Unlike many code generation schemes, pyccca lets you handcraft your own algorithms to implement modeled activities. This capability is crucial when you grapple with the idiosyncrasies of legacy code or other peculiar aspects of your target platform.

Transparency

The correspondence between the model, the pyccca source, and the generated C code is clear and direct. You can understand the role that each part plays in your program. The particulars of what constructs are supported by a drawing tool or how a drawing tool stores model content does not affect the translation to code.

Availability

Pyccca is freely available, and all it takes to get this scheme to run is a little brain power and a C compiler.

A Final Word About UML and Standards

We use the UML notation as a lingua franca for presenting model diagrams in this book. When it comes to models, however, we are mostly concerned with the stuff underneath the notation: model execution semantics and platform independence. For this, we turn to xUML, which provides us with exactly what we need: a strong mathematical foundation and platform-independent rules for running and testing models.

Occasionally, the UML notation is at odds with the xUML semantics. This is largely because UML is biased in the direction of an object-oriented programming paradigm, whereas xUML is platform neutral. As we consider the notation the less important factor, when push comes to shove, the execution semantics will always prevail in our work. After all, it is easier to bend the interpretation of a graphical symbol than to overcome the surprise of a whole new notation. Like the English language, the consequence of UML being a lingua franca is that everyone is allowed to give the language its own regional flavor (sometimes to the annoyance of the native speakers!).

You may also notice that we don't claim to adhere to any particular UML standards. We think the best standards in software are those that codify existing practice or attempt to ensure interoperability. Existing practice in UML is targeted in many directions, and interoperability is demonstrated only by explicit testing. With its dizzying selection of diagram types, UML is used for activities ranging from cocktail-napkin sketches to, as we present here, formalized statements of software requirements. We find it difficult to pin down existing industry practice. Usually, published standards result in nonstandard implementations. From C compilers to SQL query languages, standards compliance does not seem to inhibit extending functionality and limiting interoperability, because implementation necessities always win out in the end. Noncompliance to a certain standard is just another engineering trade-off that a project team must evaluate. We are not averse to standards that contribute value to a project team, and if it is important to your project team to have UML diagrams that adhere to a particular version of the UML standard, then you should pursue that goal. We still hold to the proposition that it is not the *shape* of the boxes and arrows that matter; rather, it is the *meaning* attributed to them that determines true value.

What's Next?

Our goal in this book is to teach the key principles of how models are translated into code by using detailed examples. Model translation is not magic and need not be shrouded in mystery. We do our best to be grounded firmly in the engineering realities of producing running software. Though we present completed models and explain what they mean, it is not our intent to teach modeling here. While we demonstrate how pycca is used to accomplish a translation into code, this is certainly not a pycca manual. (The documentation for that is available online.) We show you C code and assume that you can read and understand it. All the details of models that don't fit here are readily available from www.modelstocode.com. There you find all the examples completely worked out.

To get you to code as soon as possible, we're going start by introducing a small example air traffic controller model in Chapter 2. This model will also serve as a nice introduction (or review, depending on your background) of xUML basics. We'll show how the model captures platform-independent application requirements and how we can walk through a model execution scenario.

We'll then review our example executing model in Chapter 3 with a tour of the types of design decisions that must be made to influence the translation process. Pycca will be introduced as a language for specifying these decisions. The first glimmer of code structures to be shaped will appear in this chapter.

Things get serious in Chapter 4, where we write pycca statements to define our full translation to C. Throughout, we will emphasize the tight correspondence between the input model, the pycca statements, and the generated C code.

In Chapter 5, we pop open the hood of the supplied model execution runtime code that will be linked with your generated code. The focus here is on how state machines are executed and events are queued, dispatched and, where necessary, delayed. We will show how real-world interactions via interrupts can be serviced in conjunction with model execution.

Having successfully translated our small model, we'll expand on how real systems are put together with a new and more challenging example. The automatic lubrication system for vehicles and machinery will be introduced, featuring multiple modeled components called *domains*. Chapter 6 introduces a domain concerned with user application logic, and Chapter 7 presents a domain called Signal I/O (SIO) that makes solid contact with physical sensors and actuators.

We'll pull it all together in Chapter 8, integrating the domains with a concept called *bridging*. This will allow us to knit the domains together into a complete functioning system.

In Chapter 9, we'll take a look at how polymorphic events are defined, signaled, and dispatched in generalization relationships.

Performance is addressed in Chapter 10, where we'll show detailed metrics for the system we have just translated. The design of the pycca program itself is discussed as well as how to target a completely different platform.

Finally, in Chapter 11, we'll step back and put the entire model and translation workflow in perspective. A reference model of this workflow is presented, and then the pycca workflow we have described is placed in the reference context.

Now, let's begin with a simple model...

CHAPTER 2



A Simple Executable Model

This chapter describes the main elements that make up an executable model for a sample application. The application is deliberately small so that we can show how to translate the whole model into running code, with no mysterious gaps. We'll use the same principles later to build larger, more complex applications.

This chapter also serves to explain our interpretation of UML graphical notation. We use UML graphical notation to represent the model, but xUML varies considerably in the semantics that apply to that notation. We want to make sure you understand the meanings we attribute to the graphical symbols.

While reading through this chapter, you may find it helpful to refer to Appendix A, which summarizes key xUML notation and semantics.

An Air Traffic Controller Application

The air traffic control application manages air traffic controllers (ATCs) in a control center. The system ensures that all local air traffic is directed and that the ATCs take appropriate breaks. Figure 2-1 shows their world.

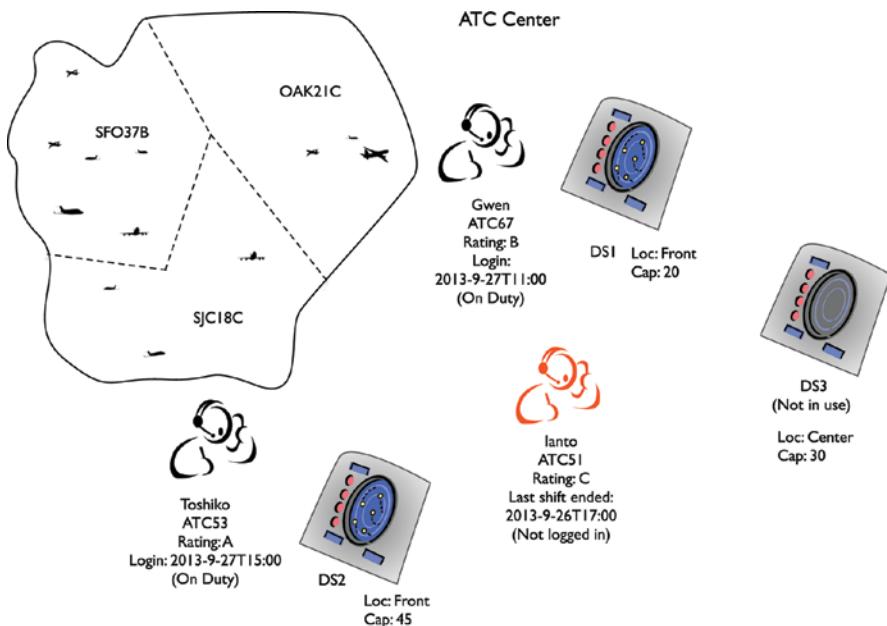


Figure 2-1. An air traffic control center

The application must do the following:

- Log an air traffic controller (ATC) onto a duty station
- Record the zones under the control of a duty station
- Verify that a control zone is always under the control of a duty station
- Record the handoff of a control zone to another duty station
- Log an ATC off a duty station
- Prevent an ATC working a shift longer than 2 hours 15 minutes

This description is surely incomplete and insufficient for us to build the application, but it's a start. Some would suggest investigating further and tying down details in a requirements document; others, of a more agile persuasion, would recommend writing executable code to explore the application in the absence of a complete understanding. We take the best of both worlds and build a detailed executable model that we can use to investigate the application in more detail. We can use the model for evaluation, we can run it, and we can test it. And when we're satisfied, we can translate it directly into code.

We build the model as three interconnecting facets, as shown in Figure 2-2.

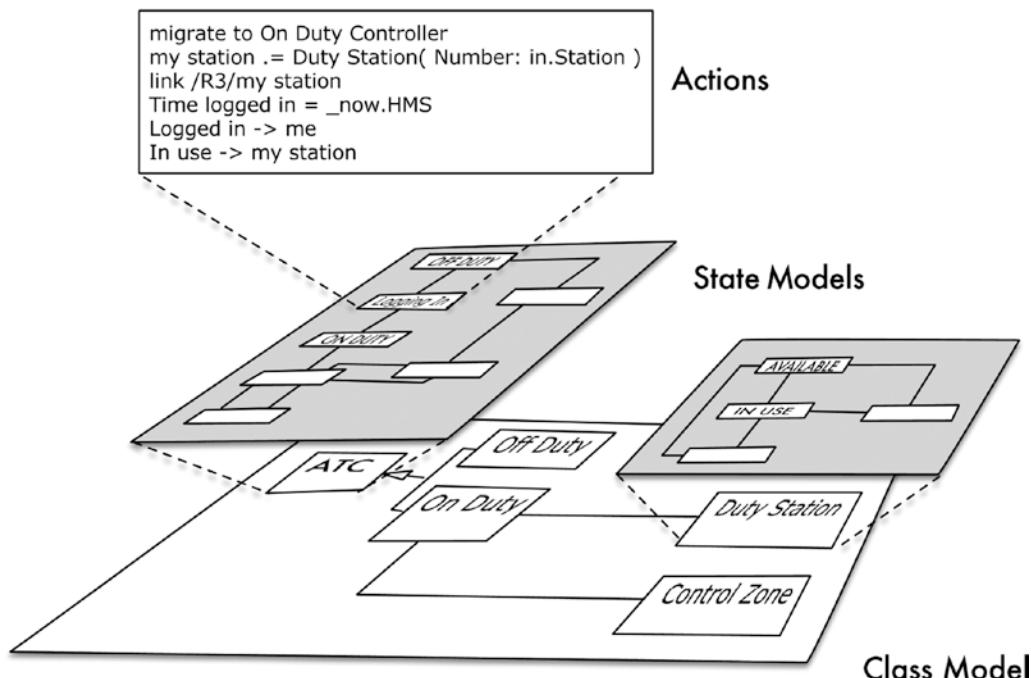


Figure 2-2. The three facets of the modeling language

The first facet is a *class model*, which declares the conceptual entities in the system. We use a limited subset of UML class diagram notation to represent the model. The class model defines classes, their attributes, and the associations between classes.

The second facet consists of *state models*, which describe the behavior of a class over time. Again, we use a subset of the UML *state machine diagram* notation to represent the model. There is a state model for each class. There is a copy of the state model, a *state machine*, for each class instance. Each state machine

and, therefore, each instance, has its own independent notion of its current state. Most of the dynamic behavior in a model takes place in the context of a state machine.

The third facet is *actions*. Actions give the details of computation and other algorithmic processing. Depending on how the actions are represented, they might look like code. However, actions are really the specification of model-level computations on model-level elements, and that is a better way to think of them.

Each of these three facets represents a single perspective of the system. They fit together into a coherent whole that represents a solution to the logical problem your application poses. The integrated facets define a complete, executable domain that can be translated into code. A *domain* is a coherent subject matter with its own set of policies and rules. Generally, applications are composed of multiple domains that interact to satisfy the whole set of application requirements. There is more to be said about domains, and we will take that up in Chapter 6. For now, it is enough to know that a modeled domain is an independently executable and testable composition of all three facets.

The order of model construction is significant:

- Class model, to capture definitions, data, rules, and constraints
- State models, to capture modes, control intervals, and synchronization
- Actions, to capture signaling, data manipulation, and computation

Each step lays a foundation and a constraining framework for the next. For example, the fact that a shift ends at a certain time (contained in the class model) is more fundamental than the exact manner in which the time is determined (contained in the state models and actions). So we nail down the elements least likely to change first, with the more volatile elements defined and added later. This approach sets up a stable underlying structure upon which to build, and decreases the overall refactoring required whenever a change occurs.

It is important to realize that executability depends on all three facets. You must know how you are computing (actions), when you are computing (state models), and exactly the data on which you are computing (class model).

Step 1: The Class Model

Figure 2-3 shows a class model for the ATC system. It has been annotated so we can point out our interpretation of the graphical symbols.

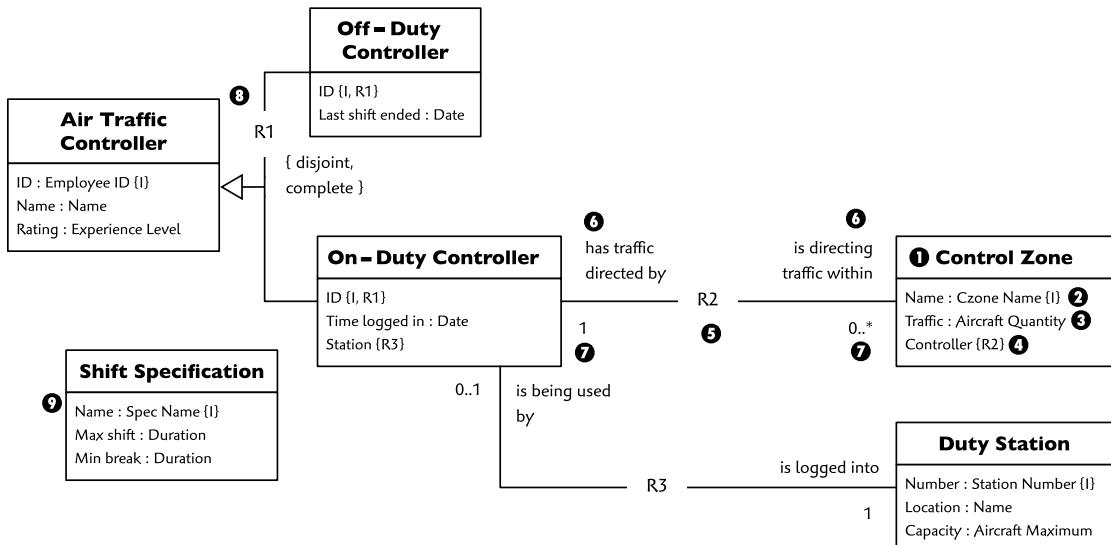


Figure 2-3. Annotated class diagram

❶ Each box represents a class. A *class* is an abstraction of a set of physical or hypothetical things having common behavior, common characteristics, and subject to the same rules and policies—in this example, a Control Zone. Each instance of a class is called an *object* or an *instance*. We will use the terms interchangeably. The second compartment contains a list of *attributes* that define the data required for each instance. Each Control Zone is described by a name, the amount of traffic, and the current controller.

❷ An *identifying attribute* is indicated by the tag `{!}`. The set of identifying attributes for a class comprises its *identifier*, which constrains the instances so that no two of them may have the same values for the identifying attributes. This also constrains the instances of a class to be a set. In this case, no two Control Zones may have the same Name. The identifier is often useful application data (the name of the Control Zone, for example), but sometimes it is entirely manufactured. When the model is translated, the identifiers may be transformed or substituted by something more appropriate to the implementation. As long as the identity constraint is met, the translation is free to transform the identifiers in any way.

❸ A *descriptive attribute* has a *data type*, shown after the name and a colon. We capture the type in application terms (for example, a whole number of airplanes between 0 and 200). After we have defined all the actions that operate on the attribute, we can decide how to represent the type in a program.

❹ A *referential attribute* is indicated by the tag `{R}`. This tells us that the attribute refers to another instance—in this case, the controller directing the Control Zone. By *refers*, we mean the value of the attribute in the instance has the same value as an attribute in another instance. The Control Zone.Controller attribute tells us which controller is monitoring a particular instance of Control Zone. However, the rules of the domain insist that the controller must be on duty. So, the value of the Control Zone.Controller referential attribute is constrained to match the value of an instance's On Duty Controller.ID. At implementation time, this may be implemented in many ways, such as a foreign-key constraint in a database or simply as a check on a pointer value.

❺ This line between the two boxes, labeled R2, is an *association relationship*. It represents the relationship that exists between On Duty Controller and Control Zone entities in the real world. Each association has a unique number—in this case, 2. By convention, we label the graphic with an *R*, followed by its number; for example, R2. The unique number is required because more than one association may exist between the same two classes (for example, a person may both drive one car and own other cars).

❻ These are *association phrases*. These verb phrases are textual annotations that tell us important semantic information about the relationship and provide a mnemonic for how to describe the real-world association. We can say that an On Duty Controller *is directing traffic within* a Control Zone. Similarly, there is a passive-voiced version of the relationship stating that a Control Zone *has traffic directed by* an On Duty Controller. The phrasing is constructed to read in both directions as a “subject—verb phrase—object” sentence with the verb phrase for a given direction being placed on the diagram nearer to the class that serves in the object role.

You may be more familiar with the UML style of using role names. We find, from extensive practice, that the verb phrase style is almost always more precise and expressive. Also, it is much easier to just say, “*is directing traffic within*,” than to contrive a role for a Control Zone. We also find that it is much easier to establish the correct multiplicity and conditionality by considering a verb phrase. In fact, the practice of carefully naming both sides of each association with precise verb phrases frequently exposes contradictions, ambiguities, and incompleteness in the source requirements.

- 7 We always further constrain an association by defining its *multiplicity* and *conditionality* for each side. The multiplicity specifies whether more than one instance may be involved in the association. The conditionality specifies whether an instance is required to participate in the association. The two concepts are combined into one graphical symbol, so, for example, “0..*” for R2 on the Control Zone side says that an On Duty Controller is directing traffic in zero or more Control Zones. The purpose of the multiplicity and conditionality of an association is to constrain the membership of the set of instances of a class.
- 8 This indicates a *generalization relationship*. The arrow points to the *superclass*, which is a generalization of all of its subclasses. In this example, an Air Traffic Controller can be either an Off Duty Controller or an On Duty Controller, and not both. A specific controller (Ianto, for example) will be both an Air Traffic Controller and an On Duty Controller. He is described then by attributes ID, Name, Rating, a Time logged in, and the Duty Station he currently controls. Our xUML interpretation of generalization is much more restrictive than that of conventional UML. We consider a generalization as a set partition and that the superclass instances are a disjoint union of all the subclass instances. UML would label this as {disjoint, complete}. Because we have only a single specific interpretation, we omit the annotation from here on. Further, we do *not* imply any notion of inheritance on the generalization. We treat the superclass and subclass instances separately. We interpret generalization relationships as meaning there is an unconditional, singular association between a subclass instance and a superclass instance, and an unconditional, singular association between a superclass instance and a subclass instance *from among all* the subclasses of the generalization.
- 9 This isn’t done often, but it is perfectly legal for a class not to be connected via any association. In this case, there is only one instance of Shift Specification that defines a couple of durations applicable to all Air Traffic Controllers. Because there is only one, it is easily selected without requiring any relationship traversal. (This would change if different groups of Air Traffic Controllers were subject to different break periods.)

There is a good deal more to be said about class models, and whole books have been written about them, such as *Executable UML: A Foundation for Model-Driven Architecture* by Stephen Mellor and Marc Balcer (Addison-Wesley Professional, 2002) and *Model Driven Architecture with Executable UML* by [Chris Raistrick](#) et al. (Cambridge University Press, 2004). Moreover, there is a good deal more to be said about how to go about constructing class models, and whole books have been written about that too. For example, *Executable UML: How to Build Class Models* by Leon Starr (Prentice Hall, 2001). We recommend that you read them, because a complete model must be constructed before translation, and translating a poor model yields an equally poor implementation.

Interpretation

The class model says a great deal about the application with just a few, well-defined elements. Class models provide a precise and unambiguous vocabulary with which to explain and discuss a problem. When we talk of a Control Zone, we mean the exact Control Zone declared on the model, as characterized by its attributes, and nothing else. It is Humpty Dumpty speak¹ that enforces the precision necessary for a meaningful discussion of problem logic.

¹“When I use a word”, Humpty Dumpty said in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.” “The question is,” said Alice, “whether you can make words mean so many different things.” “The question is,” said Humpty Dumpty, “which is to be master—that’s all.” —Lewis Carroll

For example, an On Duty Controller requires an ID, a time logged in, and a Duty Station to be working at, whereas an Off Duty Controller must have an ID and a date the last shift ended. The declaration of On and Off Duty Controllers allows us to differentiate between them, preventing an Off Duty Controller from being logged in, and requiring an Off Duty Controller to have a break. Obviously. But it would not have been obvious had we just plopped down an ATC with no subclasses. Such a model would have allowed an Off Duty Controller to direct traffic within Control Zones, even though, in the real world, this is prohibited.

The placement of attributes constrains our understanding of the meaning of a class, because each of its attributes must apply uniformly to *all* instances of that class. For example, the attribute Duty Station. Capacity places a maximum number of aircraft on each Duty Station, whereas an attribute Control Zone. Capacity would place the maximum on the number of aircraft controlled by the Control Zone, thus limiting the sum of the aircraft in *each* Control Zone, rather than *all* of them. A detail, certainly. But code is detailed, and the code for those two interpretations is quite different.

We also need to be certain we understand the precise meaning of each attribute. For example, what exactly is the meaning of Control Zone.Traffic? Is it the current number of aircraft in the zone? A maximum? Something altogether different? We must tack these down before we generate code, or the code we generate will be wrong. To capture this information, we must also write descriptions of each class, each attribute, and each association.

Associations impose constraints that hold throughout the operation of the system. For example, the multiplicity “0..*” on the Control Zone side of R2 tells us that an On Duty Controller logs in, and then takes over Control Zones, rather than being implicitly logged in after they take over the first Control Zone. It also means controllers must hand over their Control Zones before they log out. The multiplicity constrains the possible behavior of the associated things, which ties down their meanings more precisely.

These statements, the definitions of the classes, attribute placement, and the constraints imposed by multiplicity must be verified against the real world. If a controller can be considered on duty only when controlling at least one Control Zone, the model is wrong and it should be changed. Is it? We have to find out. We have to question and evaluate the implications of each element on the model systematically. The attribute Off Duty Controller.Last shift ended, for example, enforces adequate break time. What break time? That wasn’t in the preceding list of requirements! (It should have been: a minimum break time of 15 minutes is required.) And so on.

These rules apply when the system is in operation, not during initialization. When the system comes into existence, there will be no Duty Stations, controllers, nor zones, but it is useful to think of some classes as having *preexisting instances*. Controllers come and go, but Duty Stations can be considered to exist before the system comes into operation. Table 2-1 shows the instances of Duty Station that exist when the system starts running.

Table 2-1. Duty Station Instances

Number	Location	Capacity
DS1	Front	20
DS3	Center	30
DS2	Front	45

Similarly, Table 2-2 shows the preexisting instances of Control Zone.

Table 2-2. Control Zone Instances

Name	Traffic	Controller
SJC18C	30	ATC53
SFO37B	25	ATC53
OAK21C	15	ATC67

Here, we have represented the *initial instance population* of the Duty Station as a table, though, as usual, the implementation may be entirely different. For example, references to the Controller in the Control Zone table might be implemented by programming language constructs rather than the value comparison implied by a table.

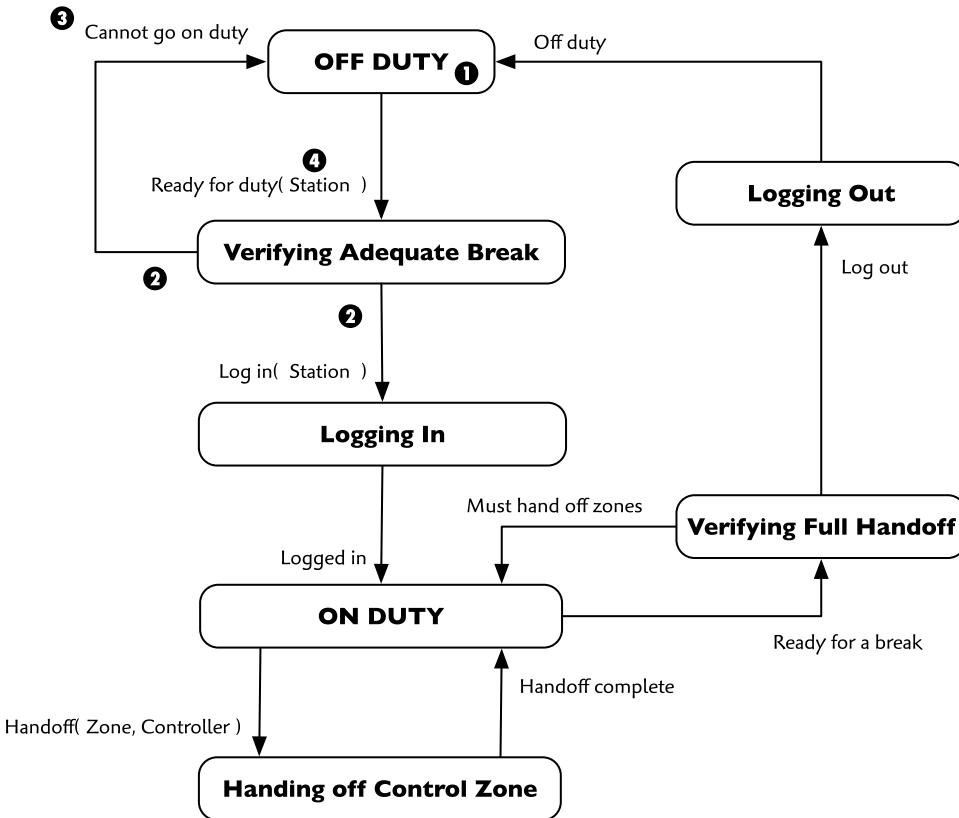
Step 2: State Models

The class model expresses behavioral constraints, but it does not define the specific behavior over time. Each instance of a class is subject to the same rules and constraints, behaving the same way over time. An Air Traffic Controller object will be off duty initially, log in at some point, become on duty for a period of time, and eventually log out again and return to being off duty. This *life cycle* common to all ATCs is captured with a state model.

A state model is defined on a class, and all instances of that class exhibit the same behavior. At any given time, each instance will be in its own state, with other instances in possibly different states. This is why we have distinguished between a state model and a state machine. A *state model* is the pattern of states and events associated with a class and describes the behavior of all the instances of that class. A *state machine* describes the behavior of a single instance that is governed by that instance's value of its current state.

Although we generally construct a state model for each class, we can omit the state model for a class that has no interesting behavior. Classes that simply come into existence and do not behave differently over time do not need a state model. In a generalization relationship, we have choices as to whether the state model is associated with the superclass or the subclasses. A complete set of rules and heuristics for their use can be found in *Executable UML: A Foundation for Model-Driven Architecture*.

Figure 2-4 shows the state model for the ATC. It also has been annotated so we can define our interpretation of the graphical symbols.

**Figure 2-4.** Air Traffic Controller life cycle

❶ The rounded rectangular boxes are *states*. For readers familiar with state models from electrical engineering and math texts, in those contexts individual states are usually represented as circles. The rounded rectangles notation is more convenient because we like to write inside the boxes. The convention of using all capital letters for some state names indicates that an instance remains in that state until an event is delivered from a class instance other than itself or one that has been requested for a future time. By contrast, an instance in a mixed case named state will exit on a transition as soon as the activity is completed. This is only a notation convention that we have found useful and is not part of the execution rules for a state model.

❷ The directed arrows represent *transitions*. An ATC who is Verifying Adequate Break may transition to either the Off Duty or Logging In state.

❸ Each transition is caused by an *event*. If the break is adequate, the ATC will log in to a station, but if not, that event triggers the transition back to the Off Duty state. Strictly speaking, a distinction can be made between an *event specification*, which defines an event name and parameter signature, and the occurrence of a corresponding event at runtime directed at a particular instance, with values filled in for any parameters. In practice, however, it is convenient to just say *event* for either case unless the meaning is not clear from the context. In UML, a *signal* is something you send, and an *event* is something you detect,

with several UML types of events defined. But, in xUML, we need only the type of event that is triggered by a signal. Consequently, in xUML, there is no distinction between *signal* and *event*, so we use the terms interchangeably.

- ④ Each event may carry *event parameters*. This is data carried along with the event that can be used by the actions associated with the state. When ATCs log in, they log in to a specific duty station. This is rendered as the event parameter Station.

Interpretation

A state machine behaves as follows:

- Each state machine is in exactly one state at a time.
- When an event occurs, a transition is triggered and the state machine moves to a new state. The new state may be the same as its previous state, but it is considered to have transitioned even if it arrives back at the same place.
- When entering a state, the state machine executes an *activity* comprising a particular number of actions (described in the next section).
- How long that takes is indeterminate, but the state machine does not respond to any further events until it finishes the activity and reaches the next state.
- When the state machine completes the activity of a state, it may respond to further events.

The last two bullets encapsulate a concept called *run to completion*: you finish what you're doing before you start doing anything else. From an analysis point of view, you need not worry that an activity might be interrupted. For example, a state activity can manipulate the data in the class model without being concerned about transient inconsistencies in the activity's processing. An activity can update longitude and latitude, for example, without having to worry about the data being misread halfway through the update. Run to completion ensures consistency of processing and simplifies the task of building models.

Beyond data consistency, this principle is critical to overall synchronization. To appreciate the importance of run to completion, consider what might happen if it were not true. Assume a hypothetical scenario in which a state activity generates a signal, and the dispatch of the corresponding event occurs as part of the signaling operation. The event dispatch might cause another state activity to execute, which in turn could signal back to the original sender. Again, if that event dispatch is executed synchronously to the signal generation, we would execute a new state activity in the instance before the original activity that generated the first signal completes. Such a situation might cause the underlying attribute values to differ, depending on whether they were accessed before or after signaling.

Events are neither saved nor lost; they are simply unavailable until the state machine settles in the next state and finishes its activity. Nor do we specify the mechanism whereby events are signaled or delivered. It is necessary only to presume that there is a pervasive, underlying means to execute the state machine event dispatch rules. The implementation of these rules can be accomplished in many ways, such as queuing an event until the state machine is able to respond. Later, we show exactly how this happens for our target execution environment. A summary of state, event, and activity synchronization rules is available in Appendix A.

There are two fundamental formulations of state models: in one, actions are associated with entering a state (Moore type model); and in the other, actions are associated with a transition (Mealy type model). Much ink, hot air, and electrons have been burned up in discussions of which formulation is better. In fact, a Mealy state model is easily converted into a Moore formulation, and vice versa, so there isn't anything that you can model with one style that cannot be accommodated by the other.

In xUML, we use the Moore formulation, which means that an activity is executed when a state machine enters a state. In UML, these are called *entry activities*, and these are the only type of UML activities we need. Again, because we have only one interpretation of activities, we omit the UML entry / reserved word. xUML uses the Moore formulation because it yields a more regular state table that is essential for both translation and verifying event-state coverage. Moore state models associate activities with states, and this is convenient both for model specification and translation.

The dynamic behavior of a collection of state machines is as follows:

- Each state machine is considered to be executing concurrently with respect to all the others. (Toshiko may be going off duty, after handing off all her Control Zones to Gwen, while Ianto is simultaneously logging in to Duty Station S3.)
- A state machine can access data synchronously from other objects. (Gwen can look at Ianto's last break time, irrespective of what Ianto is actually up to.)
- A state machine may send a signal to another state machine to cause it to change state.
- A state machine may respond to events sent as signals from other state machines, the outside world, or a signal requested some time in the past.

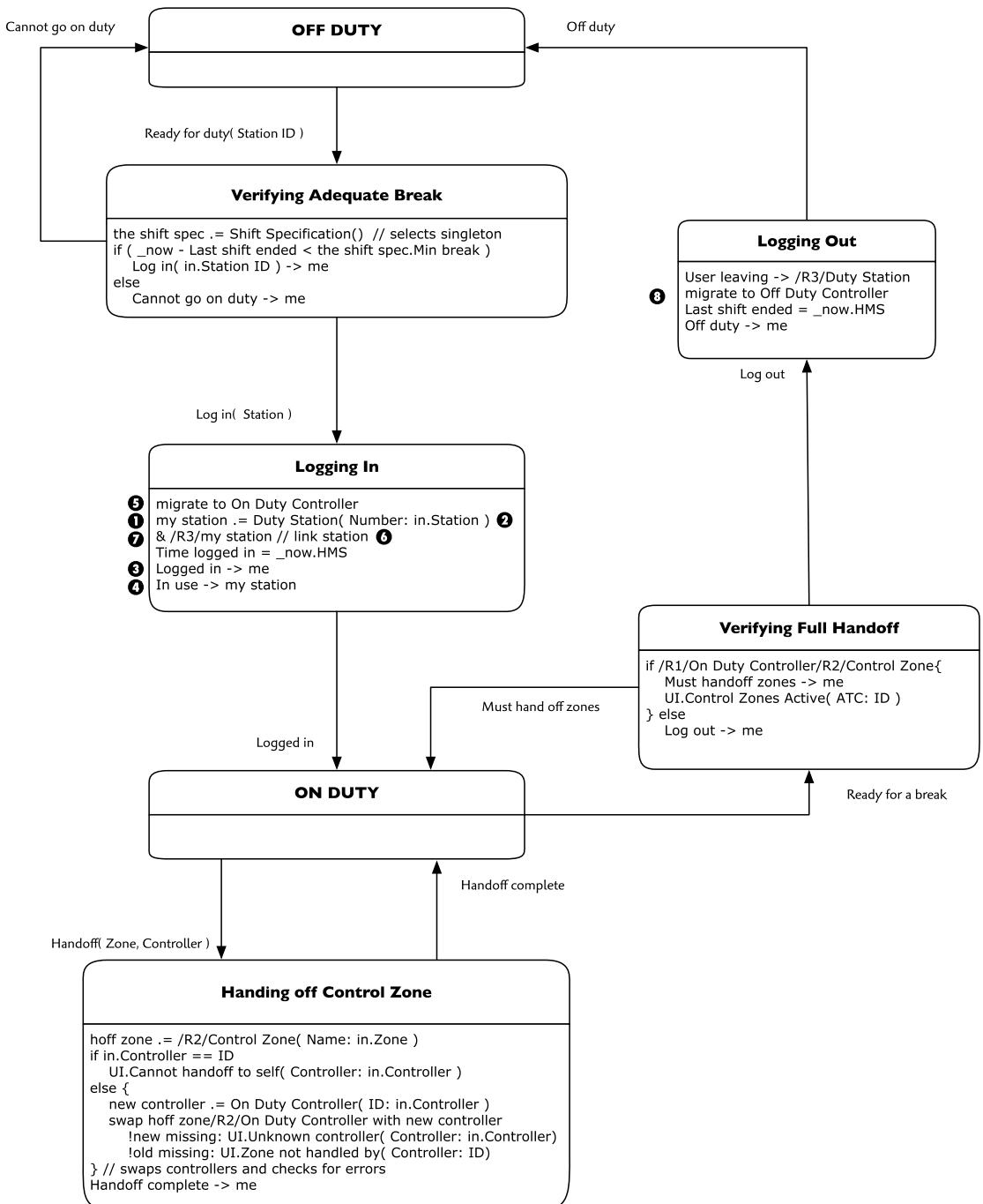
In this formulation, each object executes concurrently unless it is explicitly sequenced or synchronized with other objects. This requires the modeler to think through synchronization early, before committing to threads, tasks, processors, and so on. It also permits the implementation to be made more concurrent, because the models do not impose unnecessary or arbitrary sequencing.

Step 3: Actions

At some point, the rubber must meet the road; we must actually *compute* something. Each *activity* comprises a certain number of *actions* that must complete execution before the activity is completed.

Actions are expressed using an *action language*. There are many possible action languages; they have (almost) the same underlying building blocks. They must all, for example, provide a means to traverse associations, select and access instance data, communicate between objects, invoke external services and libraries, and perform computations on selected model data.

Our action language is designed to be easy to write, with a minimum of language elements, while remaining readable, which is a key purpose of modeling in the first place. A summary is provided in Appendix B. We discuss the pros and cons of this language later in this chapter. For now, let's look at how it is used in our ATC example. Figure 2-5 shows the state model again, this time with actions filled in. Again, we have included annotation to explain our interpretation.

**Figure 2-5.** Air Traffic Controller actions

- ❶** `. =` is an assignment. In this example, `my_station` is an instance reference variable, valid only within the activity where it is initialized. The single `.` in the `. =` operator limits the selection to at most one instance.
- ❷** The `<class>(<attribute>:<value> ...)` syntax uses the criteria in the parentheses to find matching objects. Because an identifier attribute is used in this example, one object, at most, will be found. Hence, `Duty_Station(ID:in.Station)` returns a reference to the Duty Station object that matches the Station number that was passed in.
- ❸** The `->` symbol specifies the immediate signaling of an event directed toward a set of instances, typically one. In this case, the Air Traffic Controller object sends the `Logged in` signal to itself so that it is not permanently stuck in the `Logging In` state. The `me` keyword serves as a reference to the local object (`self`). Note that although the signal is immediately sent, the event will not be processed until the entire activity associated with the `Logging In` state has completed.
- ❹** In this case, the target of the signal is the related Duty Station object.
- ❺** The `migrate` action changes the subclass of an object. Formally, this involves dissolving the generalization from the subclass instance, creating the new subclass, and reforming the relationship to the superclass. In practice, implementations find ways to store subclasses so that the formality is met efficiently (for example, using a union to store the subclass would mean that we could reuse existing memory space rather than creating a new subclass object).
- ❻** To refer to data arriving with an event, preface it with the `in` keyword. In this example, `in.Station` refers to the station number passed in with the event.
- ❼** The `&` and `!&` link/unlink operators create and delete an instance of the association. In this example, `& /R3/my_station` relates itself to the station referred to in `my_station` across R3. The `/` is the hop operator, which specifies a hop across a relationship. When it is not preceded by any explicit instance, navigation is assumed to start with the local instance. Note that linking/unlinking is a conceptual operation for the modeler. There are no "links" as such in xUML. What's really going on is that the link operator is setting the local instance's `Station` attribute to the value of `my_station.ID`. We could just as easily have written `Station = Duty_Station(Number:in.Station).Number`, but such a statement risks overlooking the concept of an association being instantiated.
- ❽** There is no need to explicitly unlink the Duty Station, because an Off Duty Controller has no `Station` referential attribute and the On Duty Controller instance has been deleted.

Time and Other Details

Figure 2-6 shows the Duty Station state model and illustrates how to manage time.

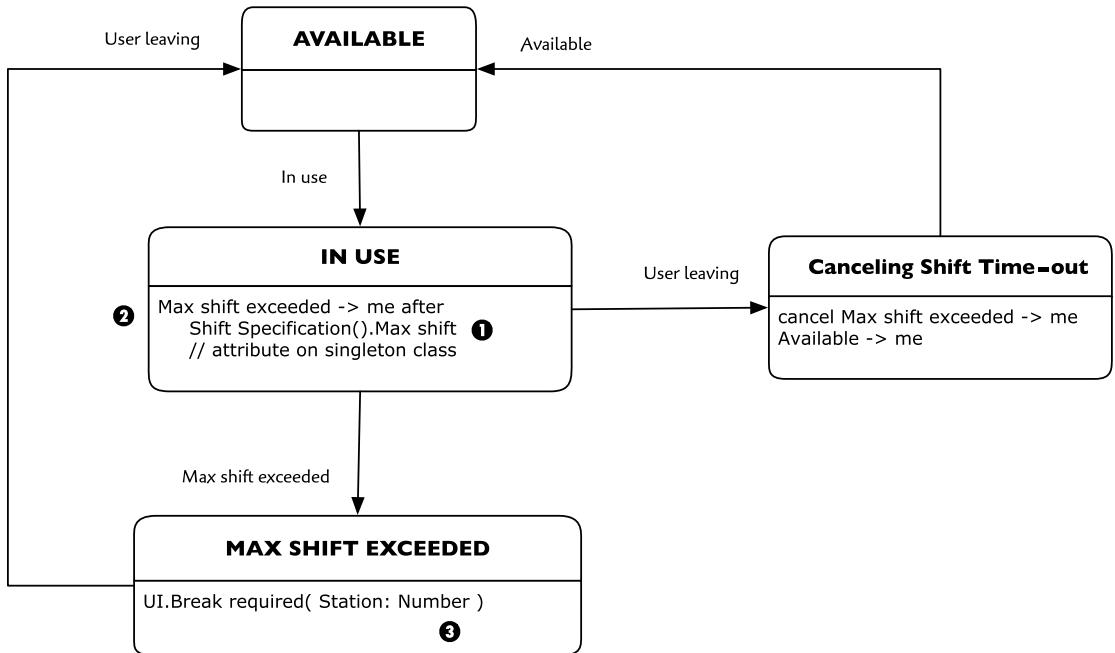


Figure 2-6. Duty Station life cycle

There are a couple of actions to note here:

- ① In this example, the Shift Specification singleton class holds a single instance with a delay value. The extra step of first assigning the singleton instance to an instance variable is instead folded into a single statement.
- ② By default, a signal is immediately sent, but a delay may be specified. The delayed signal serves as a time-out, so it is canceled if the Duty Station logs out within the maximum on-duty duration.
- ③ UI refers not to a class, but to an external entity. An *external entity* is a proxy for something outside the domain boundary. In this case, the ATC domain assumes that there is a user interface (UI) that can post the warning message.

Discussion

There are many good ways to specify algorithmic computation. Figure 2-7 shows a data-flow representation of the Logging In activity.

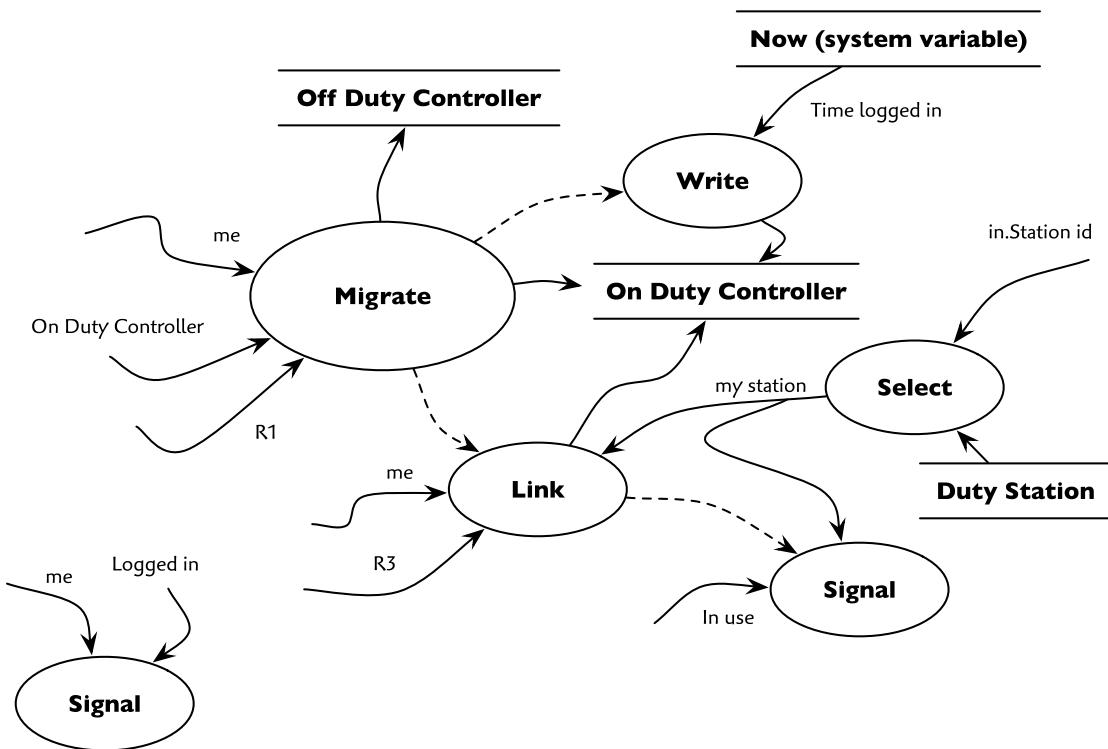


Figure 2-7. Data flow diagramming an activity

Each oval in the diagram represents an action. An action may execute when all of its inputs become available. Data from sourceless arrows and class data stores (parallel lines) are available upon state entry. Dashed arrows provide no data, but pass a control token to signal completion of the source action. The Migrate, Logged In Signal, and Select actions can execute immediately upon state entry. The Write and Link actions must wait until Migrate is finished. Finally, the In Use Signal action may execute.

Data-flow notation specifies sequencing only where it is essential to the problem space. It therefore highlights opportunities for concurrent processing in the implementation. On the downside, data-flow diagrams can be difficult to draw and edit, and they may make your brain hurt as you try to comprehend sequencing.

Text-based action languages are much easier and more familiar to read, write, and edit. You can rely on the familiarity and power of your favorite text editor to get the job done. Some textual action languages make no attempt to eliminate arbitrary sequencing. Others mimic data-flow concurrency by using text symbols rather than graphical symbols. In this book, we use Scall as a pseudo-action language, as it is as easy to edit as text and can be mapped to a data-flow representation for downstream translation. Its main elements are summarized in Appendix B.

How much of the inherent concurrency of a state activity is realized as parallel execution depends on the translation mechanism and the target platform. We are targeting a platform that has only a single processor core, so no real parallel execution is possible. A more capable target platform allows the translation mechanism to arrange how the concurrency in the model is realized as parallel execution.

Why not just write the actions in C directly? The whole point of this type of modeling is to capture the logic of the problem in an implementation-independent fashion. We want to specify actions in terms of non-C concepts such as object state machine communication, data access via compound relationships, and instance set manipulation. By keeping all model facets, class, state, and actions at the same level of

abstraction, we have a single consistent set of rules and increase the opportunities during translation to choose a wider variety of efficient implementations. This also makes the modeling easier by concentrating on the logic of the processing without having to consider all the details of programming language syntax and semantics.

Executing the Model

Now that all three facets of our example application are modeled, we can execute them as an integrated unit to verify correct behavior. This can (and should) be done prior to writing or generating any code. A complete description of the execution rules can be found in the Executable UML books mentioned earlier in this chapter. They are also summarized in Appendix A. Here, we just want to give you a feel for how the models can be executed by walking through a simple scenario.

For this scenario, we will attempt to activate an Off Duty Controller by logging in to an available Duty Station. Assume that ATC53 is in the Off Duty state and that the Duty Station to be selected, DS3, is waiting in the Available state. A Ready for Duty(DS3) event will be addressed to ATC53 to kick things off. The value DS3 is passed as the desired Duty Station number. The choice of Duty Station happens outside our system, and our models are simply told that ATC53 is attempting to use DS3. Our models must, however, ensure that certain conditions are fulfilled before allowing this to happen.

Upon receipt of the event, ATC53 makes a transition to arrive in the Verifying Adequate Break state and executes the activity upon entry into the state. The Verifying Adequate Break activity checks to see whether a sufficiently long break has been taken. Assuming it has, ATC53 signals a Log in(DS3) event to itself. Having completed the activity, ATC53 then receives that same Log in event, matches it to an outgoing transition, and proceeds to the Logging In state.

ATC53 executes the Logging In activity that results in its migration from off to on duty (delete off-duty subclass instance and create on-duty subclass instance referring to the same superclass instance), linking of the new ATC53 on-duty subclass instance to DS3, and logging the current time and the signaling, this time of two events: ATC53 signals a Logged in event to itself and sends an In use event to DS3. The signaling occurs in no particular order, as would be obvious only from the DFD.

Now we have two pending events, each of which will be consumed, again, in no required order. When the Logged in event is processed by ATC53, it transitions to the On Duty state. When DS3 consumes the In Use event, it transitions to the In Use state and addresses a Max shift exceeded event to itself, delayed by the Max shift duration as indicated in the singleton Shift Specification instance. In essence, a time-out is established so that if the On Duty ATC does not take a break, the delayed event will fire and trigger a warning in the Max Shift Exceeded state.

At this point, the scenario is complete. ATC53 has successfully logged in at Duty Station DS3. Notice that the state models and actions specify behavior for nonspecific instances that are interpreted during runtime by individual instances consuming events and traversing from one state to the next.

Our scenario conveniently sidesteps the possibility of an instance consuming an event in a given state for which no transition is specified. This situation is resolved through the construction of a state table, which is required for each state model. A state table shows how an instance reacts if an unexpected event arrives, or if an expected event arrives either when the instance is ready for the event or when it is not ready for the event. Chapter 3 covers state tables in detail.

Standard Action Languages

We distinguished between elaboration and translation approaches in Chapter 1. When the UML came into existence, elaboration was king—no one saw the need for an action language. But a few people persevered, and some years later, UML finally has an action language, the Action Language for Foundational UML, or Alf. The specification can be found on the Object Management Group website (www.omg.org).

The specification is some 439 pages long—longer than this book. We have no wish to teach you the details of that language, or any other language for that matter. Rather, we wish to use the concepts of an action language to show how translation works. Accordingly, we have used an invented informal language that we hope can be understood intuitively with minimum effort. We describe this language in Appendix B.

Using an informal language for execution is clearly peculiar, at best. But the alternative is to teach you another language, which is not the purpose of this book. Moreover, we wish to expose the steps required to get from models to code. Consequently, we formalize the actions in later chapters. It may look as if we are writing the code twice. We're not. We're simply bypassing the details of a formal language.

Summary

We've introduced a simple application as a fuzzy and incomplete set of requirements and cast it into a concrete and executable set of models. The models are executable with a lean set of execution rules, and those rules make it possible to step through and test application scenarios in much the same way you would step through running code. The execution rules and semantics are organized into three facets: a single class model, a state model for each class, and interesting behavior and actions within each state corresponding to modeled data, modeled control, and modeled computation, respectively. The models are constructed in this sequence to minimize refactoring.

What we have *not* done is include in this model any implementation concepts. We have coordinated state machines by sending signals, but we have not said how those signals are sent; we've said we have associations between instances of classes, but we have not said how they are implemented either. As we said in Chapter 1, *exclusion* is neither the removal of that detail, nor the deprecation of it. The implementation is critically important; it is the subject of this book.

From here on in, our focus is on translation. We will map these model elements, along with the execution behavior defined by the modeling language, into a platform-specific bundle of C code that will run on a computing device supported by our target platform. When we focus on code production, we will not question the application requirements or try to expand or modify the application scope. Instead, we'll assume that the modelers knew what they were doing, code the models as they were given to us, get it running in a testable environment, and move on to the next application. If the models don't cover the requirements, neither will the code!

CHAPTER 3



Making Translation Decisions

In the previous chapter, you saw how vague statements can be transformed into a concrete and unambiguous specification of the problem logic realized as a set of executable models. The models contain problem-oriented concepts that you can discuss with your customers and users to make informed decisions about the relevant problem logic. Although completed models define exactly *what* the implementation should do, they do not say just *how* to do it.

Now we must decide how to translate the various model elements into code:

- How are classes and instances represented in the program?
- How will the identifier attributes be realized in memory?
- How will instances be accessed?
- How will relationships be navigated?
- How will events be signaled and dispatched?

And so on. We must ensure that every model element is translated into code to realize all of the required functionality. And foremost, we must end up with a functioning and efficient implementation. When models are being created, we are focused on analyzing the requirements and capturing the logic of the problem in executable modeling constructs. When models are translated, we must analyze the models themselves to make decisions about the appropriate code constructs to implement the model logic.

This chapter serves as an introduction and overview of the key decisions that must be made when translating the ATC model. In the subsequent chapter, we get our hands dirty and start building a pycca model script to specify those decisions. For now, though, we just take an inventory of what needs to be done.

The types of decisions discussed here are specific to both our target platform and to the way pycca accomplishes a model translation. This does not imply that the process of translation is unique to our platform and methods. Different platforms necessarily require varied strategies for the implementation and, consequently, a distinct mapping from model elements to implementation. Nonetheless, we continue to focus on pycca and our particular target platform to illustrate how our example becomes a running program.

Reviewing the Target Platform

The target platform sets boundaries for a practical implementation. Our target of small, embedded microcomputers, for example, leads us to hold all of our application data in directly addressable memory. Because we intend to use C as an implementation language, and C exposes variable addresses, we have decided to use the memory address of an instance as an implementation-based identifier. Where possible, we replace each identifier from the model with this implementation-defined identifier. For example, *Duty Station.Number* is replaced with its instance's memory address. Using a pointer-based identifier then allows us to use pointer values to navigate relationships, resulting in many implementation efficiencies.

Our platform uses C as the implementation language, but the same principles can be applied to any other statically typed language such as Java. In that event, a Java-specific translator with compatible runtime libraries (*pyjca*, let's say) would be required. But that's a different book (bonus points if you get the acronym *pajama* to work).

Figure 3-1 illustrates our overall process leading from models to code, using *pycca*. The model is transformed in two steps.

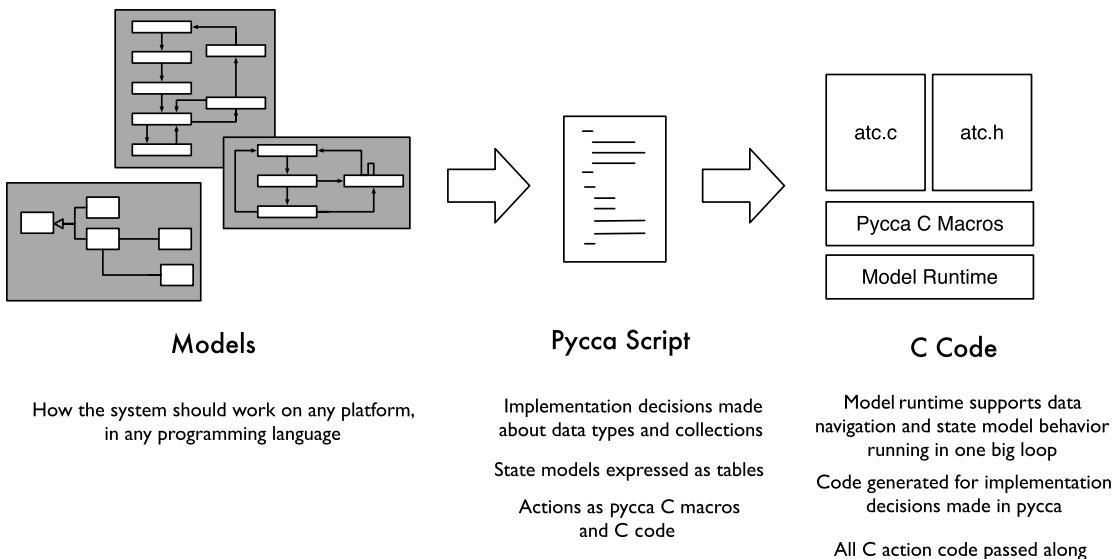


Figure 3-1. Overview of *pycca* translation

The first step is to encode all of the model elements, from all three facets, as a series of *pycca* language statements in a single file. We call this the *model script*. This script captures the structural aspects of the models as well as the processing they perform. This script incorporates, along with the model elements, numerous design decisions. You also include C code in the script at a later point. (*pycca* is an acronym for “pass your C code along,” and that’s the C you’ll be passing along.) The set of available *pycca* statements defines a domain-specific language (DSL) for an xUML model implementation. We’ll introduce the language as the example is worked out and guide you through the key concepts. If you wish to read all the details about *pycca*, they are readily available in online materials. The DSL is summarized in Appendix C.

When your model script is complete, you feed it to the *pycca* program, which transforms any *pycca* statements along with included C code for the activities into a specially organized source and header file pair. The C code for the activities contains supplied preprocessor macro invocations to interface to the runtime code. From here, the C preprocessor, compiler, and linker do all the work: the *pycca* C macros are expanded and *pycca*-supplied runtime libraries are linked to yield a runnable program.

When modeling, we consider the three facets of data, behavior, and processing, in that order. When translating, we follow the same order. Translation decisions for implementing the data facet of the model impact the other facets, so it is important to pin down the application data structures first. In the following sections, we show the types of implementation decisions that must be made to translate each facet of the model.

Working with the Class Model

We first script the entire structure of the class model. This consists of the classes, attributes, data types, and all relationships. We also populate the structure by specifying an initial instance population. Figure 3-2 shows the pycc script elements that are derived from the class model.

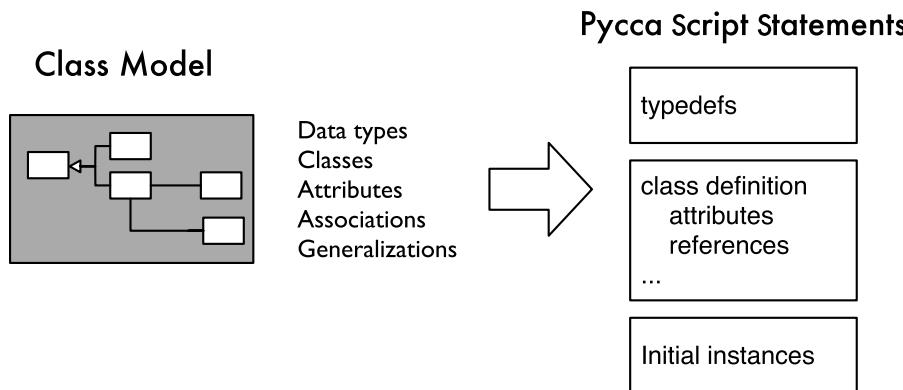


Figure 3-2. Defining the class model as a script

A class is made up of attributes, and each attribute is constrained by a data type. Data types, then, are essential building blocks that must be defined before we can proceed with the rest of the class model.

Data Types

Figure 3-3 shows the transformation of model data types to implementation types.

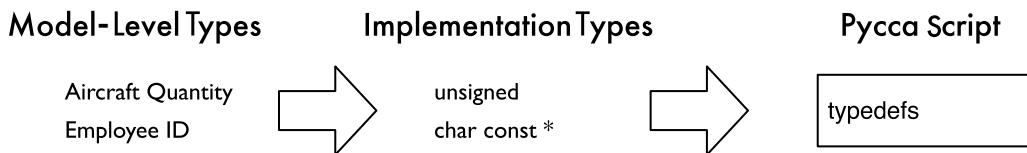


Figure 3-3. Data type decisions

A model data type defines the set of values that may be assigned to a class attribute. Aircraft Quantity is a model-level concept that represents an actual quantity of aircraft. This means that it is a positive integer, because it is nonsensical to talk about negative quantities of aircraft. An implementation data type defines a set of allowable values for a programming language variable. The C language data type corresponding to Aircraft Quantity would probably be `unsigned`. Be sure to choose an implementation data type that supports all operations required by the modeled data type.

C `typedef` statements bind each model data type with its associated implementation data type. These `typedef` statements can then be included at the top of the generated code file.

Classes and Attributes

Each modeled class is specified with a `pycc` `class` statement. Just as a model data type is different from an implementation data type, a class in the model is different from its corresponding implementation class. Because `pycc` converts each `class` statement into a C struct, there are no classes in the generated code at all.

Figure 3-4 shows the transformation from modeled class to C structure.

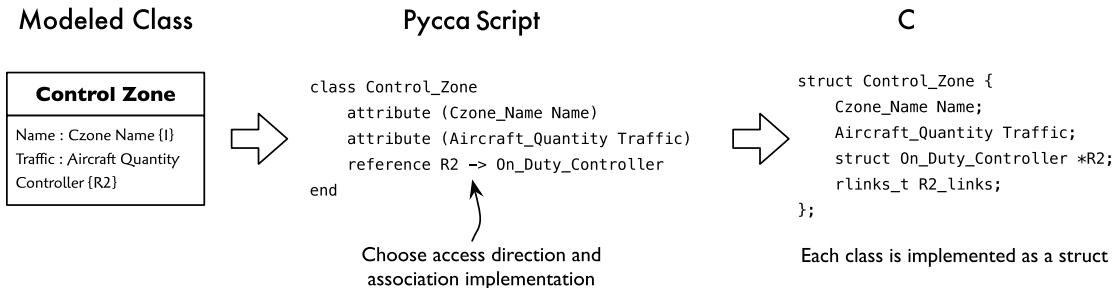


Figure 3-4. Translating model classes to C structures

The `class` statement encloses a series of `attribute` and `reference` statements. The descriptive, identifier, and referential roles of each attribute require different implementations.

A descriptive attribute, such as `Control Zone.Traffic`, has real-world meaning. Its implementation data type defines the set of values consistent with this real-world meaning. An identifying attribute, such as `Control Zone.Name`, which is used in a descriptive manner, is handled the same way.

By contrast, an identifier attribute with no real-world meaning, such as `Duty Station.Number`, is simply deleted. `Pycc` creates an implementation-defined identifier for each class instance. If a model identifier is not used for any descriptive purposes, we can substitute the generated identifier for it. This technique is so commonly used in implementation languages that we tend to overlook the fact that we are substituting one identifier for another.

Associations

A referential attribute, such as `Control Zone.Controller`, is handled in the context of associations. The specification of associations and referential attributes requires more consideration because a significant gap exists between the “this class is related to that class” model-level abstraction, and the implementation collection and pointer traversal mechanisms. The model purposely leaves many choices up to the implementer. We could, for example, choose to use a linked list or array traversal mechanism for a particular association. Or a self-balancing tree. To decide, we need to examine the runtime characteristics of each association, and then encode that decision in a `pycc` statement and let the `pycc` processor generate the relevant C code.

When an instance must refer to a single related object, the simplest way is to map a referential attribute to a pointer. Because referential attributes refer to identifying attributes of the associated class, and because we are using the address of the instance as an implementation-defined identifier, storing a pointer value in the place of a referential attribute realizes the model-level association.

This simple approach works well when following the pointer, but would be cumbersome in the other direction, because this would mean we’d have to search the objects of a class looking for matching pointer values. For the price of the memory occupied by a pointer, storing a pointer in both classes makes following in both directions efficient. On the other hand, the single-sided reference approach would be sufficient and efficient if there were no accesses from the “one side” to the “other side.”

So, we approach the implementation of an association by examining the needs of each side and considering each model association as two independent access directions. Figure 3-5 shows how the association, R2, is decomposed into one direction referring from On Duty Controller to Control Zone, and one referring in the opposite direction from Control Zone to On Duty Controller.

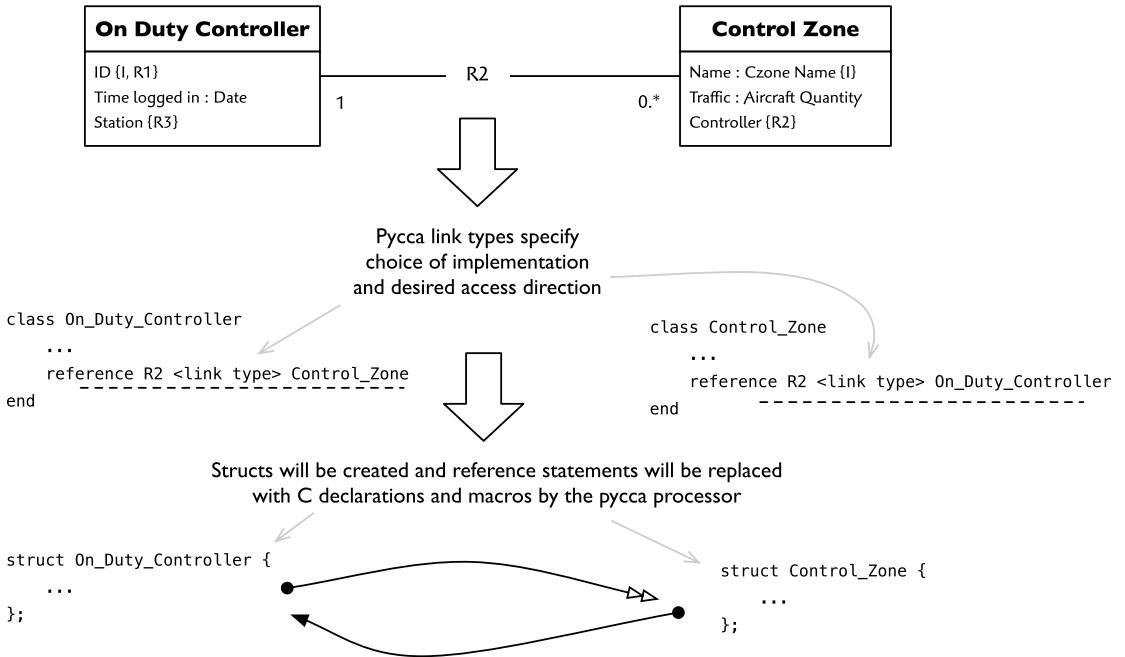


Figure 3-5. Decomposing a model association

Then we decide on an implementation for each direction of the association based on direction of access, multiplicity, and changeability:

- **Access:** Is the association accessed from one side only? Or from both sides? Storage for a reference is required only for the side that is accessed. What is the frequency of that access?
- **Multiplicity:** How many instances participate in the association from each side? A multiplicity of one is implemented as a single pointer value. If the multiplicity is greater than one, multiple pointer values must be stored, and pyccia gives us several common arrangements for that storage.
- **Changeability:** Are the participating instances static, or do they change over time? If the instances participating in the association change, we will store the pointer values in data structures that are efficient to update.

When we translate the model, we decide on an implementation for each side of the association. So for the preceding example, we may choose to implement the association from the Control Zone to the On Duty Controller as a pointer, while a linked list would be preferable in the direction of On Duty Controller to Control Zone. We then encode those decisions in text by using the proper reference statement types. There are many more choices, but the main point is that you must decide on an implementation while encoding the model, and that decision should be based on the execution characteristics of the model.

Generalizations

In our interpretation of a generalization, each instance of a subclass refers to exactly one instance of the superclass, and each superclass instance refers to exactly one instance of a subclass *from among all the subclasses* of the generalization. This implies that when we traverse a relationship from the superclass instance to a particular subclass, we either find a single related instance, or we come up empty because the superclass instance is not actually related to an instance of that particular subclass. To implement the traversal from a superclass to a subclass, pycca uses a subtype statement as a part of the superclass class definition. The result is that pycca adds an extra member to the generated C structure that encodes the related subclass. This lets us navigate the generalization from the superclass to a subclass and know whether we have found a related instance.

Storage for the subclass objects can be implemented in one of two ways. Figure 3-6 shows the two alternatives.

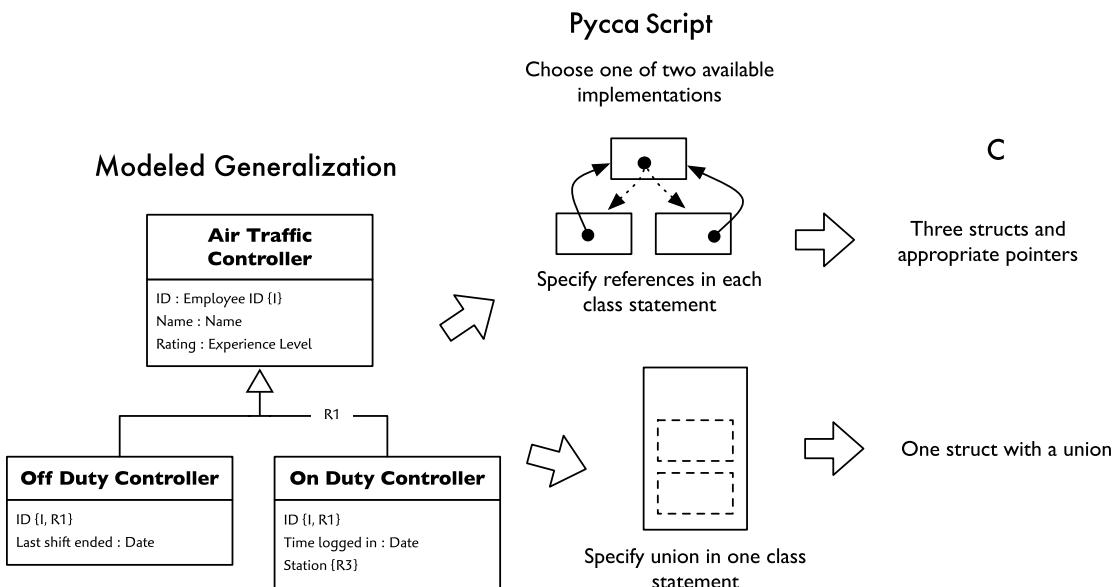


Figure 3-6. Generalization implementation alternatives

One approach is to treat each sub/superclass relationship as a bidirectional association. In this formulation, each On and Off Duty Controller instance would require a reference to its corresponding Air Traffic Controller instance. The subclass statement in Air Traffic Controller would be declared as a reference subclass, and a pointer to the subclass would be generated as another ATC structure member.

The other approach is to store the related subclass as a member of a C union. In this formulation, the subclass statement in the ATC superclass would be declared to be of the union type. Pycca generates the structure of the superclass to have a C union member containing the structures of the subclasses of the generalization. Space for the subclass instance is then included as part of the space for the superclass instance. This technique is quite common in C to implement a form of inheritance, even though inheritance is not our intent here. The subclass instances no longer need a reference to the superclass, and navigating the generalization is done via pointer arithmetic rather than pointer indirection.

Both approaches have their tradeoffs. For simple cases, the union formulation is more space-efficient.

Initial Instance Population

The last part of data translation is to specify the initial instance population. These are the instances of a class that are to be in place when the system starts. It is analogous to specifying the starting state for a state model. As the system evolves in time, attribute values change from known starting values.

Our strategy is to place the initial instance data directly into memory as variable initializers so that no code is required to create instances at initialization time. There is usually a small amount of initialization code that copies data initializers into their proper memory location. This behavior is a guarantee of the C language, and the code is usually supplied as a compiler library and is run before `main` is invoked. This eliminates wasting memory on code that will be executed only once. And we also avoid having to write tedious sequences of code just to set the values of structure members. Just as important, the initial instance population gives us insights prior to runtime for generating more-efficient code.

The initial population of the domain was specified as a table in Chapter 2. We can encode the table for each class, and store the instances of that class as an initialized C array. Descriptive attributes are assigned to the appropriate structure member in the initializer for the array element as shown:

table Control_Zone			
	(Czone_Name Name)	(Aircraft_quantity Traffic)	R2
@sfo	{"SF037B"	{27}	-> atc53
@oak	{"OAK21C"	{18}	-> atc67
@sjc	{"SJC18C"}	{9}	-> atc51
end			

Because we have generated an implementation-defined identifier (namely, the address of the object in memory) for each class instance, we need a way to refer to instances to specify the association and generalization pointer values. We cannot use actual addresses, because they are not known until link time. So pycca allows you to associate an optional name with the @ symbol on the left and to use that name when specifying the reference values. Pycca keeps track of where the named instances reside in the allocated storage array and emits the proper address expressions as initializers. Naming an instance is optional, but if you don't name it, you can't refer to it later.

Pycca also allows you to specify whether a class population is `static` or `dynamic`, indicating whether instances are created or deleted at runtime. A `static` population is allocated only enough space to hold the initial instance population. For a `dynamic` population, the total allocated space holds both the initial instance population plus the number of additional slots declared for it. All classes have a fixed-size storage pool allocated to them at compile time. The runtime library manages each pool for dynamic allocation, but there may never be more class instances than allowed by the size of the storage pool. This means that class instances are never allocated on the system heap, and there are no calls to `malloc` by the runtime library. This is in keeping with the usual design conventions for embedded microcontrollers.

Populations can also be marked as `constant`. For embedded systems, memory is usually partitioned into read-only and read/write types. There is usually much more read-only memory than read/write memory. A `constant` population is placed in read-only memory, which means that its attributes cannot be updated and that it cannot have an associated state model. Model classes that contain only specification data usually can be marked as `constant` and so save RAM memory.

Describing the State Models

We depicted state models graphically in the previous chapter, because the diagrams make it easier to visualize and achieve a well-designed solution. But this representation is incomplete for testing, implementation, and error analysis because a state diagram highlights expected normal behavior. The diagram does not show how to process an event that occurs and does *not* trigger a transition. When implemented, state machines have an annoying tendency to execute those unexpected transitions. Does the unexpected event require an error condition? Or was the event anticipated, but not warranting a transition?

There are three possible responses when an event is dispatched:

- A transition to a new state occurs.
- The event is ignored.
- It is a logical impossibility for the event to occur in the given state.

The first response, a transition, represents common, expected behavior.

Sometimes the best response to events, like children, is to ignore them. Events may arrive too late or too early. Consider the response to an Open button pressed after an elevator door is already open. An ignored event is in the realm of normal behavior and does not constitute an error of any sort. It simply does not merit a transition, and is signified by an IG (ignore) response.

The third response is quite different. It says that there is no logical way the event could occur given the current state and is signified by a CH (can't happen) response. If it does occur, it is deemed a serious error. When the runtime code encounters a CH transition, it causes a system error. It is important not to interpret a CH transition as a “shouldn’t happen” or “doesn’t happen very often” situation. A CH transition means that it logically cannot under any circumstance happen and, if it does, we no longer know how to proceed. Regardless of how small the probability that an event may occur in a state, it is *not* a CH transition if the probability is nonzero.

Figure 3-7 shows the progression from state diagram to its translated code. A state table specifies a definitive response (one of the three indicated previously) for every event in each state. In the first two cases, transition and ignore indicate that the response can and should be modeled. But with a CH response, the modeler indicates that the model is not designed to handle that case.

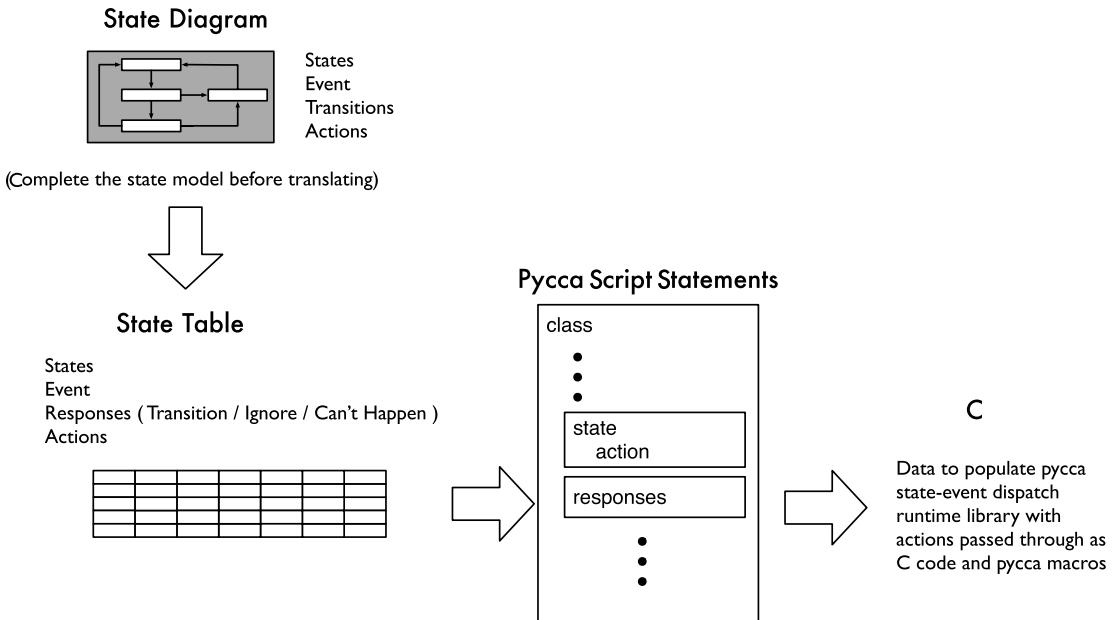


Figure 3-7. Translating a state model

We now specify each state model state by using a pyccua machine statement. Each state and its activity code is specified by using a state statement. Each state table cell is specified by using a transition statement. A transition statement records the new state into which an event causes a transition from the object's current state.

```

class Air_Traffic_Controller
    #
    # attribute and reference statements
    #

    machine
        #
        # state and transition statements
        #
    end
end

```

States

Each state has an activity associated with it, which we capture as a function. Because the activity is executed on state entry, all inbound transitions to the state must carry the same event data, which is passed in as parameters.

Each state and its associated activity, executed upon entry into the state, is represented with a state statement:

```

state Verifying_Adequate_Break(Station_number Station_id)
{
    // C code intermixed with C macros that perform common
    // model-level operations such as signal generation and
    // relationship navigation.
}

```

We cover the activity specification in Chapter 4, but you can see here that the activity in a state is enclosed within a state statement.

Events, Transitions, and Responses

State transitions specify responses to events by the state model:

```
transition OFF_DUTY - Ready_for_duty -> Verifying_Adequate_Break
```

This statement says that, when in the OFF_DUTY state, the Ready_for_duty event causes a transition to the Verifying_Adequate_Break state. There are no pycca statements for events. Their names are taken from the transition statements.

Executing State Machines

When we translate class models, some aspects of data operations, such as access to an attribute, are directly supported by corresponding C language constructs. State machines, on the other hand, are not a native C construct. To execute them, we must provide additional code, also written in C, to enable state machine behavior such as dispatching signals and calling state activities when transitions are invoked.

The transition statements encode state models statically, and we need runtime code to use the generated data structures to cause transitions. When a signal is dispatched, the runtime code uses the event signaled and the current state to determine the new state. Then, by calling a single function, it executes the logic of the new state. In Chapter 5, we show exactly how the runtime code signals and dispatches events.

Translating Processing

The last step is to translate the processing expressed by the action language of the model into C. Translation does not add to the processing logic in any way. We are concerned only with producing the C code that is logically equivalent to the processing specified by the action language statements. All of the processing is expressed by action language. When translated into C, a state's actions are packaged by pycca into a function that is invoked upon state entry.

Coding from Models

The strategy is to take each line of action language and write the corresponding C. As we analyze the actions, we'll find that that the elements fall into a small handful of categories. These are typically model-execution functions, data-access operations, and standard mathematical expressions and flow-control statements.

An action language statement may invoke an aspect of the model execution architecture. These include signal generation and relationship traversal. These types of elements may be translated by substitution with one of the many pycca-provided C macros. These macros hide decisions about data structures and just how a signal might be implemented.

Data access is provided by the normal C mechanism for access to structure members via a pointer. To realize this, each state action has a `self` variable that points to the instance memory. Action code can find and manipulate data across an association by using macros such as `link` and `unlink`, which hide the association implementation.

State action code must also interface to the runtime event dispatcher to generate events. We provide a set of helper C preprocessor macros so that we can use the same names for events as used in the state model. Pycca generates preprocessor symbol definitions to encode the state and event names into numbers. The naming conventions of that encoding are hidden by a set of preprocessor macros so you can use the names in the state model definition even though we must ultimately produce something that the C compiler recognizes.

All other actions, such as computation and flow control, are coded in C statements that are passed directly through to the output code file.

Translating a Model

There is a method at work here, and the order of doing the tasks has been worked out to give the best results. In the next chapter, we show the translation of the ATC model by using the decision-making process described in this chapter. We repeat the steps taken here, substituting the specifics of the ATC model.

Summary

We have examined the key translation decisions that must be specified in a pycca model script so that the appropriate C code can be generated and, in some cases, passed along.

Class models are broken into data type, class, attribute, and relationship definitions. We convert class definitions into pycca class definitions that end up as C structure declarations. Initial instance populations are specified in tabular form, to be converted into a set of array initializers.

State models, expressed as tables, are specified as a collection of state and event/response statements. State models define data structure values, including a state table, used by the runtime event dispatch mechanisms. Each state activity is converted into a function.

The correspondence between the encoded source and C output is direct and predictable. Pycca generates the many kinds of arbitrary encodings that are required, such as encoding event names into integers; it calculates the pointer values for relationship references, and it orders the resulting code in a way that suits the C compiler. The results are an ordinary C code and header file pair that can be integrated into the system build.

CHAPTER 4



Translating the Air Traffic Control Model

We took a tour of the key decisions made during translation of an xUML model in the previous chapter. Now, we get busy making those decisions for the ATC example introduced in Chapter 2. A pycca model script is prepared, and we examine the resulting C code. The subject of this chapter is implementation oriented, and we assume you are familiar with the C language. Space doesn't permit us to present all of the code here, but if you are interested in reviewing the entire ATC project, including all of the code, feel free to download it from www.modelstocode.com.

We translate the three facets of xUML in the order of classes, states, and actions. Translating the class model specifies the entities on which the code operates. Translating state models determines the sequencing and synchronization of the code and establishes the structure on which most of the actions are executed. Translating actions implements the algorithmic processing. And by translating all three model facets, we get a complete runnable program.

Overview of Pycca Syntax

Pycca reads one or more model source files and produces a single C header and source file pair. The C code written directly in the model file will be rearranged and mixed in with additional C code generated by pycca. Pycca has a few syntax conventions:

- *Comments*: A comment begins with the hash character (#) and continues to the end of the line.
- *Symbol names*: Symbol names follow the syntax of C identifiers. Names must begin with a letter, followed by an arbitrary number of letters, decimal digits, or underscore () characters. Models may use whitespace in class or attribute names, and these must be eliminated or replaced during translation because the names will be used directly as C identifiers.
- *C code*: Anything enclosed in braces ({{}}) is passed directly on to the C compiler.
- *Variable declarations*: Anything enclosed in parentheses (()) is taken as a comma-separated list, possibly empty, of C variable declarations.

Organization of a Pycca File

The largest unit of organization in a pycca file is a *domain*. The ATC example is centered around a single application domain, though you may have noticed a few actions that coordinate with external domains. In Chapter 6, we'll pursue a multidomain example, but for this pycca file, we'll keep to a single domain.

A domain definition is cumulative. If the same domain name is encountered later in the input, any content in the later definition is simply incorporated into the existing domain. A domain is defined as follows:

```
domain atctrl
    # Model statements (data types, classes, etc)
end

domain atctrl
    # Additional atctrl model statements
    # For example, the initial instance population.
    # Domain statements may also be placed in separate files.
end
```

The model statements that make up the content of a domain will be introduced as needed as we go about translating the Air Traffic Controller model. For reference, all possible model statements can be found in the online materials. And you don't need to worry about the ordering of model statements within a domain section. Pycca parses all of its input before attempting to generate any output. It orders the generated output to satisfy the C compiler's needs.

Pycca assumes the existence of runtime functions that implement data management and the event processing required by the state models. We discuss the runtime library in Chapter 5. For now, it is sufficient to know that a small piece of code manages execution by generating signals and dispatching the corresponding events, causing transitions that invoke state activities. You can use the preprocessor macros that pycca supplies as a convenient way to interface to the runtime code for such activities.

Translating the Class Model

We first translate those aspects of the model that deal with data. We specify the data types, followed by the class definitions and, finally, an initial instance population.

Data Types

What follows is the data type implementation for the domain. Data types are grouped in the interface or implementation categories. The interface category is for those types that must be visible outside the domain for the purposes of interaction. They are put into a header file that can be included by other code. The implementation category is for all other data types in a domain. Generally, the majority of data types are in the implementation category. If it helps, you can think of these two categories as public and private with respect to a domain.

```
domain atctrl
    ①
    # ....
    # Other domain components
    # ....
```

```

interface prolog {
    typedef char const *Employee_ID ;
    ❷
    typedef char const *Station_Number ;
    ❸
    typedef char const *Czone_Name ;
}

implementation prolog {
    ❹
    typedef char const *Name_T ;
    ❺
    typedef time_t Date_T ;
    typedef unsigned Aircraft_Quantity ;
    typedef unsigned Aircraft_Maximum ;
    typedef unsigned Duration ;
    typedef char const *Experience_Level ;

    #include <assert.h>
    #include <time.h>
    #include <string.h>
    #include "atctrl.h"
}

# ....
# Other domain components
# ....

```

end

❶ In this pycca source, we show the surrounding `domain/end` statements. Later in this chapter we assume all domain component definitions are enclosed in an outer `domain` statement and dispense with including the containing statements themselves.

❷ This section defines declarations that are needed as part of the external interface of the domain. Here we include those `typedef` statements that describe parameters exchanged with other domains. For the `atctrl` domain, there are external calls to report error and warning conditions (presumably to a user interface domain) that include the value of an employee identifier. So, we must make the data type for that value available for the interface. Interface `prolog` code is placed at the top of the generated C header file. Any additional required `include` files or other declarations can be included as part of the `interface prolog`.

❸ We are careful to declare immutable values as `const`. For embedded systems, we can direct the linker to place constants in nonvolatile memory (for example, flash), of which we usually have much more than RAM. In C, string literals are constant, and we would not want a piece of code changing a character in an employee's name.

- ❸ This section defines declarations used inside the domain and, therefore, not visible externally. The `implementation prolog` text is placed at the top of the generated C header file. This section can also be used to include declarations for external libraries that might be used as part of the processing of the domain (for example, standard C libraries or a math library or a core algorithm from a legacy system).
- ❹ We have called these types `Name_T` and `Date_T` in variance to the model names because of a limitation in `pycca` that does not allow attributes and types to have the same name.

Class Definitions

`Pycca` will translate each class definition in the model file into a C structure definition. In the following sections, we show the `pycca` class definitions for some of the model classes of the Air Traffic Controller model from Chapter 2. Please refer to the model classes in Chapter 2 as you read along.

Duty Station

We start with a simple class, `Duty Station`. Figure 4-1 shows the correspondence between the class model graphic and `pycca` source. In the `pycca` source, we adhere closely to the names of attributes and relationships as they are found in the model. Nothing is gained by arbitrarily changing names and mucking up the correspondence between the model and the code!

Pycca Class Statement

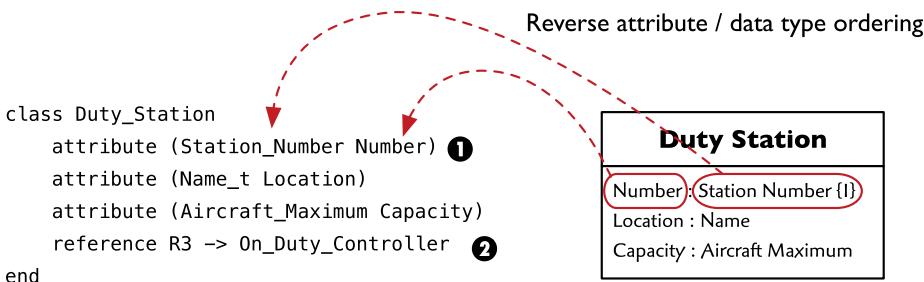


Figure 4-1. Specifying the `Duty Station` class

- ❶ We have retained the `Number` identifying attribute because its value is used in an activity.
- ❷ `Duty_Station` participates in a simple relationship, `R3`. The multiplicity of `R3` traversing from `Duty_Station` to `On_Duty_Controller` references at most one instance of `On_Duty_Controller` (`0..1` in UML notation). The reference statement with `->` denotes this multiplicity. A zero in a multiplicity expression expresses conditionality, which means that there may be no related instances at any given time. This idea is implemented by letting the `R3` member take on a `NULL` value.

Where the model shows attributes as names followed by the data type, we follow C syntax in the model source file with the data type first, followed by the attribute name. Anything enclosed in parentheses must follow C variable declaration syntax because it will be passed directly to the C compiler.

Pyccca translates each class statement into a C structure definition. Figure 4-2 shows the correspondence between the model graphic for the Duty Station class and the C structure generated by pyccca.

C code generated for a class

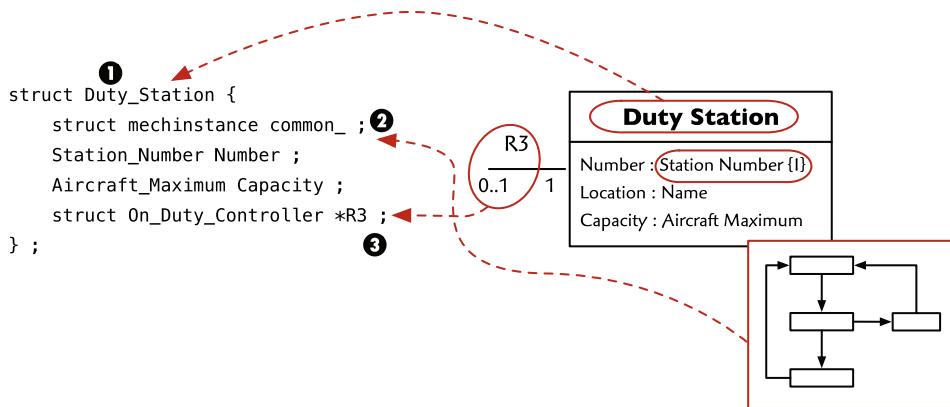


Figure 4-2. C generated for the Duty Station class

① Note that the class names are used directly as structure names, so you must choose class names that are valid C identifiers.

② Pyccca inserts this member for the runtime to use when managing the data and state processing of the class. Classes that don't require those functions of the runtime will not be given this structure member. The use of this structure member is described in detail in Chapter 5.

③ The singular reference statement is converted into a pointer to a structure.

This structure definition has a direct correspondence to the pyccca statements. Class attributes become structure members. Simple relationships are implemented as pointers to objects of the related structure type.

Air Traffic Controller

For the Air Traffic Controller class, we follow the same principles for the attributes as with the Duty Station class. The additional consideration for this class is the R1 generalization. Figure 4-3 shows the encoding of the model graphic into pyccca statements.

Pycca Union Statement

```
class Air_Traffic_Controller
    attribute (Employee_ID ID)
    attribute (Name_t Name)
    attribute (Experience_Level Rating)

    subtype R1 union ❶
        Off_Duty_Controller
        On_Duty_Controller ❷
    end
end
```

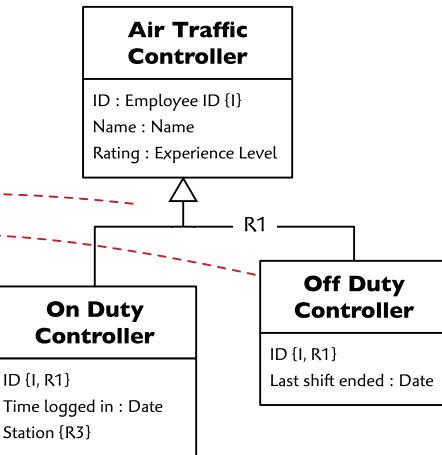


Figure 4-3. Specifying a superclass with a union

❶ This statement declares R1 as a generalization whose subclasses are to be held as a union member within the `Air_Traffic_Controller` structure. We chose the union implementation as this works best in simple cases.

❷ It's okay that we haven't defined the subclasses yet. Pycca imposes no particular order on the definitions.

Similar to the Duty Station class, pycca generates a C structure definition for the Air Traffic Controller class:

```
struct Air_Traffic_Controller {
    struct mechinstance common_;
    Employee_ID ID;
    Name_T Name;
    Experience_level Rating;
    SubtypeCode R1_code; ❶
    union {
        struct Off_Duty_Controller R1_Off_Duty_Controller;
        struct On_Duty_Controller R1_On_Duty_Controller;
    } R1;
};
```

❶ The new constructs seen in this declaration are the `R1_code` and `R1` members. The `R1_code` member holds a small integer that encodes the union element currently related to the superclass instance, and pycca supplies preprocessor definitions for the encoded values. The `R1` structure member is a union of all the subclasses that participate in the `R1` generalization.

From a model execution point of view, when navigating `R1` from the Air Traffic Controller class to one of its subclasses, we must be able to determine whether the traversal across `R1` comes up empty. So, if we have an instance of Air Traffic Controller and we navigate across `R1` to the Off Duty Controller class, it could be the case that the Air Traffic Controller instance is not currently related to an Off Duty Controller class instance (that is, it is currently related to an instance of On Duty Controller). The navigation operation yields the empty set, and we must be able to determine that.

From an implementation point of view, we need to know how to interpret the R1 structure member. By testing the value of R1_code, we can determine which element of the union is currently in use. By placing the subclasses in the R1 union member, no other storage for the R1 generalization is needed. We can traverse R1 by appropriate pointer arithmetic. Pyccca provides macros to do the heavy lifting and helps prevent common errors. Don't be alarmed that pyccca uses the keyword subtype in this context.

On Duty Controller

Our last example of data translation is the On Duty Controller class. A key decision for this class is how to store information for the R2 association. By considering the domain actions, we see that R2 associates one On Duty Controller instance to many Control Zone instances and that association is dynamic in nature. Actions will link and unlink instances of On Duty Controller to an arbitrary number of Control Zone instances. We need a data structure for the relationship storage that supports the multiplicity and dynamics of R2. Pyccca provides a suitable linked list mechanism. That works for R2, but it is often the case, especially in embedded systems, that associations are static over the running time of the application. So pyccca also provides a means for storing reference values suited to static associations. Figure 4-4 shows how the On Duty Controller class is specified.

Pyccca Reference Statements

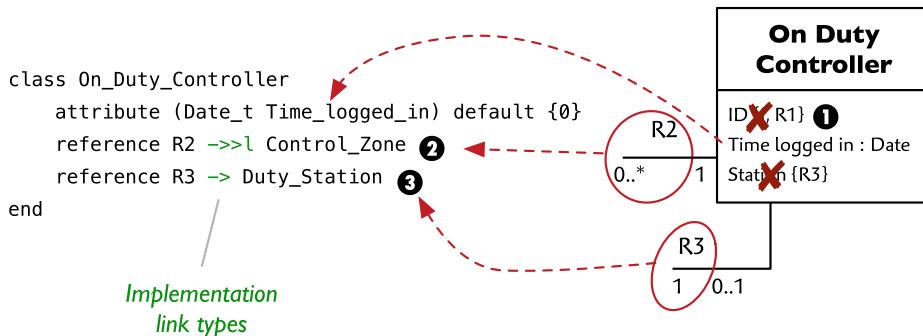


Figure 4-4. Specifying the On Duty Controller class

❶ Because we have chosen to hold the subclasses as a union composed into the superclass, we have discarded the ID attribute. It is not needed to implement the R1 relationship, and its value is available from the superclass if we need it. The default value for Time_logged_in is zero because we are using a POSIX representation of time, and zero is used to indicate an unknown login time.

❷ The ->>1 (lowercase letter l) notation tells pyccca to use a linked list to store instance references for the R2 association in the Control Zone direction. This is one of the link types mentioned in Chapter 3 for storing instance references to implement associations.

❸ The -> notation tells pyccca that storage for R3 will be a single instance reference to a Duty_Station class.

Pyccca generates the following code:

```

struct On_Duty_Controller {
    Date_T Time_logged_in ;
    rlink_t R2 ;           ❶
    struct Duty_Station *R3 ; ❷
} ;

```

- ❶ Pycca will provide the necessary data structures and code to deal with the linked list.
- ❷ The storage for relationship R3 is simply a pointer to the struct `Duty_Station`. Because the multiplicity of the R3 relationship from On Duty Controller to Duty Station is one, at no time should the value of the R3 member be `NULL`. The R2 member is the head of the linked list onto which we can link a set of `Control_Zones`.

The other modeled classes follow a similar pattern, so we won't reproduce them here. Complete pycca source for the model, in the form of a [literate program](#), can be found at the [website](#) for this book.

Initial Instance Population

After the class model has been fully translated, it requires one more thing before it can be used: an initial instance population. This is a set of instances that are present when the system is initialized. Pycca can place an initial population into memory as initialized variables, so no code is required to create instances during initialization. This is particularly important in embedded systems in which instance populations are frequently static. For dynamic populations, instances may be created both before and during runtime.

In Chapter 2, example instances in the ATC domain were shown as tables. Pycca supports a tabular syntax for specifying initial instance populations. (There is also a syntax that is more convenient for specifying single instances.) The following pycca code specifies the initial instance population for our example:

```
table Air_Traffic_Controller
    (Employee_ID ID) (Name_T Name) (Experience_Level Rating) R1
@atc53 {"53"} ❶ {"Toshiko"} {"A"} -> On_Duty_Controller.atc53 ❶
@atc67 {"67"} {"Gwen"} {"B"} -> On_Duty_Controller.atc67
@atc51 {"51"} {"Ianto"} {"C"} -> On_Duty_Controller.atc51
end

table On_Duty_Controller
    R2          R3
@atc53 -> sfo end -> s2 ❷
@atc67 -> oak end -> s1
@atc51 -> sjc end -> s3
end

table Control_Zone
    (Czone_Name Name) (Aircraft_Quantity Traffic)      R2
@sfo   {"SF037B"}     {27}                         -> atc53
@oak   {"OAK21C"}     {18}                         -> atc67
@sjc   {"SJC18C"}     {9}                          -> atc51
end

table Duty_Station
    (Station_Number Number) (Name_T Location) (Aircraft_Maximum Capacity)
@s1    {"S1"} ❸ {"Front"}           {20}
@s2    {"S2"}           {"Center"}        {30}
@s3    {"S3"} ❸ {"Front"}           {45}
end
```

❶ For relationship storage, we are dealing with pointers. By naming the instance (using the @<name> syntax), you can refer to the instances in relationship initializers. Pycca keeps track of where the named instances reside in the allocated storage array and emit the proper address expressions. When specifying the subclass instance for a generalization, it is also necessary to give the subclass name by using the dot notation shown.

❷ For relationship paths that are singular, the -> notation indicates the name of the instance to be related.

❸ Pycca stores all instances of a class in a C array. The descriptive attributes, such as `Location`, are assigned to the appropriate structure member in the initializer for the array element. The values are specified in braces ({}), because pycca passes anything in braces directly through to the C compiler. This means we can use any valid C constant expression as an attribute initializer.

Pycca converts the initial instance population into an array of structures with initializer values. Each class has a separate array that serves as its memory pool. The following are the C variables and initializers generated for the initial instance:

```
/*
Initial Instance Storage for, "Air_Traffic_Controller"
*/
static struct Air_Traffic_Controller Air_Traffic_Controller_storage[3] = { ❶
    {❷.common_ = {1, 0, &Air_Traffic_Controller_class}, "53", "Toshiko", "A", .R1_code = ←,
     1, .R1 = {.R1_On_Duty_Controller = {0, .R2 = {.next = &Control_Zone_storage[0], ←
     R2_links, .prev = &Control_Zone_storage[0].R2_links}, .R3 = &Duty_Station_storage ←
     [1]}}, ←
    {.common_ = {2, 0, &Air_Traffic_Controller_class}, "67", "Gwen", "B", .R1_code = 1, . ←
     R1 = {.R1_On_Duty_Controller = {0, .R2 = {.next = &Control_Zone_storage[1].R2_links ←
     , .prev = &Control_Zone_storage[1].R2_links}, .R3 = &Duty_Station_storage[0]}}, ←
    {.common_ = {3, 0, &Air_Traffic_Controller_class}, "51", "Ianto", "C", .R1_code = 1, . ←
     R1 = {.R1_On_Duty_Controller = {0, .R2 = {.next = &Control_Zone_storage[2].R2_links ←
     , .prev = &Control_Zone_storage[2].R2_links}, .R3 = &Duty_Station_storage[2]}}, ←
    } ; ←
/* *Initial Instance Storage for, "Control_Zone"
*/
static struct Control_Zone Control_Zone_storage[3] = { ❸
    {"SFO37B", 27, .R2 = &Air_Traffic_Controller_storage[0].R1.R1_On_Duty_Controller, . ←
     R2_links = {.next = &Air_Traffic_Controller_storage[0].R1.R1_On_Duty_Controller.R2, ←
     .prev = &Air_Traffic_Controller_storage[0].R1.R1_On_Duty_Controller.R2, }}, ←
    {"OAK21C", 18, .R2 = &Air_Traffic_Controller_storage[1].R1.R1_On_Duty_Controller, . ←
     R2_links = {.next = &Air_Traffic_Controller_storage[1].R1.R1_On_Duty_Controller.R2, ←
     .prev = &Air_Traffic_Controller_storage[1].R1.R1_On_Duty_Controller.R2, }}, ←
    {"SJC18C", 9, .R2 = &Air_Traffic_Controller_storage[2].R1.R1_On_Duty_Controller, . ←
     R2_links = {.next = &Air_Traffic_Controller_storage[2].R1.R1_On_Duty_Controller.R2, ←
     .prev = &Air_Traffic_Controller_storage[2].R1.R1_On_Duty_Controller.R2, }} ←
} ; ←
/* * Initial Instance Storage for, "Duty_Station"
*/
```

```
static struct Duty_Station Duty_Station_storage[3] = {
    {.common_ = {1, 0, &Duty_Station_class}, "S1", "Front", 20, .R3 = NULL},
    {.common_ = {2, 0, &Duty_Station_class}, "S2", "Center", 30, .R3 = NULL},
    {.common_ = {3, 0, &Duty_Station_class}, "S3", "Front", 45, .R3 = NULL}
};
```

❶ Recall that the `Air_Traffic_Controller` class is a union containing the subclasses of R1, so no separate storage is allocated for the subclasses. The values for the subclass attributes are included directly in the initializer for the superclass.

❷ The `common_` member holds an allocation number, the current state, and a pointer to class invariant data. This is the data required by the runtime code to manage the instance. In Chapter 5, we discuss instances from the perspective of the runtime code and exactly how this structure member is used.

❸ The `Control_Zone` class has no state model, and all of its instances are part of the initial instance population. Consequently, there is no `common_` member of its class structure because the runtime does not have to manage the instances in any way.

Recall that R2 was implemented as a linked list originating at the `Air_Traffic_Controller` instances. Pycca properly initializes the link pointers in both the `Air_Traffic_Controller` and `Control_Zone` storage arrays. The pointer values in the initializers are constant expressions that index into the storage arrays and onto the element structure members. Pycca allocates instances to the storage array and can therefore specify these address calculations. Note that these expressions are bound to a memory address at link time after the base address of the storage array has been placed by the linker.

Translating State Models

The second aspect of translating is to specify the state models. All model classes with a state model will have a corresponding machine definition in pycca. State models specify both the transitions and the C code executed when a state is entered. The C code for a state is placed directly in the definition of the state model for a class. We leave out the C code for the time being so that we are better able to show just the transition aspects of the state model. In the next section, “Translating Actions,” the C code will make its appearance. We have included the action language in the state definitions as a C comment. It is particularly convenient to have the action language nearby when translating it into C code.

Duty Station State Model

We recommend referring to the Duty Station state model in Chapter 2, Figure 2-6, as you read along so you can readily see the correspondence between the model elements and the pycca statements.

The state model is encoded using a `Machine` statement:

```
class Duty_Station
    # ... Other parts of Duty_Station definition, attributes, etc.

    Machine
        state AVAILABLE()
```

❶
❷

```

{
    // empty
}
transition AVAILABLE - In_use -> IN_USE ③

state IN_USE()
{
    //+ Max shift exceeded -> me after Shift
    //+ Specification().Max shift // selects singleton ④
}
transition IN_USE - Max_shift_exceeded -> MAX_SHIFT_EXCEEDED
transition IN_USE - User_leaving -> Canceling_Shift_Timeout

state MAX_SHIFT_EXCEEDED()
{
    //+ UI.Break required( Station: Number )
}
transition MAX_SHIFT_EXCEEDED - User_leaving -> AVAILABLE

state Canceling_Shift_Timeout()
{
    //+ cancel Max shift exceeded -> me
    //+ Available -> me
}
transition Canceling_Shift_Timeout - Available -> AVAILABLE
end

```

- ❶ The state model for a class is completely specified within a `machine` statement.
- ❷ Here we define a state. This statement specifies the name of the state along with any parameters passed through incoming transitions. A corollary of the Moore state model formalism is that all incoming transitions to a state must supply the same set of parameters. As it turns out, the Duty Station state model has no parameters defined on any of its transitions.
- ❸ The `transition` statement defines the state model transition (that is, what happens when events are received by an instance of this class). Transitions are defined by the starting state, a received event, and the new state that is the destination of the transition. Events do not require a separate definition. Event names are gathered from the `transition` statements.
- ❹ The state activity will ultimately be C code, not generated, but written by hand directly into the model source file. Don't be alarmed; pycca provides a set of C macros that you can substitute for most model actions. We'll get to all of that later. For now, we are copying in the modeled action language as comments. Even after we replace it with C code, it is a good idea to retain the original action language as a comment.

The pycca source is a direct transliteration of the state model graphic. The syntax to specify a state is reminiscent of a function definition complete with parameters (from the triggering event, none in this example) and the code (activity) to be executed.

No particular order is imposed on the state and transition statements. Here we have placed outgoing transitions immediately below the state to which they apply. Another convenient organization is to place all the transition statements together to emphasize the correspondence to the transition matrix for the state model.

A set of data structures encodes all the state behavior for a given class. The runtime code uses these data structures to dispatch events to state machines. An important function of pycca is generating the data structures used by the runtime event dispatch code. Pycca transforms the state model specification into a set of initialized variables whose values describe how to dispatch events and what processing to execute when transitions occur. The dispatch code uses a tabular encoding of the state model. The behavior of the event dispatch is the same for all the instances of a class. The algorithm for determining transitions and executing state activities is the same for all classes.

The following listing shows the data structures generated by pycca for the Duty Station state model. The data are initialized variables for a dispatch block, transition table, and action table. In Chapter 5, we show the structure of this data and exactly how it is used by the runtime code to cause state transitions to happen. Don't be too concerned if the following doesn't make much sense yet. After you've read through Chapter 5 and have a better understanding of how state transitions are managed and events are dispatched in the runtime environment, it should all seem a bit more reasonable.

```
static PtrActionFunction const Duty_Station_acttbl[] = {
    Duty_Station_AVAILABLE,
    Duty_Station_IN_USE,
    Duty_Station_MAX_SHIFT_EXCEEDED,
    Duty_Station_Canceling_Shift_Timeout,
} ;
```

❶

```
static StateCode const Duty_Station_transtbl[] = {
    MECH_STATECODE_CH, // AVAILABLE - Available -> CH
    1, // AVAILABLE - In_use -> IN_USE
    MECH_STATECODE_CH, // AVAILABLE - Max_shift_exceeded -> CH
    MECH_STATECODE_CH, // AVAILABLE - User_leaving -> CH
    MECH_STATECODE_CH, // IN_USE - Available -> CH
    MECH_STATECODE_CH, // IN_USE - In_use -> CH
    2, // IN_USE - Max_shift_exceeded -> MAX_SHIFT_EXCEEDED
    3, // IN_USE - User_leaving -> Canceling_Shift_Timeout
    MECH_STATECODE_CH, // MAX_SHIFT_EXCEEDED - Available -> CH
    MECH_STATECODE_CH, // MAX_SHIFT_EXCEEDED - In_use -> CH
    MECH_STATECODE_CH, // MAX_SHIFT_EXCEEDED - Max_shift_exceeded -> CH
    0, // MAX_SHIFT_EXCEEDED - User_leaving -> AVAILABLE
    0, // Canceling_Shift_Timeout - Available -> AVAILABLE
    MECH_STATECODE_CH, // Canceling_Shift_Timeout - In_use -> CH
    MECH_STATECODE_CH, // Canceling_Shift_Timeout - Max_shift_exceeded -> CH
    MECH_STATECODE_CH, // Canceling_Shift_Timeout - User_leaving -> CH
} ;
```

❷

```
static struct objectdispatchblock const Duty_Station_odb = {
    .stateCount = 4,
    .eventCount = 4,
    .transitionTable = Duty_Station_transtbl,
    .actionTable = Duty_Station_acttbl,
    .finalStates = NULL
} ;
```

❸

❶ In the next section, you will see how activity functions are defined. Here, all the activity functions for a state model are collected into a single array so the appropriate one can be located when a given state is entered.

❷ CH means *can't happen*. Any event that causes a CH transition results in a system error.

❸ The object dispatch block is the collection of information used by the runtime code to accomplish event delivery. This is explained further in Chapter 5.

Air Traffic Controller State Model

The state model for the `Air_Traffic_Controller` class is somewhat longer but follows the same pattern as shown previously for the `Duty_Station` class:

```
class Air_Traffic_Controller

    # ... Other parts of Air_Traffic_Controller definition, attributes, etc.

    machine
        state OFF_DUTY ()
        {
            // empty
        }
        transition OFF_DUTY - Ready_for_duty -> Verifying_Adequate_Break

        state Verifying_Adequate_Break(Station_Number Station)
        {
            //+ the shift spec .= Shift Specification() // selects singleton
            //+ if ( _now - self.Last shift ended < the shift spec.Min break )
            //+     Log in( in.Station ) -> me
            //+ else
            //+     Cannot go on duty -> me
        }
        transition Verifying_Adequate_Break - Log_in -> Logging_In
        transition Verifying_Adequate_Break - Cannot_go_on_duty -> OFF_DUTY

        state Logging_In(Station_Number Station)
        {
            //+ migrate to On Duty Controller
            //+ my station = Duty Station( ID: in.Station )
            //+ link /R3/my station
            //+ Time logged in = now()
            //+ Logged in -> me
            //+ In use -> my station
        }
        transition Logging_In - Logged_in -> ON_DUTY

        state ON_DUTY()
        {
            // empty
        }
```

```

transition ON_DUTY - Ready_for_a_break -> Verifying_Full_Handoff
transition ON_DUTY - Handoff -> Handing_off_Control_Zone

state Handing_off_Control_Zone(
    Czone_Name zone,
    Employee_ID controller)
{
    //+ hoff zone .= /R2/Control Zone( Name: in.Zone )
    //+ if in.Controller == ID
    //+     UI.Cannot handoff to self( Controller: in.Controller )
    //+ else {
    //+     new controller .= On Duty Controller( ID: in.Controller )
    //+     swap hoff zone/R2/On Duty Controller with new controller
    //+         !new missing: UI.Unknown controller( Controller: in.Controller )
    //+         !old missing: UI.Zone not handled by( Controller: ID )
    //+ } // swaps controllers and checks for errors
    //+ Handoff complete -> me
}
transition Handing_off_Control_Zone - Handoff_complete -> ON_DUTY

state Logging_Out()
{
    //+ User leaving -> /R3/Duty Station
    //+ migrate to Off Duty Controller
    //+ Last shift ended = _now.HMS
    //+ Off duty -> me
}
transition Logging_Out - Off_duty -> OFF_DUTY

state Verifying_Full_Handoff()
{
    //+ if /R1/On Duty Controller/R2/Control Zone{
    //+     Must handoff zones -> me
    //+     UI.Control Zones Active( ATC: ID )
    //+ } else
    //+     Log out -> me
}
transition Verifying_Full_Handoff - Log_out -> Logging_Out
transition Verifying_Full_Handoff - Must_hand_off_zones -> ON_DUTY
end
end

```

We don't show the generated output, because it follows the same pattern you have already seen.

Translating Actions

In this section, we translate the processing associated with actions in the state activities. Pycca turns each state activity into a C function and arranges for the function to be included in the data structures generated for the runtime dispatch mechanism.

State activities almost always result in updating instance attributes and/or signaling an event. Data access is provided by the normal C mechanism for access to structure members via a pointer. Pycca augments the C code you provide for the state activity with the necessary declarations for a `self` (me) variable and a `rcvd_evt` variable used to access event parameters.

Air Traffic Controller State Activities

We begin with the Air Traffic Controller class showing only selected state activities. The state activity for the Logging In state is shown in the following code. We have included the action language for the state as a comment and have interspersed the C that implements the corresponding actions:

```
class Air_Traffic_Controller

    # ... Other parts of Air_Traffic_Controller definition, attributes, etc.

    machine

        # ... Other parts of the state model definition

        state Logging_In(Station_Number Station)
        {
            //+ migrate to On Duty Controller
            PYCCA_migrateSubtype(self, Air_Traffic_Controller, R1,
                On_Duty_Controller) ; ❶

            //+ my_station = Duty_Station( ID: in.Station )
            ClassRefVar(Duty_Station, my_station) ; ❷
            PYCCA_selectOneStaticInstWhere(my_station, Duty_Station,
                strcmp(my_station->Number, rcvd_evt->Station) == 0) ;
            assert(my_station != EndStorage(Duty_Station)) ; ❸

            //+ & /R3/my_station
            ClassRefVar(On_Duty_Controller, ondc) =
                PYCCA_unionSubtype(self, R1, On_Duty_Controller) ; ❹
            ondc->R3 = my_station ; ❺
            my_station->R3 = ondc ;

            //+ Time logged in = now()
            ondc->Time_logged_in = time(NULL) ; ❻

            //+ Logged in -> me
            PYCCA_generateToSelf(Logged_in) ; ❼

            //+ In use -> my station
            PYCCA_generate(In_use, Duty_Station, my_station, self) ;
        } ❽

        # ... Other part of the state model definition

    end
end
```

- ❶ Pycca provides a macro to perform the subclass migration for union-based generalizations. This hides the internal instance structure members.
- ❷ The ClassRefVar macro declares a local variable that is a reference to an instance. In this case, `my_station` is a variable that references an instance of `Duty_Station`.
- ❸ Finding a `Duty_Station` is accomplished with a linear search of the storage array for `Duty_Station`. Pycca provides a convenience macro to perform the iteration. You would choose other searching techniques if the number of instances of `Duty_Station` is large. Note that the `Station` event parameter is obtained via the implicit `rcvd_evt` variable that is automatically generated by pycca.
- ❹ Because `R3` is between an On Duty Controller and a Duty Station, we must traverse `R1` first. `R1` is implemented as a union, and the macro performs the required pointer arithmetic.
- ❺ Each class participating in `R3` is keeping a reference to the corresponding instance, and so linking involves setting both pointer values.
- ❻ Accounting for time is platform specific, and pycca does not supply any time functions. Here we are assuming a POSIX environment.
- ❼ Here, a signal to `self` is generated. Note that `Logged_in` is the same name used in the state model definition.
- ❽ The general form of signal generation specifies the signal, the class of the target instance, the target instance itself, and the instance that is the source of the signal.

Pycca turns the state action into a C function as shown here:

```
static void Air_Traffic_ControllerLogging_In(void *const s_, void *const p_)
{
#define THISSTATE__ Logging_In
    struct Air_Traffic_Controller *const self = (struct Air_Traffic_Controller *)s_ ; ❶
    struct Air_Traffic_ControllerLogging_In_rcvd_evt {
        Station_Number Station ;
    } const *const rcvd_evt = (struct Air_Traffic_ControllerLogging_In_rcvd_evt const *)p_ ; ❷

    //+ migrate to On Duty Controller on R1
    PYCCA_migrateSubtype(self, Air_Traffic_Controller, R1, On_Duty_Controller) ;

    //+ my_station = Duty Station( ID: in.Station )
    ClassRefVar(Duty_Station, my_station) ;
    PYCCA_selectOneStaticInstWhere(my_station, Duty_Station,
        strcmp(my_station->Number, rcvd_evt->Station) == 0) ;
    assert(my_station != EndStorage(Duty_Station)) ;

    //+ & self.R3.my_station // link station
    ClassRefVar(On_Duty_Controller, ondc) = PYCCA_unionSubtype(self, R1, On_Duty_Controller) ;
    ondc->R3 = my_station ;
    my_station->R3 = ondc ;
```

```

//+ Time logged in = _now.HMS
ondc->Time_logged_in = time(NULL) ;

//+ Logged in -> me
PYCCA_generateToSelf(Logged_in) ;

//+ In use -> my station
PYCCA_generate(In_use, Duty_Station, my_station, self) ;
#undef THISSTATE_
}

```

❶ Preprocessor #define statements keep track of the current state. The current state is used by other preprocessor macros.

❷ Pycca arranges the declarations of `self` and `rcvd_evt`.

Other than the declarations of `self` and `rcvd_evt`, pycca has simply turned the state action into a function of file static scope and passed along the C code that was specified in the state model definition. The name of the function has the class name prepended to make it unique within the file.

We show one more example of translating a state activity. Following is the pycca source for the Logging Out state of the Air Traffic Controller class:

```

class Air_Traffic_Controller

# ... Other parts of Air_Traffic_Controller definition, attributes, etc.

machine

# ... Other parts of the state model definition

state Logging_Out()
{
    //+ User leaving -> /R3/Duty Station
    assert(self->SubCodeMember(R1 ==
        SubCodeValue(Air_Traffic_Controller, R1,On_Duty_Controller)) ;
    ClassRefVar(On_Duty_Controller, ondc) =
        PYCCA_unionSubtype(self, R1, On_Duty_Controller) ;           ❶
    ClassRefVar(Duty_Station, ds) = ondc->R3 ;                      ❷
    assert(ds != NULL) ;
    PYCCA_generate(User_leaving, Duty_Station, ds, self) ;

    //+ migrate to Off Duty Controller
    ondc->R3 = NULL ;                                              ❸
    ds->R3 = NULL ;

    PYCCA_migrateSubtype(self, Air_Traffic_Controller, R1,
        Off_Duty_Controller) ;
    assert(self->SubCodeMember(R1) ==
        SubCodeValue(Air_Traffic_Controller, R1,
        Off_Duty_Controller)) ;
}

```

```

//+ Last shift ended = _now.HMS ClassRefVar(Off_Duty_Controller, offdc) =
    PYCCA_unionSubtype(self, R1, Off_Duty_Controller) ; ④
offdc->Last_shift_ended = time(NULL) ;

//+ Off duty -> me
PYCCA_generateToSelf(Off_duty) ;
}

# ... Other part of the state model definition

end
end

```

❶ This traverses R1 from Air_Traffic_Controller to On_Duty_Controller. The assertion checks that the instance of ATC is indeed related to an instance of On_Duty_Controller. If not, something is terribly wrong.

❷ Traversal of R3 from an instance of On_Duty_Controller to an instance of Duty_Station is a simple matter of retrieving a pointer. The relationship is unconditional from On Duty Controller to Duty Station; therefore, a valid pointer value must be stored in Duty_Station. R3. The assertion checks the data integrity.

❸ Because each class contains a reference to the other, maintaining the relationship storage means updating both pointers.

❹ Because we have migrated subclasses, we must obtain a reference to the new subclass.

The generated C code follows:

```

static void Air_Traffic_ControllerLogging_Out(void *const s_, void *const p_)
{
#define THISSTATE__ Logging_Out
    struct Air_Traffic_Controller *const self = (struct Air_Traffic_Controller *)s_ ; ❺

    //+ User leaving -> /R3/Duty Station
    assert(self->SubCodeMember(R1) == SubCodeValue(Air_Traffic_Controller, R1,
        On_Duty_Controller)) ;
    ClassRefVar(On_Duty_Controller, ondc) =
        PYCCA_unionSubtype(self, R1, On_Duty_Controller) ;
    ClassRefVar(Duty_Station, ds) = ondc->R3 ;
    assert(ds != NULL) ;
    PYCCA_generate(User_leaving, Duty_Station, ds, self) ;

    //+ migrate self to Off Duty Controller on R1
    ondc->R3 = NULL ;
    ds->R3 = NULL ;

    PYCCA_migrateSubtype(self, Air_Traffic_Controller, R1,
        Off_Duty_Controller) ;
}

```

```

//+ Last_shift_ended = _now.HMS
ClassRefVar(Off_Duty_Controller, offdc) =
    PYCCA_unionSubtype(self, R1, Off_Duty_Controller) ;
offdc->Last_shift_ended = time(NULL) ;

//+ Off duty -> me
PYCCA_generateToSelf(Off_duty) ;
#endif THISSTATE_
}

```

- ➊ There is no definition of `rcvd_evt` for this action because there are no event parameters.

The C output is as expected. The action is turned into a function, and the C code has been passed through.

Summary

The Air Traffic Control model from Chapter 2 has been translated here by applying the principles discussed in Chapter 3. The model was encoded in a text file by using pycca statements intermixed with C code for the state activities and following C language rules for naming. This file was then fed through the pycca processor, which generated additional C code from the pycca statements, passing along the intermixed C code in a form and order acceptable to the compiler. To complete the translation, the generated pair of C header and source code files were then processed by a C compiler. Linking the object file for the domain with the runtime code yields a complete executable program.

All three facets of the model were specified in the pycca script language to create a complete model source file. First, the class model was specified. This consisted of a set of data type definitions split into those required for external interaction and those private to the domain. These were followed by a set of class statements. Each class statement defined the modeled class name, its attributes, its relationships, and an optional state model. Some design decisions were made at this point, based on static vs. dynamic populations and required paths for relationship navigation. In the process, some of the identifier and referential attributes from the class model were not needed and so eliminated. Nonetheless, their fundamental logical intent was still present. For example, a referential attribute may have been replaced by a pointer.

For each class with a state model, a machine statement was embedded within the `class` statement to specify the states, transitions, events, and activities. Each activity is initially filled with a comment containing the action language copied from the corresponding state.

The last facet of the model, the actions, was then completed by inserting C code within each state. In many cases, the C code consists of pycca-provided C preprocessor macros that provide the implementation of common model actions such as signal generation, instance selection, and relationship navigation. Additionally, user-written C code is used for flow of control, computing expressions, and other algorithmic processing.

CHAPTER 5



Model Execution Domain

Until now, we have glossed over the operations of the translation's runtime component. We have described some of its functions, but have not said how it accomplishes anything. No longer! In this chapter, we explain how code for managing model execution works and the important role it plays in generating a running program.

Our focus remains on getting the air traffic control model to run, so this chapter discusses the model execution of our target platform. With the means in place to control the data and sequencing of a model, we are in a position to run the code derived from the translation. We do not explain all the runtime operations in this chapter, only those parts needed to run our example. In Chapter 9, we fill in the remaining details.

The important insight here is that all the decisions about how execution is sequenced and managed have been factored into one place. In fact, the data, rules, and policies of model execution form a distinct subject matter entirely independent of any particular application or service. This subject matter is, in fact, a domain just like any other. We call it the *Model Execution domain*, or *MX domain* for short. This domain has historically been referred to as the *software architecture*, but we'll avoid using that term without qualification because the wider computing community also uses this term to describe vaguely the overall composition of design elements of a software system.

We start with an overview of the role that a Model Execution domain plays in building a running program by translation. We want to tell you how it fits in the larger scheme of a translation. But quickly, we get down to the specifics of our example target platform. In Chapter 10, we discuss other possible platforms and some general considerations that go into a Model Execution domain.

Role of the Model Execution Domain

The Model Execution domain (MX domain) forms the logical platform on which the models execute. In the same way that a processor instruction set is the target of a programming language compiler, the building blocks of the MX domain are the target of model translation. Every modeled system requires an MX domain, as it provides the mechanism to run the model execution rules.

A processor instruction set consists of operations such as ADD and MOV. The building blocks of the MX domain involve implementations of model execution elements such as event control blocks, transition tables, and action tables, all of which are introduced in this chapter. The process of model translation maps model elements such as classes, relationships, and states onto the MX building blocks.

Some of the MX capabilities are provided directly by the implementation language. For example, invoking a function and accessing data directly or indirectly are common operations in C programs. When activities are translated, those C features are used directly. Other model execution rules are not supported directly by the implementation language. There are no C statements, for example, to signal an event and have that event dispatched to an instance. To bridge this gap, the MX domain provides C functions for event signaling and dispatch. The translation process maps state actions that signal events onto the invocation of these MX-provided C functions to implement xUML event-signaling behavior.

Although the subject matter of the MX domain is distinct from the modeled application and service domains it runs, it is strongly influenced by the computational demands imposed by those domains. The rules of model execution are well defined (for example, in *Executable UML: A Foundation for Model-Driven Architecture* referenced previously), but the computing strategies and technology to implement them can vary dramatically. Scale, speed, and parallelism are all performance features that a modeled domain may demand of its implementation. The availability of memory and its usage are also important considerations. There is no *universal* MX domain that can cover all application demands in an optimal way. This is hardly a surprise, as there is no universal processor or universal programming language that can cover all computing circumstances. Decisions about which computing technology to apply to a particular problem involve many engineering trade-offs, often contradictory. To obtain an optimal balance of cost, capability, and efficiency, an MX domain needs to provide the best match of computing technology to the needs of the modeled domains.

Neither is it necessary to build a separate MX domain for every software project! A well-designed, configurable MX domain can accommodate the demands of a broad class of applications and platforms. This should come as no surprise, as we are all accustomed to using components such as user-interface toolkits, math libraries, or database engines optimized for a limited range of applications and platforms. No one would seriously suggest that such a component must be redesigned for every software project. In the same sense, we can carefully tailor the flexibility designed into an MX domain so it can run efficiently on a class of platforms and support a wide range of applications. In Chapter 10, we show how an MX domain for a different platform could be constructed.

In this chapter, we discuss one particular implementation of an MX domain called *Single-Threaded Model Execution*, or ST/MX. It is designed to target bare-metal, microcontroller-based embedded systems of the type we are considering.

Overview of ST/MX

To translate a model, we must specify the general characteristics of the environment and the type of computing technology on which the resulting program runs. As discussed earlier, we have chosen microcontroller-based embedded systems as the target. This decision places constraints on what constitutes an acceptable solution and limits the types of applications that can be deployed on the target. ST/MX was designed to handle the most important constraints of a microcontroller-based system:

- *Limited memory*: Microcontrollers have small memories that are typically partitioned into RAM and read-only (for example, flash) types. There is a single physical memory space with no hardware-based memory address translation.
- *Limited execution cycles*: Microcontrollers have limited speeds and are often deployed in battery-powered applications for which processor execution is one of the more significant power-consuming activities.
- *Interrupts*: All modern computers have interrupts that allow external peripherals to indicate significant happenings. ST/MX must support model operations and interrupt service execution in a coherent manner.
- *Timely response*: Embedded systems usually react to stimuli from their external environment and produce a response back into this environment. Often, time constraints exist on producing the response. ST/MX strives to minimize execution latency to meet real-time constraints.

To give you a good sense of how ST/MX works, we limit discussion in this chapter to the way the sequencing of modeling behavior is managed. We focus on the way signals are issued and state machine events are dispatched to the appropriate instances. Then we demonstrate how this relates to the execution of our example ATC models.

ST/MX is atypical in that the actual behavior of model elements is driven strictly by supplied data values. A primary role of pycca, as was shown in the Air Traffic Controller example, is to generate the data values to drive ST/MX. ST/MX itself knows nothing about the particular behavior of an Air Traffic Controller or any other model element. This is no surprise. Real computers know nothing of the meaning of what they compute and strictly act according to the program they are presented.

ST/MX is implemented as an ordinary C library. Just like any other library, it has an interface made up of functions, parameters, and data structures. Unlike a typical C library, however, much of the public interface consists of preprocessor macros. These hide the underlying low-level functions and ease the process of translating action language. Consequently, the processing and algorithms are discussed in general terms. We do not give complete invocation interfaces because, when translating with pycca, the low-level ST/MX function interfaces are referenced only in the pycca-supplied macros. For those readers who wish to see the code details and the design rationale, a complete description of the ST/MX domain, including the C code that implements it, can be found on the book's website.

The ST/MX View of a Class Instance

ST/MX doesn't care about the meanings of individual instances. It just needs nimble access to them for activities such as creating, deleting, locating a transition table, dispatching an event, and so forth. None of these activities require knowledge of particular attribute values. Consequently, instances are seen as relatively opaque blobs of user data. Generic data about an instance such as its class (to find the transition table), its location in memory, the current state, and other management information are what's of interest to ST/MX.

Class instances are stored as an array of C structs. To manage an instance, ST/MX places generic data about an instance as the first member of the class structure declaration by using the following struct:

```
typedef struct mechinstance {
    AllocCount alloc ;
    StateCode currentState ;
    struct mechclass const *instClass ;
} *MechInstance ;
```

The `MechInstance` pointer type defined in this source code snippet is used later. Here, we focus on the incorporation of the `mechinstance` struct as the first member of an example generated pycca class. Figure 5-1 shows the class-specific structure for an Air Traffic Controller. Remember that the Air Traffic Controller class amalgamates the modeled super- and subclasses within a union.

Air Traffic Controller

Example Instance Data Structure:

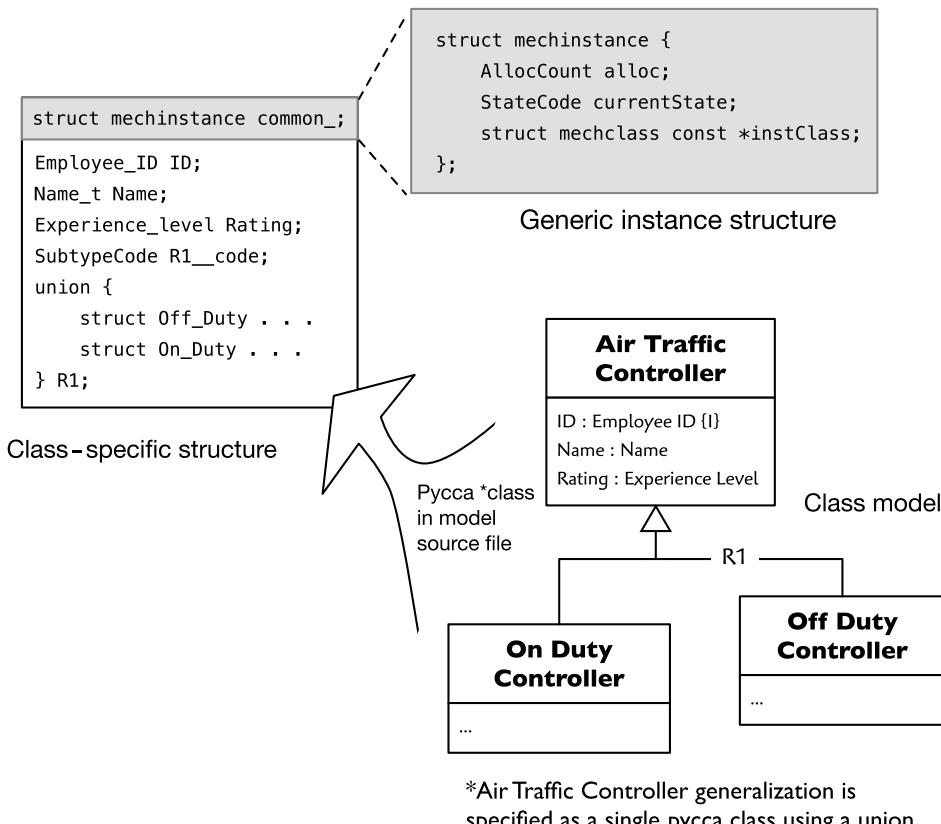


Figure 5-1. Instance data structure

Placing the `mechinstance` structure as the first member of every class structure declaration allows a pointer to an arbitrary instance of an arbitrary class to be treated as a pointer to a generic instance. The technique is common when using C in a more object-oriented programming style, however type unsafe it may appear. We have seen this structure before, in the generated code from pycca. The instance structure defines generic elements applicable to all instances and links the class instance back to data describing its behavior.

The nonzero value for the `alloc` member determines whether a particular instance is currently in use. The value of the `alloc` member is changed each time instance storage is allocated so that we can determine whether it is being reused after a create-destroy-create sequence. The `currentState` member holds a small integer value that is the state in which the instance currently resides. The `instClass` member is a pointer to data that applies to all instances of the class.

Figure 5-2 shows what an Air Traffic Controller looks like with some example data values.

Example Dynamic Instances in Memory

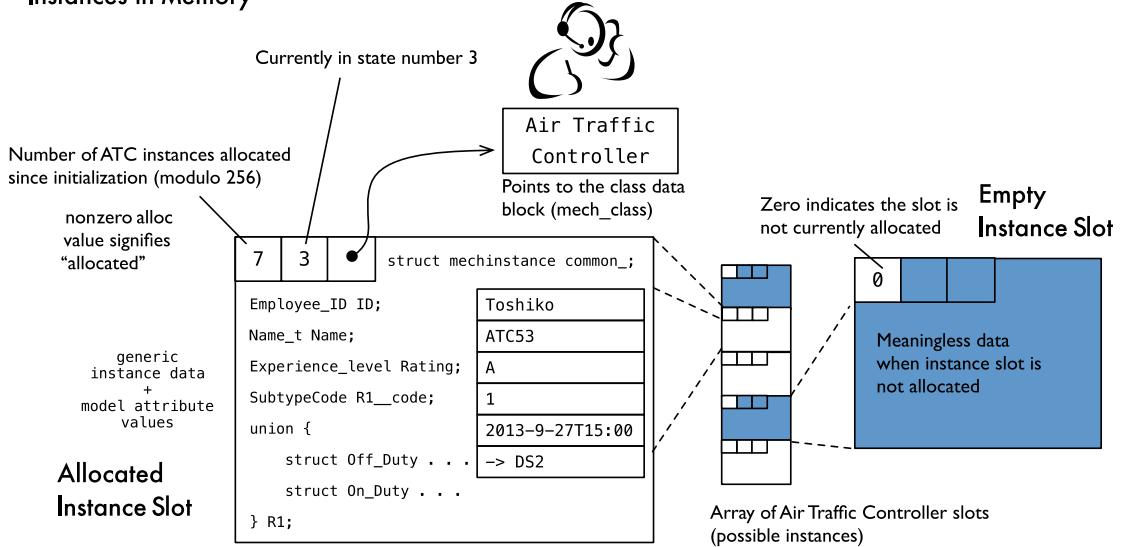


Figure 5-2. Dynamic instances in memory

ST/MX provides all the necessary functions to manage dynamic instance creation and deletion. We postpone the explanation of how instance memory is managed. For now, you just need to know that ST/MX sees an instance as a `struct mechinstance` and ignores the class-specific attributes. The class-specific code ignores the `struct mechinstance` structure. Of course, this is C, and the language itself does not impose any restrictions on access to structure members.

Managing Execution

ST/MX is specifically designed for bare-metal microcontroller-based systems. Its execution capabilities match those provided directly by the hardware, namely:

- There is a single thread of execution provided by a single processing core.
- An interrupt may preempt the execution of the single thread, vectoring to code that services the interrupt. Once serviced, the preempted thread is resumed.

In Chapter 2, we discussed the concept of run to completion. The model execution rules require each activity complete before any other activity can run. ST/MX handles run to completion by deferring the delivery of events until no state activity is executing. The simple mechanism of queuing signaled events and then dispatching them from the queue accomplishes this.

ST/MX provides the C `main` function, which organizes the flow of execution as shown in Figure 5-3.

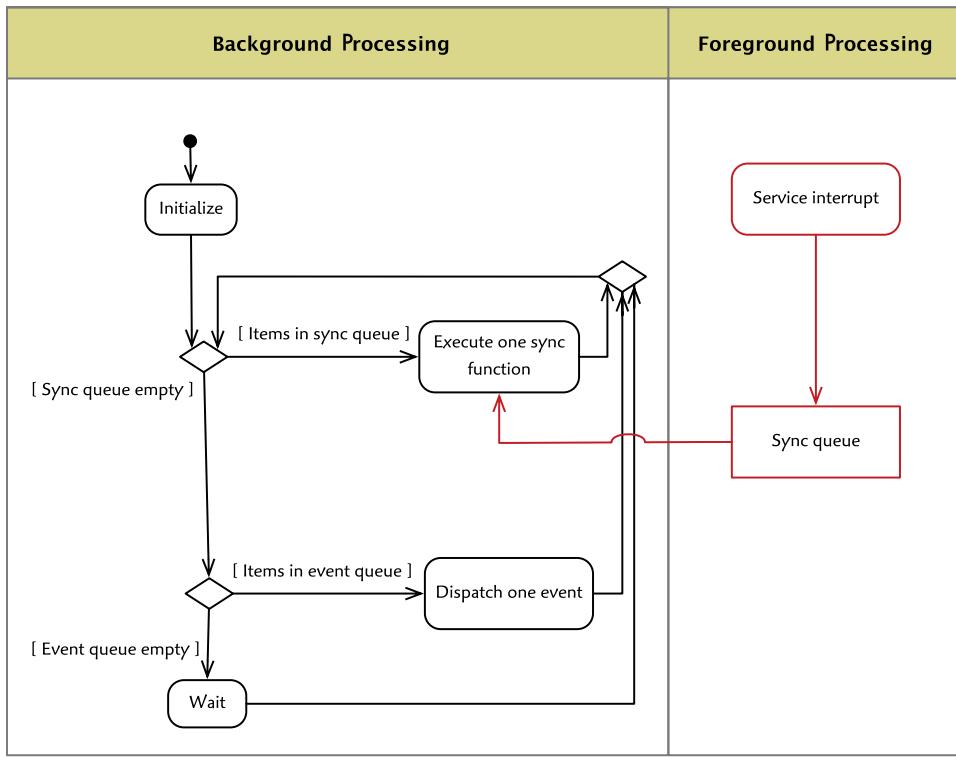


Figure 5-3. Main loop activity diagram

After initialization, execution proceeds in an infinite loop servicing the sync and event queues. The *sync queue* is a queue of function execution requests posted by interrupt service routines. The *event queue* is a queue of events signaled by domain activities. Most of the rest of this chapter is devoted to how the event queue operates. The flow diagram shows that *sync functions* are executed one at a time until the sync queue is empty. When there are no more sync functions, a single event is removed from the event queue and dispatched. Interrupts are enabled and can be serviced when either a sync function or a state activity is executing. If there is no sync function to execute nor any event to dispatch, then the execution waits. Execution waits until an interrupt arrives and, after the interrupt processing has completed, waiting stops returning control flow to the top of the main loop. The single thread of execution that is preemptable by interrupts provides the familiar concept of foreground vs. background processing.

ST/MX does not support any notion of *processor context* or *task* as they exist in real-time operating systems (RTOS). There are no semaphores, message queues, or controls over multiple execution threads. Sync functions and state activities always execute to completion before the next computation is considered. Interrupts are always active, except for small critical sections associated with access to the sync queue.

You may be thinking that the execution scheme of ST/MX does not accommodate long duration computations very well. If so, you are correct. When a state activity goes compute bound for a long time, the rest of the system is not responsive, because all must wait for access to the single processor in the system and there is no preemption of state activities other than interrupt service. Interrupt service may queue requests for code to be executed only when no state activity is running (and for servicing the interrupting hardware as required) and is specifically *not allowed* to access the event queue.

If this characteristic of execution management will not meet the computation demands of the modeled domains, ST/MX may not be the appropriate computing technology for the application, and you will need an MX domain that either shares the processor in a different way or operates on a multicore processor. That said, numerous applications fielded on bare-metal microcontrollers are reactive in nature: they detect a stimulus from the environment, usually via an interrupt, and generate an immediate response, often to control some aspect of the real world. For those types of applications, single-threaded execution domains such as ST/MX work very well. They use computer resources efficiently because their capabilities are closely aligned to those provided directly by the microcontroller hardware.

Event Control Block

When events are signaled by an activity, they are represented by an event control block (ECB). You can think of it as a form that must be filled out for each generated signal. Here is the ECB data structure:

```
typedef struct mechecb {
    struct mechecb *next ;
    struct mechecb *prev ;
    RefCount referenceCount ;
    EventCode eventNumber ;
    AllocCount alloc ;
    MechEventType eventType ;
    union {
        MechInstance targetInst ;
        MechClass targetClass ;
    } instOrClass ;
    MechInstance srcInst ;
    MechDelayTimedelay;
    EventParamType eventParameters ;
} *MechEcb ;
```

- `next` and `prev` members are pointers that implement a doubly linked list. An ECB is queued to one of three lists managed by ST/MX.
- `referenceCount` ECBs are reference counted, and that count is held in the `referenceCount` member. Reference counting ECBs is an advanced feature of ST/MX used in certain periodic signaling situations. We do not use reference counts in our examples.
- `eventNumber` is a small integer encoding the event number. Event numbers are zero-based sequential integers.
- `alloc` is a small integer number used to ensure that an event is not dispatched to an instance that no longer exists. This is discussed later in this chapter.
- `eventType` encodes the type of the event. For now, assume that this is an event that attempts to trigger a state transition. The other event types, creation and polymorphic, are discussed in Chapter 9.
- `instOrClass`, for transitioning events, holds a `targetInst` that points to the instance that will receive the event.
- `srcInst` is a pointer to the instance signaling the event.

- delay is used for delayed signals (as we discuss later in this chapter).
- eventParameters contains the values of any parameters specified by the event.

An ECB is queued to one of three lists managed by ST/MX:

- A *free queue* is a list of blank ECBs available for use in signaling events. ECBs are limited resources queued here when not in use.
- The *imminent event queue* is a list of filled-out ECBs awaiting imminent dispatch.
- The *delayed event queue* is a list of filled-out ECBs to be delivered at a designated time in the future. These represent delayed signals.

Signaling an Event

The action of signaling an event is translated to the following procedure:

1. Obtain a free ECB.
2. Fill in the ECB with the required information.
3. Queue the ECB into the appropriate event queue (imminent or delayed).

These steps are demonstrated in Figure 5-4.

Signaling during runtime

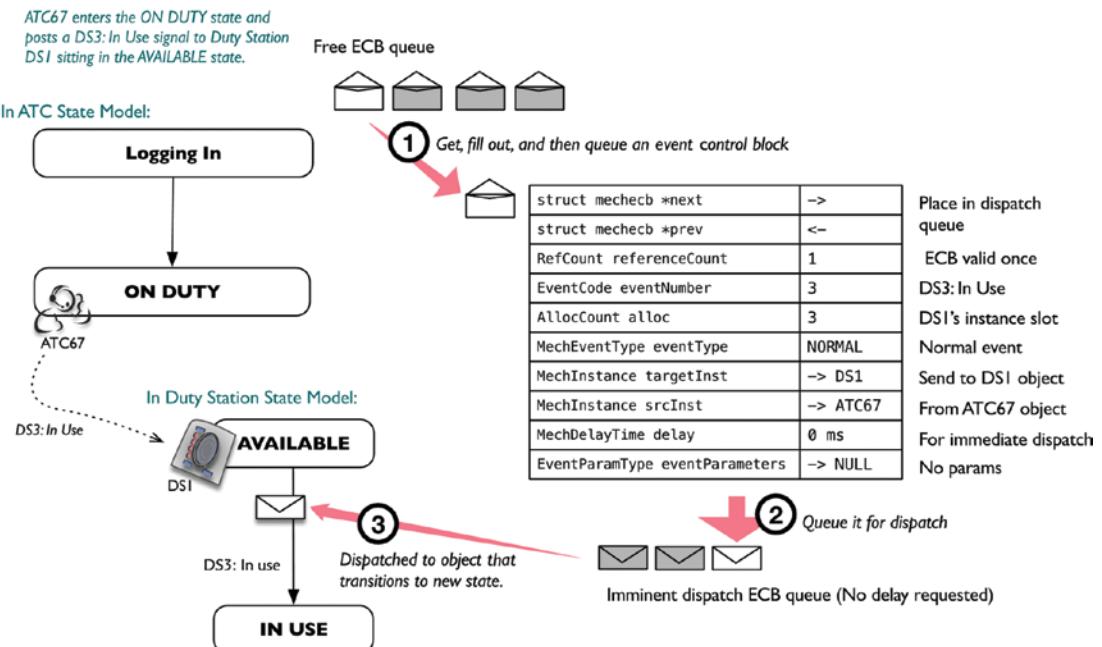


Figure 5-4. Signaling an event

The `eventNumber` member holds a zero-based sequential integer encoding of the event. Event encoding is on a per pycca class basis, so each pycca class has its own set of events, and each set is sequentially encoded starting at zero. This choice of encoding means that the event number may be used as an index into an array, and pycca generates the numbers to ensure proper indexing.

The `srcInst` is a pointer to the instance that is signaling the event. If the event is signaled outside an instance context (by a domain function or a class-based operation, for example), then `srcInst` is set to `NULL`.

Any parameters of the event are placed at the end of the ECB in the parameters block. Parameters are passed by value. Typically, only small amounts of data are passed as event parameters. Passing references to data imposes the additional burden of managing the storage allocated to the parameter values. ST/MX does not provide any facilities to manage event parameter data passed by reference.

Catching the Event-in-Flight Error

The `alloc` member of the ECB deserves special comment. ST/MX must be able to detect when an event is dispatched to an instance that no longer exists. It is, for example, possible for an event to be signaled to an instance and for that instance to be destroyed *before* the event is delivered. Worse yet, it is possible for a new instance to be created in the same memory location as a destroyed instance before the event is delivered! This is known as the *event-in-flight* error and is one of the few runtime errors diagnosed by ST/MX. It is considered an analysis error because it is the modeler's responsibility to ensure proper synchronization so that an instance is not deleted while any event destined for it is still in flight.

Because the consequences of delivering an event to a nonexistent instance or the wrong instance are severe, ST/MX takes steps to prevent the condition. To diagnose the error, when an event is signaled, the `alloc` member value is set to the same value as the `alloc` member of the target instance. When the event is delivered, the two values must still agree. Because each time an instance is allocated it receives a new value for the `alloc` member and an instance that is free has an `alloc` member value of 0, it is possible to detect when an instance is destroyed while it has an event in flight. For the purposes of dispatching an event, the `alloc` member supplements the identification of the target instance provided by its memory address alone.

After filling in the values of the ECB members, the ECB is queued for dispatch. The state machine rules require that an event directed by an instance to itself is dispatched before any events signaled from other instances. So, *self-directed* events are placed at the front of the event queue, and *non-self-directed* events are queued to the rear.

Delayed Signals

ST/MX supports the notion of a *delayed signal*. Delaying a signal is a request to have the event dispatched no sooner than the requested delay time. Delay times are specified in milliseconds, and the delay time may be zero (which results in the event being posted immediately to the imminent event queue). Delayed events are posted to a separate, *delayed event queue* while they are awaiting delivery. In Executable UML, there may be only one outstanding delayed event of a given event number between any sending/receiving pair of instances (which need not be distinct—that is, a delayed event may be self-directed). This limits the number of identical outstanding delayed events to one. ST/MX interprets any attempt to post a duplicate delayed event as a request to cancel the first delayed event and to post the new one delayed by the new time. This is the most convenient interpretation in practice.

The `delay` member of the ECB is used to implement delayed dispatch. When inserted into the delayed event queue, the `delay` member stores the time difference between events rather than the actual requested time delay. During insertion, the `delay` member of an ECB is set to the additional amount of time the inserted event requires that is beyond the cumulative time of the previous events in the queue. This results in the delayed event queue being ordered by increasing value of the time differences.

Delayed event dispatch uses a single timer that expires at the time indicated by whatever ECB is at the head of the delayed event queue. When this time expires, the ECB is transferred to *the rear* of the imminent event queue, and the timer is then reset to the time indicated by the ECB now at the head of the delayed event queue. Even self-directed delayed events are placed at the rear of the imminent event queue. This ST/MX mechanism embodies the idea that a delayed event is simply a request for a normal event to be delivered at a later time, even if that request comes from the same instance that will receive the event.

Figure 5-5 shows an example.

Signaling a Delayed Event

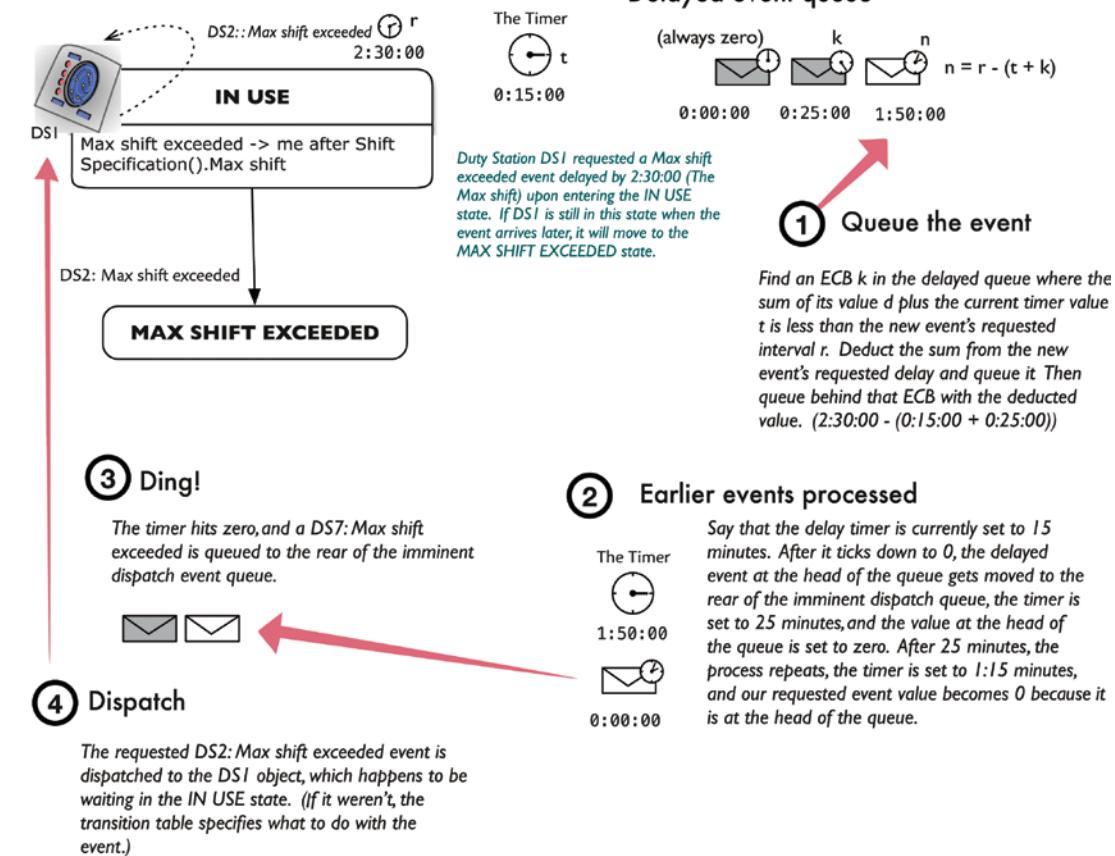


Figure 5-5. Processing a delayed event

In the preceding example, the current timer is set to 15 minutes. The time difference value of each ECB in the delayed event queue is relative to the next earliest delivery, so the head of the queue always has a zero value. When the timer counts down to zero, the ECB at the head of the delayed event queue is transferred to the rear of the imminent event queue, where it now represents a normal event awaiting to be dispatched.

The timer is then loaded with the delay value from the next entry, in this case 25 minutes, and the timer resumes counting down. So, looking at the figure, you can see that the second ECB will be delivered in 15 min + 25 min = 40 minutes. The third ECB is dispatched in 40 min + 1 hr 50 min = 2 hr 30 min, which happily matches that ECB's requested delivery delay.

Boundary conditions and the need to start and stop the timing resource make the implementation logic somewhat more complicated than this description implies. The ordering of the queue implies that inserting into the delayed event queue requires a search to find the appropriate place, accounting for the cumulative times associated with entries already in the queue. The design trades off computation at insertion time for a simpler operation when delay times expire. Note that there is no *periodic* wake-up to scan the delayed queue. Waking up to determine that there is no action to perform is undesirable in a battery-operated environment.

To implement delayed signals, ST/MX must have access to a timing resource. This is necessarily system-specific and usually means that a timer peripheral of the microcontroller is dedicated to running the delayed event queue. Care must be taken to ensure that the timer continues to run even if the microcontroller is halted or placed in a low-power mode while waiting. Consequently, ST/MX delegates low-level timer operations to a set of functions that can be adapted to the specifics of the microcontroller timer peripheral.

Delayed signals may be canceled. ST/MX guarantees that after canceling a delayed signal, it will not be delivered. This implies that canceling a delayed signal would remove it from the delayed event queue or the imminent event queue if it had already expired. It is not an error to cancel a delayed event that has not been signaled or has already been delivered (that is, failing to find the requested delayed event during the cancel operation is not an error).

Finally, ST/MX supports requesting the amount of time remaining for a delayed event. This duration can be used to adjust the delay time.

Event Dispatch

Now that you know how immediate or delayed events are queued when signaled, we can examine how these events are later dispatched. Figure 5-3 shows when in the main loop that event dispatch occurs. Figure 5-6 illustrates the data structures we need to navigate to get everything required to support event dispatch.

Finding what we need to process an event

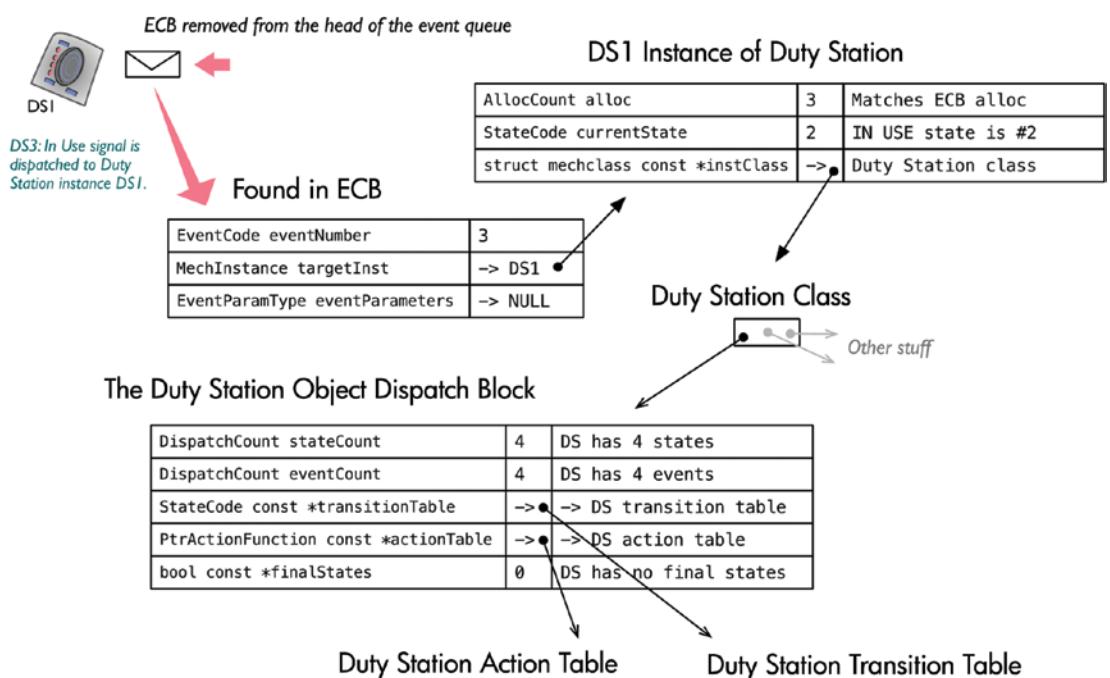


Figure 5-6. Locating the transition and action tables

Event dispatch starts by removing the ECB at the front of the imminent event queue. (From now on, we'll just call it the *event queue*.) We follow the `targetInst` member of the ECB to get to the instance that will receive the event. Each instance has a `currentState` member in its structure as well as a pointer, `instClass`, to data about its class. The class structure contains data about a class that applies to all of its instances. For dispatching events, we are interested in the *object dispatch block* (ODB). The ODB is part of the data generated by pycca for a class. You saw this data in Chapter 4, when we showed the pycca output of translating a state model. From the ODB, we obtain the *transition table*, a matrix of states and events, and the *action table*, an array of function pointers indexed by state number. After the destination state of a transition is determined, a pointer to the code to be executed is obtained from the action table. The C definition of an object dispatch block is as follows:

```
typedef struct objectdispatchblock {
    DispatchCount stateCount ;
    DispatchCount eventCount ;
    StateCode const *transitionTable ;
    PtrActionFunction const *actionTable ;
    bool const *finalStates ;
} const *ObjectDispatchBlock ;
```

The `transitionTable` is a `stateCount`-by-`eventCount` matrix held in a one-dimensional array of `StateCode` type values. The new state value is obtained by indexing into the `transitionTable` by using the instance `currentState` value and the `eventNumber` from the ECB of the dispatched event.

The new state is then used as an index into the `actionTable` array to find a pointer to an action function (a more UML-correct term is *activity function*, because it leads to all actions of a state activity, but we'll stick to the terminology used in the code to minimize confusion). The `actionTable` member is a pointer to an array of `stateCount` number of pointers to action functions with this signature.

```
typedef void ActionFunction(void *const inst, void *const params) ;
typedef ActionFunction *PtrActionFunction ;
```

Figure 5-7 illustrates the indexing process.

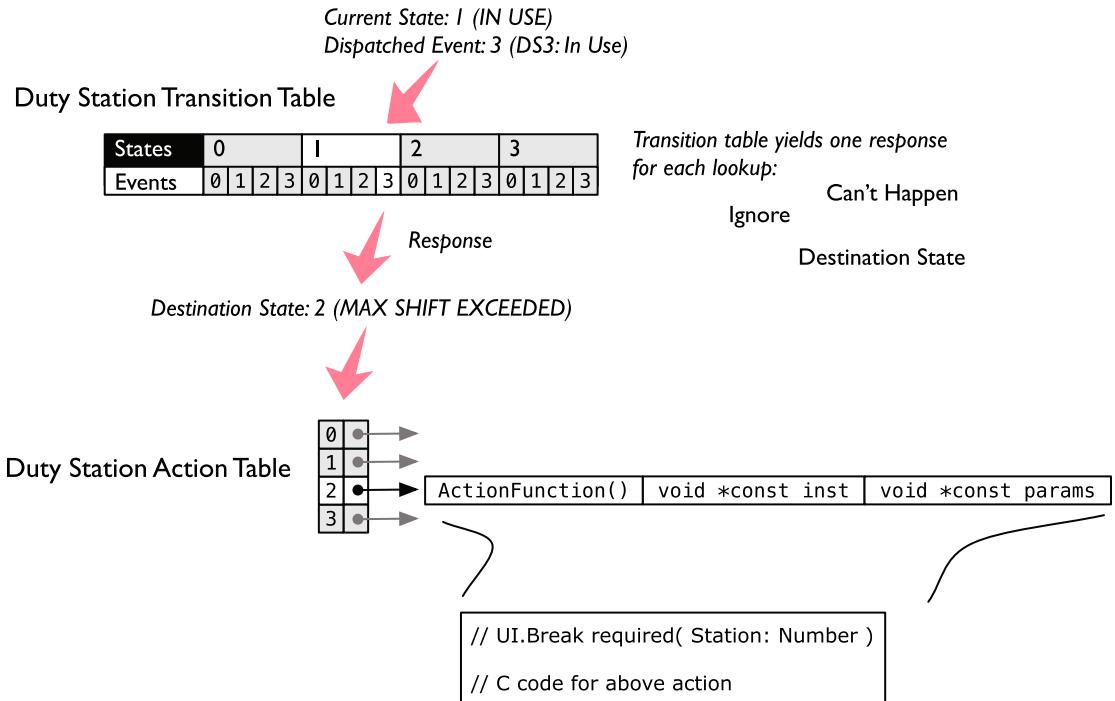


Figure 5-7. Processing a dispatched event

The action function is invoked with the value of `targetInst` and a pointer to the `eventParameters` ECB member. The action function arguments are typed as void pointers so that the data type of the action table can be the same for all classes. Pycca adds code to the beginning of the state activity to cast these pointer values back to the correct type of the instance and its parameters.

Upon return from the action function, the `finalStates` array is consulted. The `finalStates` member is a pointer to a `stateCount`-sized array and, if it is not NULL, holds a Boolean indication as to whether the new state of the instance is a *final* state. If the instance enters a final state, it is automatically deleted after the action function is invoked.

Two other transition possibilities exist. The new state value can be either *ignore* or *can't happen*. When the new state is *ignore*, no transition happens, and the event is simply discarded. A *can't happen* transition is an error. The transition is deemed logically impossible, and should it occur at runtime, results in what can be considered a *panic* situation.

Figure 5-8 is an overview of how all the key pieces of data necessary to process an event are related and how the current state and event are used to find the transition and execute its activity.

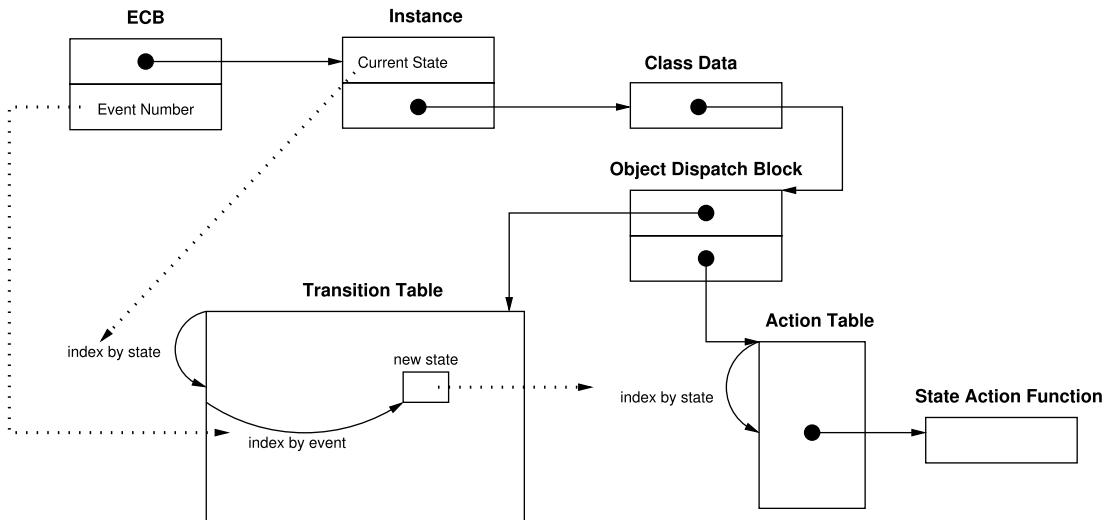


Figure 5-8. Data used in dispatching an event

To tie together these figures and, in the spirit of having no mysterious gaps, we present the C code from ST/MX that dispatches an event. At this stage, the ECB has already been removed from the event queue. The following code computes the new state, invokes the state activity for the transition, and handles all the dispatch rules:

```

static void
dispatchNormalEvent( MechEcb ecb)
{
    MechInstance target = ecb->instOrClass.targetInst ;
    ObjectDispatchBlock db = target->instClass->odb ;

    // Test for corruption of the current state or event number.
    assert(db->stateCount > target->currentState) ;
    assert(db->eventCount > ecb->eventNumber)
    // Check for the "event-in-flight" error. This occurs when an instance is
    // deleted while there is an event for that instance in the event queue.
    // For this architecture, such occurrences are considered as runtime
    // detected analysis errors.
    if (target->alloc != ecb->alloc) {
        mechFatalError(mechEventInFlight, ecb->srcInst, target, ecb->eventNumber) ;
    }
    // Fetch the new state from the transition table.
    StateCode newState = *(db->transitionTable +
        target->currentState * db->eventCount + ecb->eventNumber) ;
#    ifdef MECH_SM_TRACE
    // Trace the transition.
    traceNormalEvent(ecb->eventNumber, ecb->srcInst,
        ecb->instOrClass.targetInst, target->currentState, newState) ;
#    endif
  
```

1

```

// Check for a can't happen transition.
if (newState == MECH_STATECODE_CH) {
    mechFatalError(mechCantHappen, target, target->currentState, ecb->eventNumber) ;
} else if (newState != MECH_STATECODE_IG) {
    // Check for corrupt transition table.
    assert(newState < db->stateCount) ;
    // We update the current state to reflect the transition before
    // executing the action for the state.
    target->currentState = newState ;
    // Invoke the state action if there is one.
    PtrActionFunction action = db->actionTable[newState] ;
    if (action) {
        action(target, &ecb->eventParameters) ; ②
    }
    // Check if we have entered a final state. If so, the instance is deleted.
    if (db->finalStates && db->finalStates[newState]) {
        mechInstDestroy(target) ;
    }
}
// Return the ECB to the pool.
mechEventDelete(ecb) ;
}

```

① Here we compute the new state from the current state and the event by indexing into the transition matrix. The transition matrix is held as a one-dimensional array of state codes. This is necessary so that all classes can have the same data type for the transition matrix, regardless of the number of states and events that are defined for the class state model. Because the transition matrix is held as a one-dimensional array, we have to undertake the index computation that segments the transition matrix into a two-dimensional matrix accessed by state and event (that is, we have to scale the “row” access by the number of events in the state model). Pycca ensures that the transition matrix cells are ordered in this manner.

② Invoke the state activity function associated with the new state. We allow for empty state activities to have a NULL function pointer and avoid the call/return overhead.

This short piece of code accomplishes the majority of the execution sequencing in ST/MX and is, except for the main loop code, by far the most frequently executed code in ST/MX.

Tracing Execution

A state model diagram does an excellent job of showing the possible ways that the life cycle of an instance can evolve and the processing that takes place during that evolution. When the domain runs, instances make transitions according to the sequence of signals that are directed to the state machine for that particular instance. So, it is insightful to see the chronological sequence of transitions made by an instance.

ST/MX can trace the event dispatch for you. Because tracing has an impact on the execution speed and code size of a program, it can be conditionally compiled in during testing phases and removed from a delivered release. The following is a text log of the event dispatch that corresponds to the execution scenario

of the Air Traffic Control model we discussed in Chapter 2. This log was generated during an actual run of the translated domain:

```
2016/06/09 11:17:12.112: (nil) - Ready_for_duty -> Air_Traffic_Controller.atc53: ←,
    OFF_DUTY -> Verifying_Adequate_Break
2016/06/09 11:17:12.113: Air_Traffic_Controller.atc53 - Log_in -> ←,
    Air_Traffic_Controller.atc53: Verifying_Adequate_Break -> Logging_In
2016/06/09 11:17:12.113: Air_Traffic_Controller.atc53 - Logged_in -> ←,
    Air_Traffic_Controller.atc53: Logging_In -> ON_DUTY
2016/06/09 11:17:12.113: Air_Traffic_Controller.atc53 - In_use -> Duty_Station.s3: ←,
    AVAILABLE -> IN_USE
```

In this log, four events were dispatched. Each dispatch is timestamped. The trace consists of the instance that signaled the event, the name of the event, and the instance that received the event. If the event was signaled outside an instance context, the source of the event is given as (nil). In this case, the scenario was initiated by signaling Ready_for_duty from outside the domain and, hence, not from any instance. Otherwise, instances are identified by their class name and the name given to the instance as part of the initial instance population. The last two fields of the trace show the transition from the current state of the instance, when the event was received, to the new state of the instance, after the transition.

The transitions in the log show the same sequences described in Chapter 2. The Ready_for_duty event causes a sequence of transitions for the Off Duty Controller, atc53, as the instance goes from OFF_DUTY to Verifying_Adequate_Break to Logging_in and finally arriving at the ON_DUTY state. The last log entry shows the Duty_Station transitioning to the IN_USE state.

Tracing execution makes it easier to understand how the domain runs and supports documenting the results of testing. Because a state model is a directed graph, it is possible to calculate a set of event sequences guaranteeing that every state activity is executed at least once (this is related to a spanning tree of the state model graph). Determining such a set of event sequences can be done with no knowledge of what the activities compute. The structure present in a state model can help determine the minimum set of test cases necessary to meet a project's quality goals. The transition traces can provide objective evidence of the actual execution of test cases.

Running in a POSIX Environment

The ST/MX execution domain is available to run on three platforms (and can be adapted to other microcontroller platforms):

- ARM Cortex-M3
- TI MSP-430
- POSIX (Portable Operating System Interface)

It may seem unusual to target microcontroller-based systems and supply a version of the runtime code for POSIX. Although you can build a POSIX application by using pycca, the primary purpose of the POSIX version of ST/MX is to support *cross-platform development*. This sort of development is essential when developing for a microcontroller. Here the models are translated and run in a simulation model execution environment on a desktop computer. After the models are fully exercised, they can be compiled, integrated, and further tested on the target platform.

A microcontroller does not have the capabilities to support compilers and other build environment elements. So developers are faced with building on a desktop system and downloading the code to an embedded platform. Microcontrollers do usually have debugging facilities, and most microcontroller development environments support some form of source-level debugging.

However, there is rarely any file I/O or persistent storage of any great size. Usually, you have to be content with simple `printf()` style debugging. There is a quandary about how much effort should be spent improving the target environment and whether that effort would be better spent running the code in simulation. This is particularly acute when the application is controlling physical entities and it is impractical or unsafe to suddenly stop when a break point is encountered. For example, a moving robot arm involves actual physics that must account for momentum. Simply stopping the processor at a break point could do physical harm. In such cases, debugging by examining execution trace data is typically the only practical solution.

For domains that are properly bridged to any system-specific facilities (for example, hardware peripherals), compiling and linking against the POSIX version of the runtime will yield an executable that may be run in most UNIX-like (for example, Linux), macOS, or Windows under Cygwin systems. Because these development systems have lots of storage and I/O, it is possible to build test fixtures or test benches to exercise the domain code outside its targeted environment. For example, compilers can be requested to track code coverage, and the coverage of unit tests can be measured. It is also much easier to deal with state machine traces when there is a place to store them. Tracking the transitions of the state models is a convenient way to measure how much of the state space of a program has been exercised. Test sets to drive the state model transitions can be obtained by inspection of the state diagrams of the model. The ability to run in a POSIX environment also opens up the possibility to use continuous integration and regression testing tools that are available. To run under simulation, it is necessary to stub or simulate those parts of the application that are system specific. That can be a large undertaking, especially when interaction with the real world is a major driver of the application execution. Project teams must evaluate the costs and benefits.

Of course, not all testing can be run in a simulation environment. Running in simulation is convenient for testing, recording the logic of the program, and driving its execution path with specific test sets. It is still necessary to integrate and perform additional testing on the target microcontroller. Running in simulation divides the initial testing into logic testing and target integration testing, and places logic testing in an environment where results are easier to obtain and record.

Handling Errors

We have also glossed over the way we deal with errors. The ST/MX domain is responsible for setting the policy for handling errors in the system. ST/MX minimizes repetitive error handling by designating all errors it detects as fatal. Consequently, all errors detected by ST/MX trigger the *panic* condition, which prevents further execution. There is no interface to the models indicating the success or failure of ST/MX operations. This is a benefit to the modeler because models signal an event or create an instance and have no need to check error returns or handle potential exceptions. Models simply assume that execution management is perfect. Anything that goes wrong is ST/MX's responsibility to handle.

The ST/MX domain diagnoses two types of errors:

- Errors that arise from internal ST/MX execution
- Errors found at runtime that indicate an error in the model analysis

ST/MX uses several internal data structures—event control blocks, for example—each of which has a fixed quantity. Like class instances, internal data structures are also allocated in arrays whose size is established at compile time. This is in keeping with ST/MX's design goal of *not* performing dynamic memory allocation. Exhausting an internal resource results in a fatal error. The same is true of requests to allocate a class instance when there are no free slots in the class instance storage array.

Analysis errors discovered at runtime are either state machine transition errors or event-in-flight errors. These errors are also fatal.

ST/MX provides a means to supply your own fatal system error handler. This is useful for logging purposes and for other types of external notifications. By default, ST/MX executes the standard C library function `abort()` on a fatal error. In embedded systems, this usually results in a system reset, and the

microcontroller begins execution anew. As draconian as the approach may seem, there is little other practical recourse for most embedded systems. Even if a state activity could know that there were no ECBs available when it attempted to signal an event, what realistic recovery recourse would it have? One cannot simply ignore the fact that a signal could not be sent or a state transition was logically impossible and then blithely continue on hoping for the best. This implies that sizing the required data structure resources must be done carefully. Analysis errors detected at runtime can be flushed out only by comprehensive test cases. Recording the reason for a fatal error can help diagnose problems, but when the system is deployed in the field and ST/MX determines that a severe error has occurred, there is little else to be done but restart and try again.

Fatal software errors may also need to be handled in the context of the system as a whole. Safety-critical systems, such as medical devices, often have dedicated hardware that can take over at least part of the software's function. Most embedded systems have one or more *watchdog timers* that must be reset periodically as a check that the software is functioning. Failure to reset the watchdog in a timely manner can trigger a processor reset through hardware or other fallback strategy. Some systems even have dedicated hardware to catch run-reset-run-reset sequences that happen too frequently and indicate an endless reset loop condition in the software. ST/MX provides the means to know only when the software execution has reached a condition where it can no longer continue. The way that condition is handled in a larger system context must be decided based on a thorough risk analysis.

Summary

In this chapter, we described the inner workings of the ST/MX domain, the runtime component of a pyccia translation. It is the last aspect of the translation needed to produce a running program.

ST/MX manages all the class data and execution sequencing of the system. ST/MX is strictly data driven, and the emphasis in this chapter was on describing how the data structures generated by pyccia are used by the algorithms in ST/MX. ST/MX has a common view of instance data that allows it to perform operations on an instance of any class.

Execution is managed primarily by queuing events for dispatch. A queued design supports the run-to-completion semantics required by model execution rules. Events in the queue are represented by an event control block. Event dispatch may also be delayed to support the need by a model activity to initiate execution at some time in the future.

Instance state machine transitions are computed using a transition table. The transition table is indexed by current state number and event number, and the indexed cell contains the new state to which the transition is made. The activity of the state is a function, a pointer to which is placed in the action table that is an array indexed by state number.

The C code for event dispatch was also given, showing exactly how the state transitions are computed. ST/MX supports producing a chronological trace of event dispatch. The execution scenario from Chapter 2 was run, and the resulting trace log verified, by actual execution, the transitions we inferred from the state model diagram.

ST/MX implements the error-handling policy for the system, and all errors detected by ST/MX are unrecoverable and deemed fatal system errors.

CHAPTER 6



An Extended Example

We presented the simple ATC application as an extended *Hello World* example to demonstrate the fundamental steps in a pyccua translation. That initial application addressed a small portion (workloads and shifts) of an isolated subject matter (air traffic control). It was defined by an integrated set of class, state, and action facets wrapped up in a single executable model package. In this chapter, we begin a new case study that requires coordination of multiple such executable packages which, in Executable UML, are called *domains*.

The code generated for the ATC application works, but it doesn't do much. That's because the application itself is only one portion of any deployed software system. You may have wondered, for example, how radar, user displays, alarm handling, logging, and other essential functionality would be incorporated into the ATC system.

Of course, you could simply extend the application models to incorporate all the required functionality, but in practice, this is a bad idea. The models would sprout excessive complexity, not just in the number of elements, but in the explosion of state combinations and awkward class relationships. The result would be an inconsistent set of abstractions in the model, and the entire analysis effort would bog down. This is symptomatic of what was called *analysis paralysis* at the time when modeling systems in this way was first undertaken.

It is better practice to organize the system into a set of relatively isolated domains. A *domain* represents a distinct set of abstractions focused on a particular subject matter. The ATC models, for example, constituted an application level of abstraction with regard to the subject matter of air traffic controller duties. Radar tracking would be an entirely different subject matter, as would the user interface and alarm handling. You can imagine entirely different class vocabularies in each model set and wholly different relationships and policies within each. The internal workings of each domain become opaque to adjoining domains, with only the fundamental services exposed via a variety of runtime and pre-runtime bindings.

A domain is a fully executable package that can be run and tested on its own. It may be modeled in Executable UML, another modeling language such as Simulink, or simply hand coded. Pyccua can knit together Executable UML and domains hand coded in C. But if you do use another modeling language and can output the result in C code, you can use pyccua to fold in the code.

The many benefits to a domain-based organization of a system (such as reduced complexity, platform independence, firewalls against changes, large-scale reuse, and so forth) are beyond the scope of this book. However, there is one benefit pertinent to model translation worth emphasizing here: domains provide a systematic way to bridge the chasm from the application level of abstraction to a complete implementation. This is done through a process of delegation. What one domain cannot do, it delegates to another. Where one domain lacks expertise, another domain must provide it. This chain of reliance continues until we get to the point where there's nothing left to delegate. In other words, we have a complete running system.

For example, the models in the ATC domain knew when a controller was working too long without a break. But they did not know how to manifest this warning in the real world. That could be the job of a user-interface domain that can make events visible, or audible, or otherwise tangible in the real world.

The UI domain should not have any built-in application-specific knowledge; otherwise, it would have little utility for other applications. Assuming for the moment that the UI domain is implemented as an existing set of libraries, there is no need to model ATC further. It will be necessary to configure only those UI libraries with information supplied by the ATC models and then somehow wire ATC events to the appropriate UI library calls. As you'll see, `pycca` provides the *somewhat*. On the other hand, a radar domain that knows all about tracks, signals, echo profiles, and such may need to be modeled to support the ATC domain with information about air traffic.

Ultimately, the ATC domain, and any other modeled domains, have their models run by the Model Execution (MX) domain, where the execution of models has been delegated. To sum up, when one domain delegates functionality, another domain must fill in the required services. This keeps happening until we hit solid ground, such as the MX domain or another existing domain. There is no unspecified layer where magic happens. All the software is deemed to live in a domain or is bridging code between domains.

From a model translation perspective, we need a way to specify the organization of models into domains and how domains, modeled or otherwise, interact with one another to define a complete software system. Almost all systems built from models consist of a combination of code generated from the models and manually written code, either existing or newly written. To see how all that works, we need to examine and generate code for a multidomain example, which is the purpose of our next case study.

The Automated Lubrication System

The Automated Lubrication System (ALS) is a service that controls the lubrication of mechanical equipment such as a vehicle engine, gear train, or heavy-lifting machinery. Although the applications of automated lubrication can vary considerably, the fundamental idea is a series of grease injectors, installed around and within the user's equipment and controlled according to a programmable cyclic schedule.

Now, were it not for the potential, and likelihood, that things could go dangerously wrong, this would be a trivial system. But the ALS must carefully monitor pressures throughout the system to ensure that equipment is not damaged and that lubrication is performed adequately. The system must react to events that occur in the equipment and may permanently or temporarily shut down lubrication. For example, an idling engine may require different lubrication than one that is revving at high revolutions per minute (rpm).

The operator must be notified about routine maintenance, such as refilling lubricant, as well as minor and serious faults. And the operator must have a way to lock out unwanted injections when a piece of equipment is under maintenance. Care is taken to isolate units of equipment so that one can be locked out without shutting down lubrication everywhere. It is also necessary to support various diagnostic modes so that the operator can experiment with different lubrication parameters. To support a variety of user equipment and applications, critical operating parameters must be configurable.

As an example of one possible user application, Figure 6-1 shows how the lubrication of a wind turbine might be configured.

An ALS Configuration for a Wind Turbine

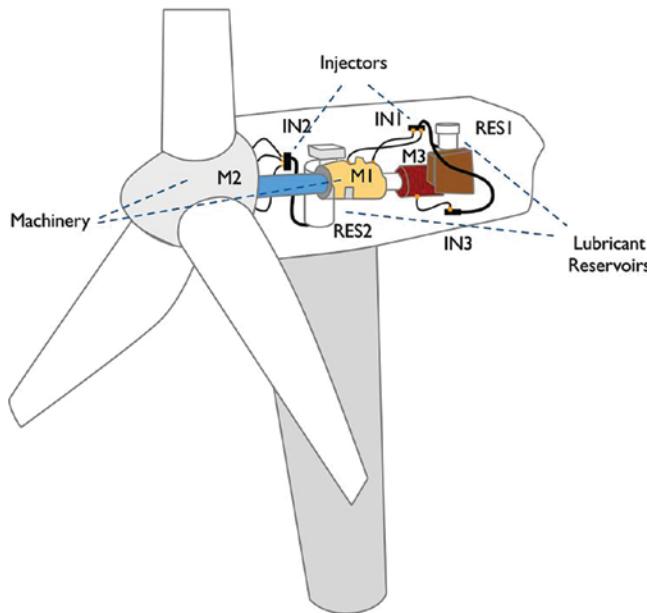


Figure 6-1. Injector layout diagram

This wind turbine is lubricated in five locations delivered by three injectors. Injector 1 (IN1) is dedicated to the gearbox designated as one unit of machinery (M1), while injector 2 (IN2) lubricates the main shaft, designated M2. A third injector (IN3) lubricates a single location on the generator (M3). Note that injectors 1 and 3 not only share the same grease reservoir; they are identical designs. Injector 2, on the other hand, is a different model of injector, designed to deliver a special lubricant under higher pressure for the main shaft.

Each injector is driven by a dedicated lubrication cycle. All injectors are rigged with sensors to detect pressure. Lubrication is started and stopped by enabling and disabling each injector's solenoid. Each injector is supplied by a lubricant reservoir. Injectors 1 and 2 are delivering lubricant to multiple sites. The delivery lines to these sites are not individually controllable, so each injection has the same effect on each line. Note also that each injector is dedicated to a single unit of machinery. This makes it possible to enable a safety lockout on one unit of machinery to stop lubrication without necessarily affecting the lubrication of neighboring machinery.

ALS Domains

Before jumping into the models, let's take a look at the domains necessary to support the ALS, as shown in Figure 6-2.

Two ways of illustrating the domain relationships

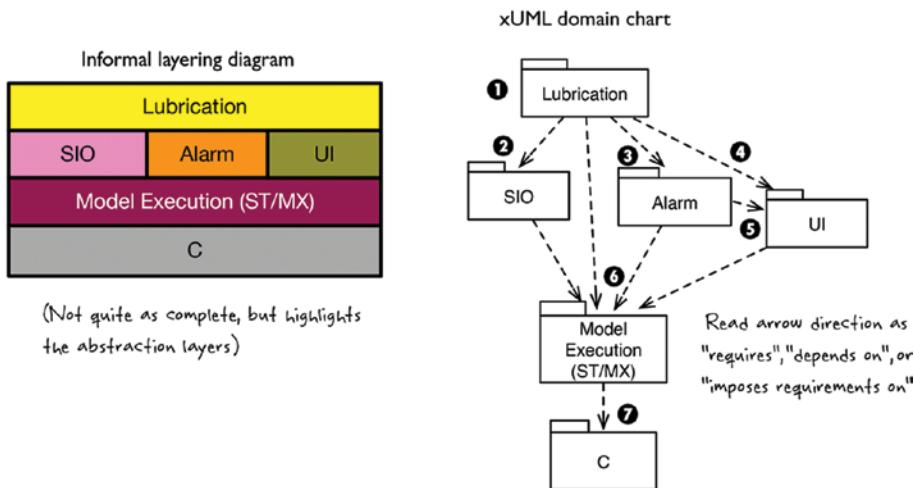


Figure 6-2. ALS domain dependencies

- ➊ At the top level, we have the Lubrication domain. It is concerned with the primary purpose of our system, which is to manage the injectors to schedule cyclic lubrication. This domain is modeled.
- ➋ The Lubrication domain needs up-to-date pressure values and events announcing key pressure thresholds. It also needs a way to convert a modeled event such as “Turn injector on” into an appropriate signal or command to the injector solenoids. Signal I/O (SIO) provides this service as indicated by the dashed dependency arrow. The SIO domain is shown in Chapter 7.
- ➌ The Lubrication domain signals alarm conditions, but needs a way to manage systematic setting and clearing of alarms as well as management of alarm categories such as operator warnings, errors, and log items. The Alarms domain handles this job.
- ➍ The Lubrication domain needs to know when to activate and deactivate lubrication schedules. It also needs a way to signal completion of schedules and other events to a human operator. It uses the UI domain for this purpose.
- ➎ The Alarms domain needs to display alarms to a human operator and allow alarms to be cleared. It also uses the UI domain.
- ➏ All modeled domains are translated to run on the Model Execution domain and need its services to implement model-level actions.
- ➐ The Model Execution domain is coded in C. Yes, the C language is also a domain. This might seem strange at first, but C is its own distinct subject matter. It isn’t modeled in Executable UML, of course, but the C language is modeled to some degree as a grammar. The entire interface to the C domain is defined pre-runtime by writing a program. The more you think about it, the more you realize there isn’t anything special at all about a program language as a subject matter.

As you can see, this arrangement of domains creates a complete path from the application-level abstractions of the Lubrication domain through generic utility services, model execution, and ultimately, code. Each domain, modeled or not, represents a distinct subject matter. Note that domains yield a largely platform-independent partitioning. The underlying hardware, devices, and user-interface technology can change, but the subject matter organization remains intact. Contrast this with many traditional high-level software architecture diagrams, which are almost always platform specific.

Whether you view the domain interactions in a layering manner or as dependencies between domains, it is important to realize that neither view represents a functional partitioning of the system, also known as *functional decomposition*. Subject matter partitioning is based on cohesive classes and relationships without regard to specific functions. Each class model defines a vocabulary and rule set that defines the subject matter of a domain. Dividing a system by common data and rules is markedly different from dividing it by common functions. Domain partitioning is based on subject matter, and the analysis of the subject matter of a domain seeks to find a consistent set of abstractions directed to the role the domain plays in the overall system. The functionality of the system is manifested when the model *executes*, and any given functional feature of the system generally requires multiple domains, each to play its role. For the ALS, injecting lubricant involves both the Lubrication domain to determine the appropriate time and the SIO domain to actuate the hardware. Both domains use the Model Execution domain to interact and are ultimately coded in C.

In Chapter 8, we show how domain dependencies are specified and then implemented using pycc. For the remainder of this chapter, we examine the models in the Lubrication domain. Bridges to the SIO, Alarms, and UI domain are shown as we examine the state models and activities.

Lubrication Domain

We start by examining the Lubrication domain. This is the application level that addresses the primary business purpose of the system. All other domains are present to provide services directly or indirectly in support of the Lubrication domain's needs. Because our purpose is to coordinate and manage lubricant injection, we would expect the application level to *know about* the injection equipment and the lubrication scheduling subject matter.

Lubrication Class Model

For a more readable illustration, we break the Lubrication class diagram into two adjacent pieces, Equipment and Schedule. This split is strictly for presentation purposes and has no implications on the model itself.

Figure 6-3 shows the equipment part of the class model, which focuses on physical components.

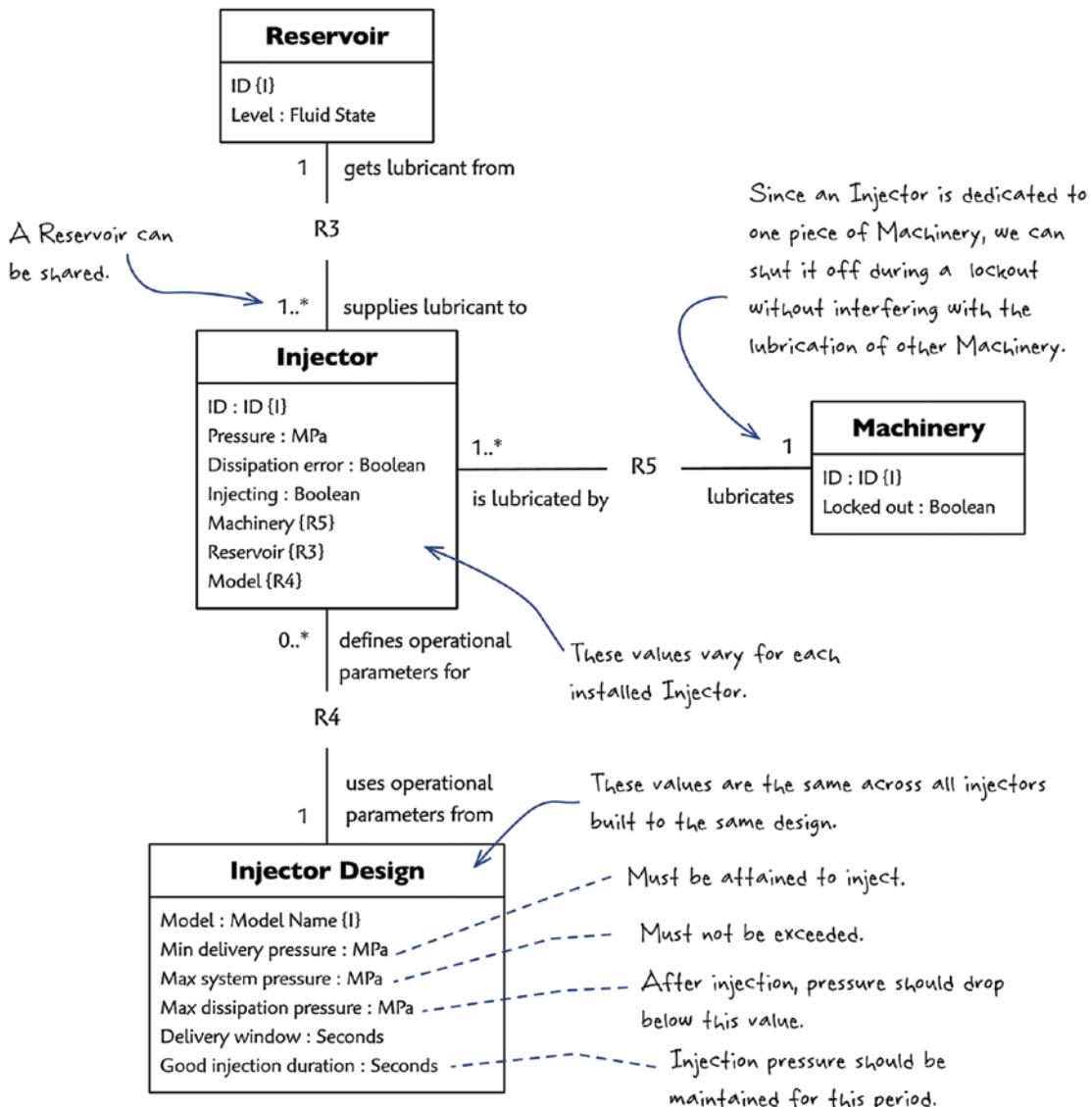


Figure 6-3. Lubrication equipment classes

Lubricant is delivered to some type of equipment. This could be anything from a robot, to a jet engine, to a heavy-lifting construction vehicle. From the perspective of the ALS, the function of the lubricated equipment is not relevant, so it is represented as generic “machinery.” The particular values established for the Lubrication Schedule and Injection Spec attributes are determined, in part, by the specific needs and function of the lubricated Machinery. But this model is not concerned with the specifics of the Machinery and leaves it to a human to define acceptable values for the attributes to meet those needs. Those attribute values appear in the initial instance population for the domain.

Injector Designs

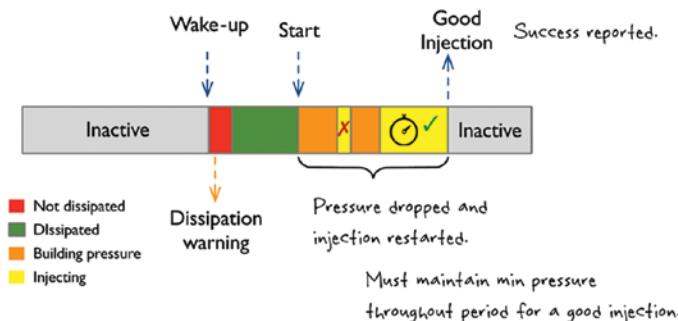
To accommodate a variety of injector designs, the model separates out those features that are particular to an individual injector and those that vary from one design to another. This is done using R4 in a common *specification class* modeling pattern. The detailed reasoning behind this pattern is covered in modeling books—for example, *Executable UML: How to Build Class Models* by Leon Starr (Prentice Hall, 2001) – See note about future update in the bibliography.

The Injectors in the example, IN1 and IN3, are both built to the same design specification, so they share the same Injector Design instance. IN2 is a different model of Injector, with a distinct set of design parameters, so it requires a separate Injector Design instance.

A Single Injection

We have enough of the model to consider the behavior of a single injection. To reduce the load on power and the local computer and network resources, an injector sits quietly until commanded. It does not even monitor internal pressure because it is unlikely that dangerous pressure will build up when the injector is inactive. Figure 6-4 shows a typical injection sequence.

Good injection sequence



Bad injection sequence

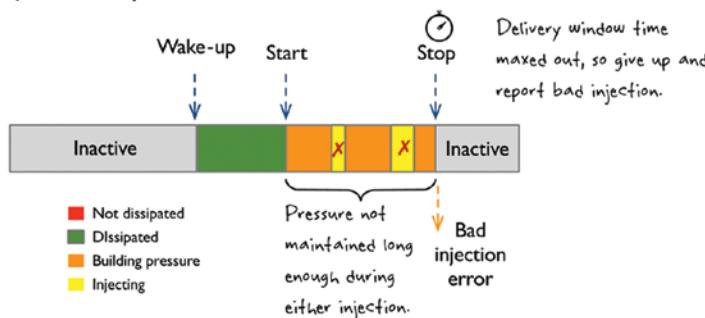


Figure 6-4. Example good and bad injection sequences

The Wakeup command causes the injector to begin monitoring pressure for error-reporting reasons. Ideally, the pressure built up from the previous injection will have dissipated by now, but that is not always the case. If any high pressure is detected, we want to report a warning. Regardless of how much the pressure fluctuates, at most one warning should be issued.

This monitoring interval ends when the Start command arrives. Now it's time to inject lubricant. To do this, the *Injector Design*.*Min delivery pressure* must be achieved and maintained for the *Injector Design*.*Good injection duration*. Upon success, a good injection is reported, and the injector goes inactive again. Otherwise, each time the pressure drops too low, it is necessary to wait until it builds up and try again. If a Stop command is received, it means that too much time has elapsed. A bad injection is logged, and the injector goes inactive.

Controlling Lubrication Cycles

Now that we have a model of the equipment, we need to consider the cyclic schedule by which the injectors are controlled. At its simplest, a cycle of injection repeats with a specified delay between injections. A period of time must also be specified to allow the injector to begin monitoring pressure. We can also specify whether the cycle should run for a default count or continuously until stopped manually. A maximum number of low-lube cycles that will be tolerated before shutting off lubrication is also part of the lubrication program. Figure 6-5 illustrates the typical cyclic control session.

An Autocycle Session

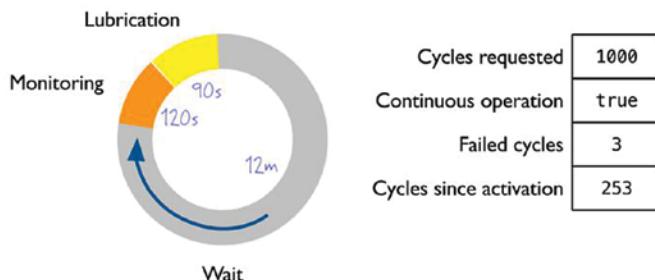


Figure 6-5. A cyclic schedule of lubrication

With these concepts in mind, we can move on to the other half of the Lubrication domain class model, which covers cyclic scheduling and control of injection. Figure 6-6 introduces two new classes to cover the scheduling requirements.

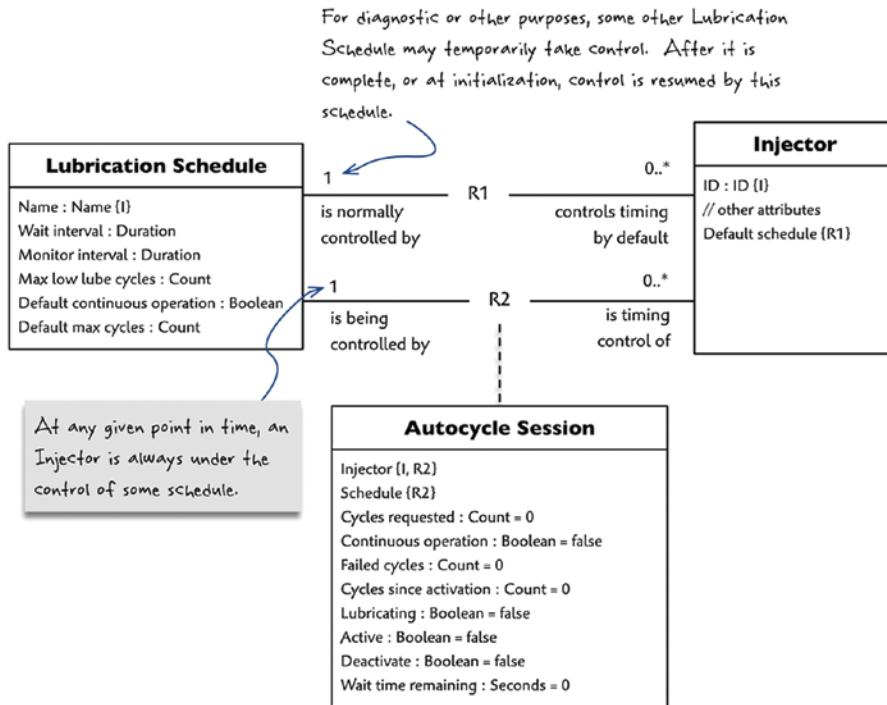


Figure 6-6. The lubrication cycle classes

Lubrication Schedule

This is a program of cyclic lubrication that can be used with any model of Injector, although typically, the values will be set with a particular injector model and usage in mind. Each has a name and most important, the main interval, which determines the wait time between injections (Wait interval), and a prep interval, which determines the delay between wake-up and start (Monitor interval), is defined. If the operation should continue without any specific total, *Default continuous operation* is set to true. A default maximum number of cycles can be specified (Default max cycles), but it has no function if continuous operation is the default. Both of these defaults can be overridden in an Autocycle Session.

Autocycle Session

R1 designates a default Lubrication Schedule to use for each Injector. This association is consulted to choose an instance to relate along R2, where only one Lubrication Schedule can be controlling an Injector at a time, though the same Lubrication Schedule could be in use simultaneously. After a Lubrication Schedule has been selected and related across R2, an instance of Autocycle Session is created. The Autocycle Session tracks timing for a specific Injector along with a count of failed cycles.

The reason for R1 is that it is sometimes necessary to choose a nondefault Lubrication Schedule for a few cycles, often for diagnostic purposes. In that case, the previous Autocycle Session is deleted, and a new one, related to the nondefault Lubrication Schedule, is created (an Injector is controlled by only one Autocycle Session at a time). Once the temporary session has completed, usually by running through a requested number of cycles, control automatically resumes with the default Lubrication Schedule found on R1.

Example Population

The scenario illustrated in Figure 6-1 is represented by the population shown in Figure 6-7.

Lubrication Schedule								
Name	Wait interval	Monitor interval	Max low lube cycles	Default cont operation	Default max cycles			
Shaft	90 sec	30 sec	10	true	10000			
Gearbox	210 sec	45 sec	8	true	5000			
Generator	120 sec	25 sec	10	true	10000			
Test2	20 sec	15 sec	1	false	200			

Autocycle Session								
Injector	Schedule	Cycles requested	Cont operation	Failed cycles	Lubricating	Active	Deactivate	Wait time remain
IN1	Gearbox	0	true	0	false	true	false	90 sec
IN2	Shaft	0	true	1	true	true	true	0 sec
IN3	Generator	0	true	0	false	true	true	0 sec

Reservoir								
ID	Level							
RES1	normal							
RES2	low							

Injector								
ID	Pressure	Dissipation error	Injecting	Default schedule	Machinery	Reservoir	Model	
IN1	20.3 MPa	false	false	Gearbox	M1	RES1	IX77B	
IN2	0.0 MPa	false	false	Shaft	M2	RES2	IHN4	
IN3	0.0 MPa	true	true	Gearbox	M3	RES1	IX77B	

Injector Design								
Name	Min delivery pressure	Max system pressure	Max dissipation pressure	Delivery window	Good inj duration			
IHN4	19 MPa	35 MPa	32 MPa	90 sec	9 sec			
IX77B	15 MPa	26 MPa	26 MPa	120 sec	11 sec			

Machinery								
ID	Locked out							
M1	false							
M2	false							
M3	false							

Figure 6-7. Scenario's initial instance population

State Models

The Autocycle Session, Injector, and Reservoir classes each have a state model. Let's start by walking through the Injector state model. Again, the model is split to make the discussion easier.

Injector State Model

Figure 6-8 shows the top half of the Injector state model.

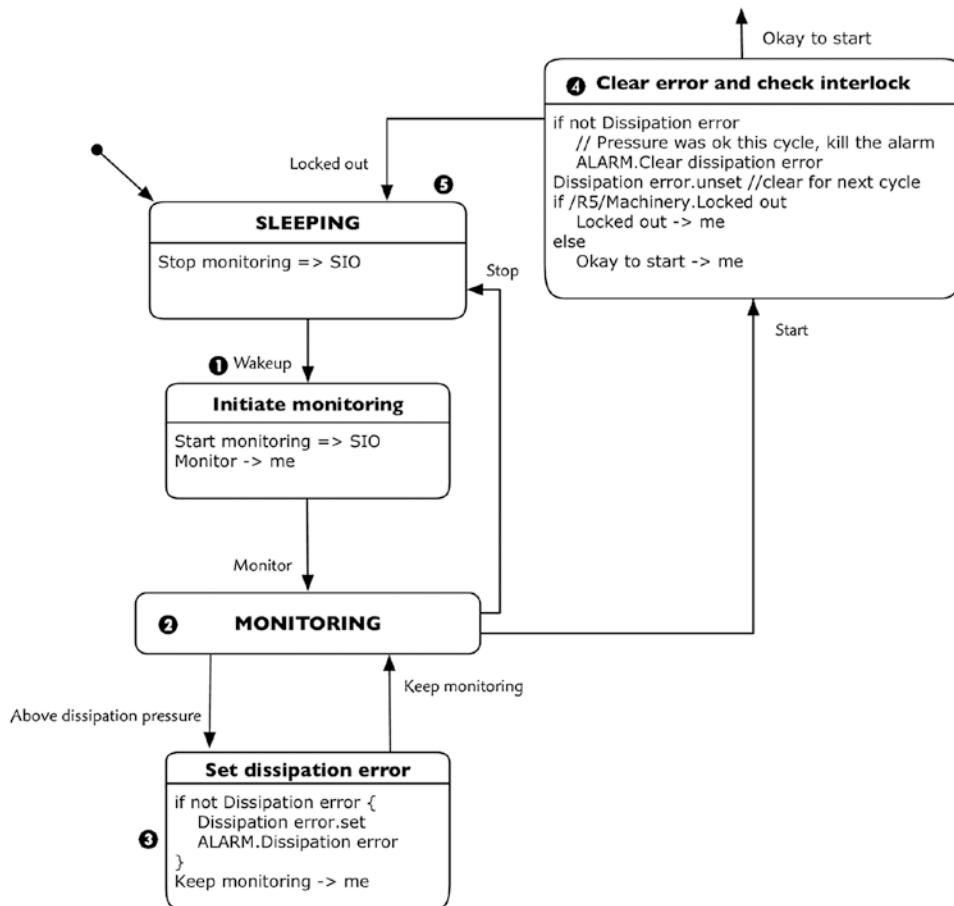


Figure 6-8. Injector state model excerpt: Sleeping to Active

- ① The Injector is in the SLEEPING state when it receives a Wakeup event, which will be sent from the Autocycle Session. The SIO domain is told to start monitoring sensor values for this Injector.
- ② In the MONITORING state, the Injector isn't doing anything, because SIO is doing all the sensor detection work. But, from this state, we react to the *Above dissipation pressure* event supplied by the SIO domain when it detects that pressure has risen above the *Injector Design. Max dissipation pressure*.
- ③ A Dissipation error should never be triggered more than once per cycle. To avoid raising multiple Dissipation error alarms, this state checks and sets the *Injector.Dissipation error* Boolean attribute value.
- ④ When the Start event is received from the Autocycle Session, it is necessary to verify that the target Machinery is not locked out before proceeding to inject.
- ⑤ Either a Lockout or a Stop event will put the Injector back to sleep.

Figure 6-9 shows the bottom half of the Injector state model.

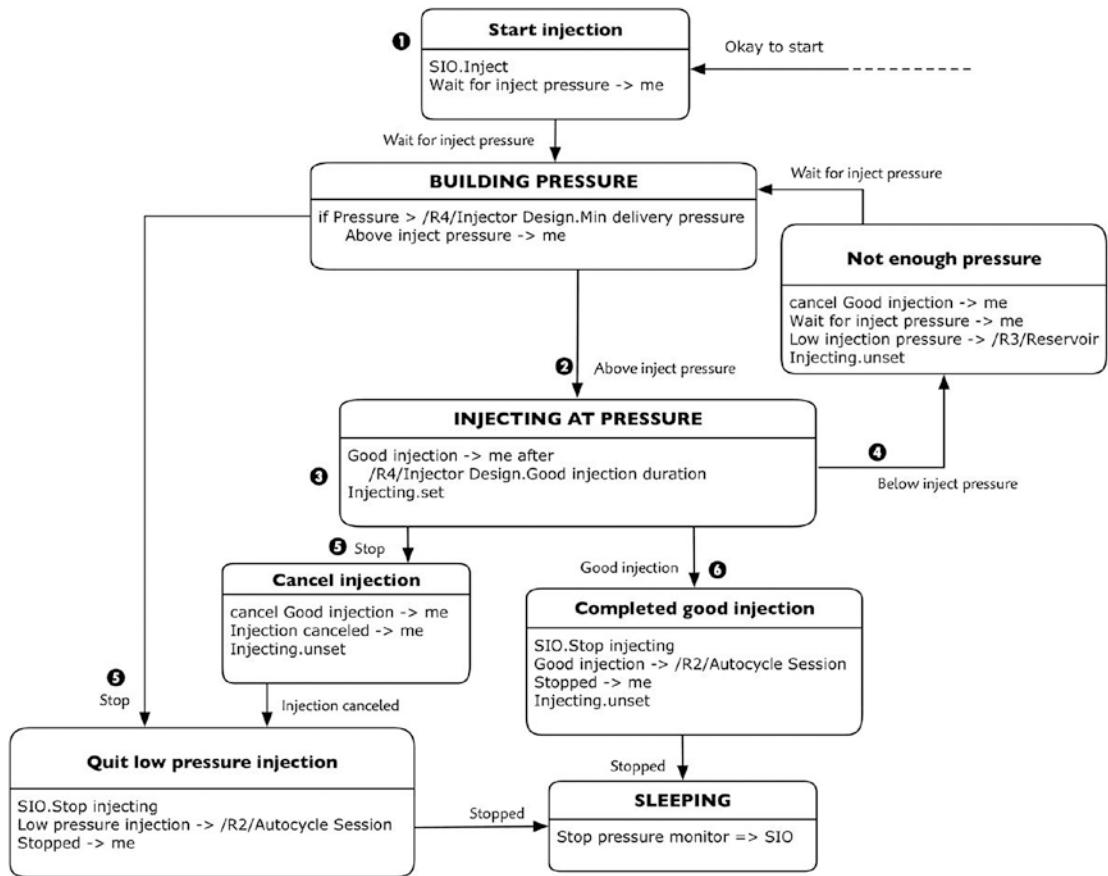


Figure 6-9. Injector state model excerpt: Active to Sleeping

- ❶ The Injector solenoid is energized by sending the Inject signal to the SIO domain. From there, the Injector waits until the *Injector Design.Min delivery pressure* is attained.
- ❷ This event can be triggered as a result of the check in the BUILDING PRESSURE state or supplied by SIO when the condition is detected.
- ❸ When the pressure is high enough, lubrication is occurring. To get a good injection, this pressure must be maintained for the *Injector Design.Good injection duration*. This delayed event will be sent if all goes well while the Injector is in this state.
- ❹ Otherwise, SIO will send the *Below inject pressure* event, which leads to the *Not enough pressure* state, where the delayed event is canceled and the Reservoir is notified. From there, the Injector continues to wait for the pressure to build in the next state.

⑤ If a long enough continuous injection is not achieved soon enough, the Autocycle Session will trigger the Stop event, which moves the Injector to the *Quit low pressure injection* state, possibly via the *Cancel injection* state, which cancels the *Good injection* delayed event.

⑥ But if all goes well, the *Good injection* delayed event occurs, the solenoid is de-energized via SIO, and the Injector goes back to sleep.

Autocycle Session State Model

The Autocycle Session state model is a bit large to display here, so we walk through a summary without the action language visible. Figure 6-10 highlights the primary part of the life cycle. All of the grayed-out areas pertain to suspend/resume and deactivation behavior.

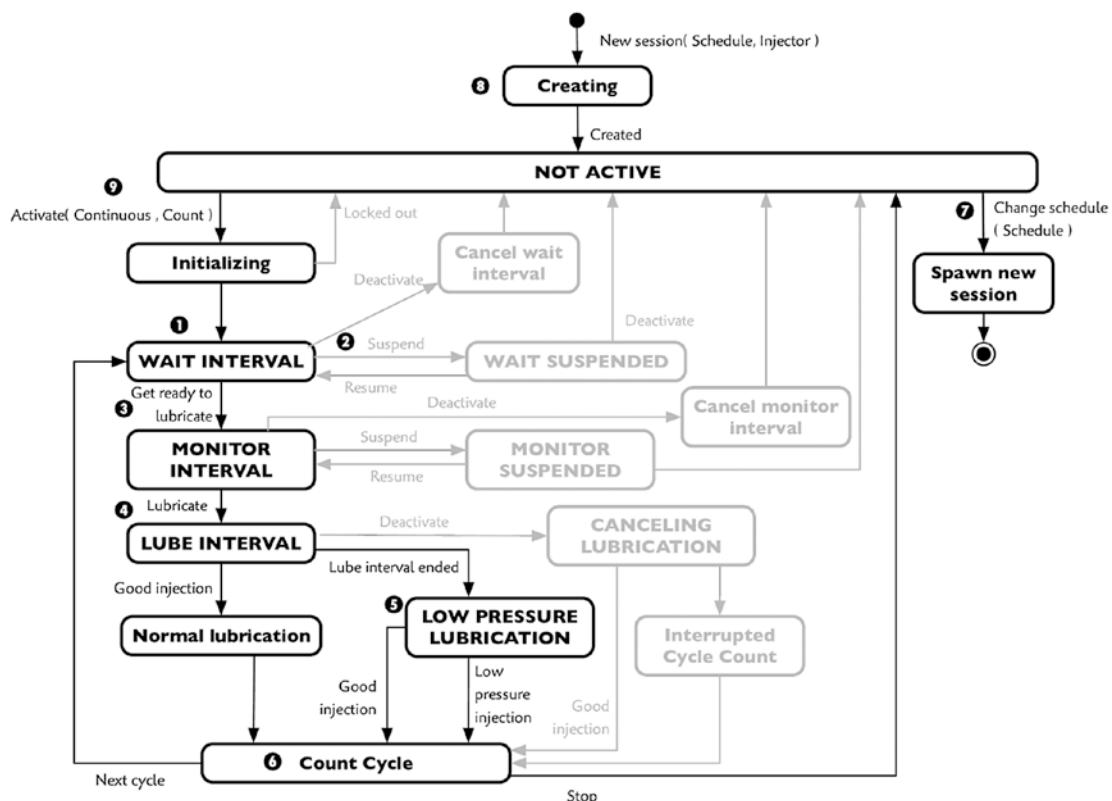


Figure 6-10. Autocycle Session state model overview

- ➊ Let's start with an active Autocycle Session in the WAIT INTERVAL state. Here, we are waiting for the delay between cycles to expire. According to R2 on the class model, there will be one such session instance per Injector at all times.
- ➋ It is possible during this state that the session may be suspended. If this happens, we will go sit in the WAIT SUSPENDED state until resumed or deactivated. We must cancel the pending *Get ready to lubricate* delayed event, but not before saving the time remaining. Upon resuming, we will reschedule the *Get ready to lubricate* event for whatever time was remaining when we were suspended.
- ➌ The scheduled event occurs, and we advance to the MONITOR INTERVAL state. This is where we send the Wakeup event to our Injector so it begins monitoring pressure. We now schedule the delayed Lubricate event for the interval defined in our Lubrication Schedule and wait. If suspension occurs in this state, the logic is similar to the previous, except that the intermediate time waited will not be saved.
- ➍ The Lubricate event occurs, and we advance to the LUBE INTERVAL state. We send the Start event to our Injector and wait for one of two events. Either our Injector succeeds and it sends us the *Good injection* event, or too much time elapses and our scheduled *Lube interval ended* event occurs. The lubrication interval is determined by the value of *Injector Design. Delivery window*. In the first case, we can cancel the pending *Lube interval ended* event.
- ➎ In the time-out case, we advance to the LOW PRESSURE LUBRICATION state, where we tell the Injector to stop and increment the *Failed cycles* count. The Injector will respond with either a *Good injection* or a *Low pressure injection* event. The first one can happen if our Lube Interval times out at roughly the same instant that the Injector succeeds. Either way, it is critical that we wait here for confirmation that the Injector has stopped before proceeding so that we don't lose synchronization with the Injector states.
- ➏ Either way, we advance to the *Count cycle* state, where the cycle is counted, and we check to see whether too many failed (low-lube) cycles have occurred. If so, we send the *Too many low lube cycles* event to our Reservoir so it can update its status. If we are operating in a continuous mode (nonstop repeating cycles), we go on to the next cycle. Otherwise, we see whether all requested cycles have been completed. If not, we also go on to the next cycle. Otherwise, with all requested cycles completed, we proceed to the NOT ACTIVE state.
- ➐ Now let's say that the user decides to run a different Lubrication Schedule on this Injector. We will require that the session is in the NOT ACTIVE state for this request to be processed. (It's ignored in all other states.) A Change Schedule event will be received with the name of the new Lubrication Schedule. This puts us in a deletion state, where we fire off the *New session* creation event just before the present instance disappears. Entry into the NOT ACTIVE state will always start by comparing the currently controlling Lubrication Schedule with the default on R1. If they don't match, it means that a temporary schedule has been running. This will result in a *New session* creation back to the designated default Lubrication Schedule.
- ➑ A new instance of Autocycle Session is created with the input Schedule and Injector instances linked together and placed in the NOT ACTIVE state.
- ➒ The user issues an Activate event along with the desired mode (continuous or not) and a desired cycle count, which should be greater than zero if the continuous mode was not requested. We'll verify that we aren't locked out by our Machinery before proceeding. If not locked out, we'll set the appropriate attributes and jump into the WAIT INTERVAL state. Note that a lockout that occurs in any other state will be detected by the Machinery that signals the Deactivate event to us.

Reservoir State Model

The Reservoir keeps track of its fill states to provide useful alarms to maintenance. During an ordinary fluid cycle, we start out with a normal level of fluid and then, driven by Injector events, descend through the states of LOW, VERY LOW, and EMPTY. Figure 6-11 shows the complete state model for the Reservoir class.

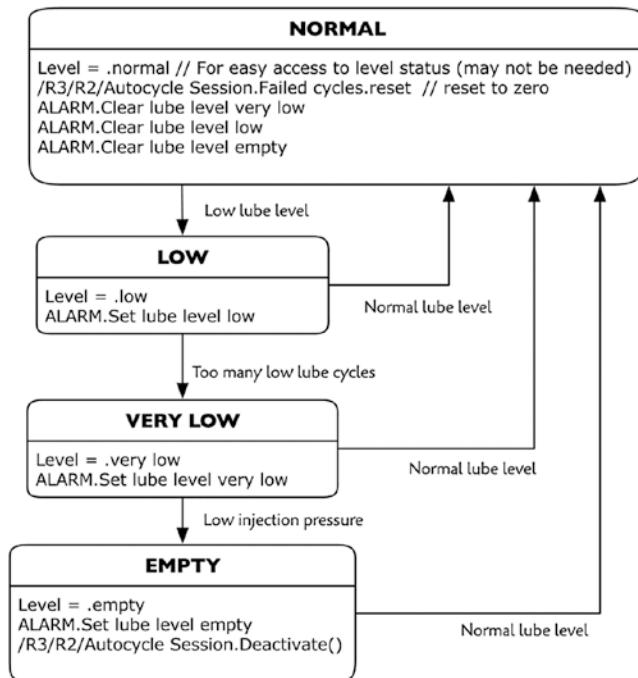


Figure 6-11. Reservoir state model diagram

Class Collaboration

The class collaboration diagram provides a nice overview of asynchronous (signal/event) and synchronous (methods invocation) interactions among state models within a domain. Class collaboration is *not* considered a separate facet of a domain model, because it can be derived from the contents of the three facets. But it is quite useful for devising a clean pattern of control within a domain (see Figure 14.8 on p. 245 of *Executable UML: A Foundation for Model-Driven Architecture* for a more complex example of control collaboration). Figure 6-12 shows how the classes of the Lubrication domain interact.

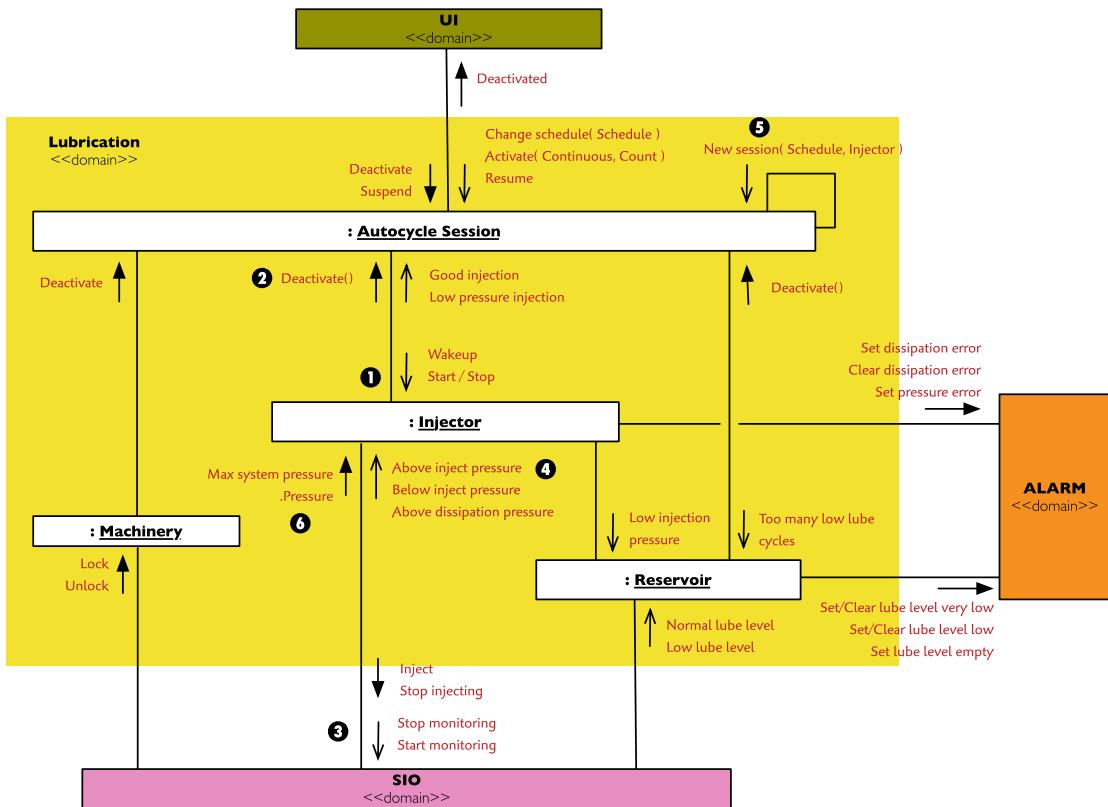


Figure 6-12. Class collaboration within the Lubrication domain

- ➊ These are events signaled in the Autocycle Session state model and detected in the Injector state model.
- ➋ This is a method defined on the Autocycle Session class. See Figure 6-13 for the method activities.
- ➌ These are signals issued by Injector actions that are translated to corresponding SIO elements. The collaboration diagram makes no assumptions about the internal elements or interfaces of external domains beyond basic services provided. All we can tell from this diagram is that the events must be mapped to something in the SIO domain.
- ➍ These events occur in the Injector state model and must be triggered by SIO. The specific mechanisms within SIO that accomplish this are unknown in the Lubrication domain.
- ➎ Events from an instance to itself are not shown on the collaboration diagram. In this case, however, the event is sent from one instance of Autocycle Session to a different, newly created instance of the same class.
- ➏ A processed signal value in the SIO domain is mapped to the Pressure attribute across the domain boundary. This attribute is read-only within the Lubrication domain.

Class Method and Other Activities

Activities in a domain are not limited to those within states. A *class method* is an instance-based activity defined on a class. A *domain operation* is an activity defined as part of a domain interface. An external domain can invoke another domain's operation to invoke a class method, for example, without interacting with any state machine.

Figure 6-13 shows the methods defined on each of the Lubrication domain classes.

Class method activities

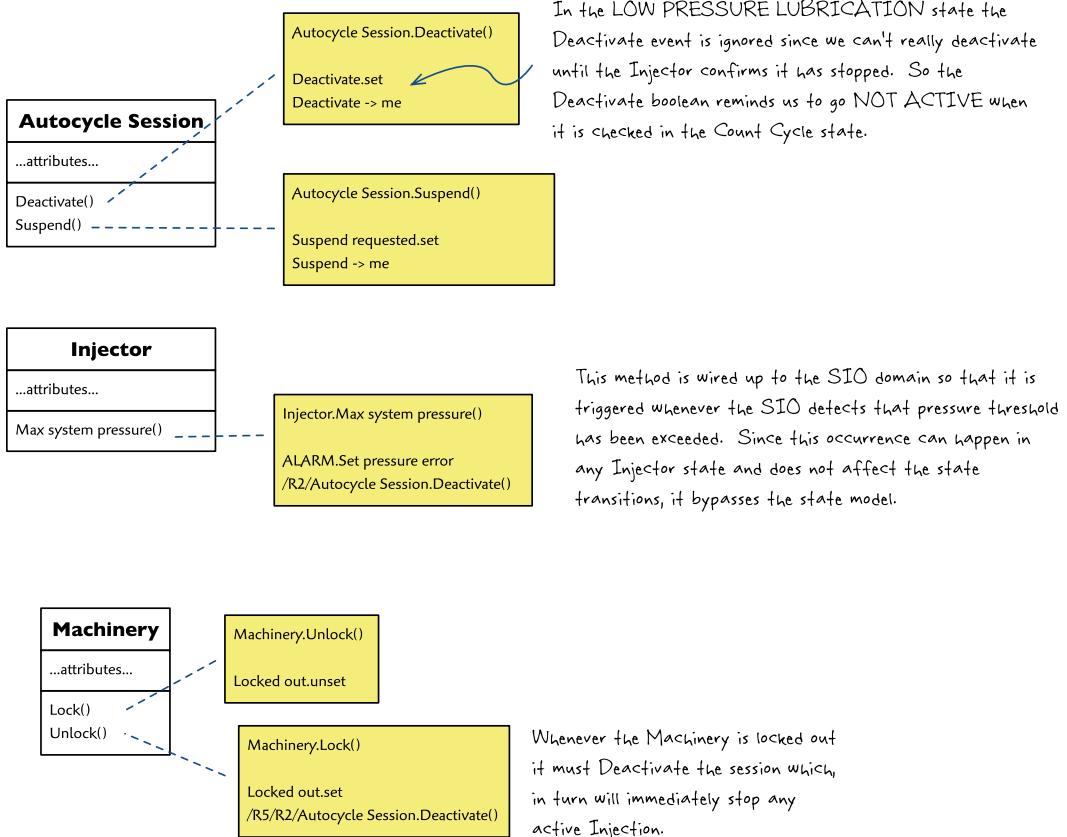


Figure 6-13. Class method activities

State Tables

A table for each state model must be completed before we are ready for pycca. The tables are filled out during the modeling process, and they must be complete before we begin specifying the translation. Some drawing tools will do this work for you.

As the Autocycle Session and Injector state tables are rather large, we examine selected excerpts in this chapter. You can download the tables as fully commented spreadsheets from the book's website if you wish to study them further.

Figure 6-14 is the first excerpt from the Injector state table.

Excerpt from Injector State Table

Wait states	Nonlocal events						Delayed 
	Wakeup	Above inject pressure	Below inject pressure	Start	Stop	Above dissipation pressure	
BUILDING PRESSURE	CH	INJECTING AT PRESSURE	IG	CH	Quit low pressure injection	IG	CH
INJECTING AT PRESSURE	CH	IG	Not enough pressure	CH	Cancel injection	IG	Completed good injection
SLEEPING	Initiate monitoring	IG	IG	CH	IG	IG	CH
MONITORING	CH	IG	IG	Clear error and check interlock	SLEEPING	Set dissipation error	CH

Figure 6-14. Injector wait states

You may have noticed that some state names in the diagrams are named with all uppercase letters, whereas others are mixed case. This is an informal style for distinguishing wait and transitory states. A *wait state* is a state with an activity that does not send an event to itself to force a transition to another state. Instead, an instance that has completed its activity will either process a pending *nonlocal event* (an event not from this instance or delayed) or just wait for such an event to occur. Less formally, you can think of wait states as those in which an instance is waiting on a process to complete in the physical world, in some other domain or in some other instance, or for a delayed event to happen.

This means that all *local events* (nondelayed, self-to-self events) are marked Can't Happen (CH) for wait states. That's because a local event is generated in another state and processed by that same state, so it can't happen here. So we can exclude all the local events, obtaining a smaller state table, and concentrate on nonlocal events.

Note the inclusion of the single delayed *Good injection* event. We can mark it as Can't Happen in all states other than INJECTING AT PRESSURE if we can demonstrate that the event will either expire or be canceled before entering any of those other states. The format of the state table forces consideration of this possibility that is so easily overlooked otherwise!

Figure 6-15 shows the second state table excerpt for the Injector.

Excerpt from Injector State Table

Transitory states	Local events						
	Wait for inject pressure	Monitor	Stopped	Locked out	Ok to start	Injection canceled	Keep monitoring
Start injection	BUILDING PRESSURE	CH	CH	CH	CH	CH	CH
Not enough pressure	BUILDING PRESSURE	CH	CH	CH	CH	CH	CH
Completed good injection	CH	CH	SLEEPING	CH	CH	CH	CH
Quit low pressure injection	CH	CH	SLEEPING	CH	CH	CH	CH
Initiate monitoring	CH	MONITORING	CH	CH	CH	CH	CH
Set dissipation error	CH	CH	CH	CH	CH	CH	MONITORING
Clear error and check interlock	CH	CH	CH	SLEEPING	Start injection	CH	CH
Cancel injection	CH	CH	CH	CH	CH	Quit low pressure injection	CH

Figure 6-15. Injector transitory states

All states in the preceding table are *transitory*, which means that each state is exited on a local (nondelayed, self-to-self) event. A transitory state makes it possible to execute an activity on one transition path leading to a wait state, thus allowing the possibility of other transition paths to that same state where different or possibly no activities are performed. Often transitory states are used to perform if-then logic leading to alternate exit transitions.

Note that each transitory state responds only to an event it sends to itself, for the following reasons:

- The event will always be signaled by the state activity.
- Events that an instance signals to itself are dispatched before any other event.

This means that all other events may be marked CH, as shown in the preceding table excerpt. Because CH will be marked for all the nonlocal events shown in the first excerpt, there isn't much insight to be gained from viewing that part of the table.

Excerpt from Autocycle Session State Table

Wait states	Nonlocal events						Delayed			
	Change schedule	Suspend	Resume	Activate	Deactivate	Low pressure injection	Good Injection	Lubricate	Lube interval ended	Get ready to lubricate
NOT ACTIVE	Spawning new session	IG	IG	Initialize	IG	CH	CH	CH	CH	CH
WAIT INTERVAL	IG	WAIT SUSPENDED	IG	IG	Cancel main interval	CH	CH	CH	CH	MONITOR INTERVAL
WAIT SUSPENDED	IG	IG	WAIT INTERVAL	IG	NOT ACTIVE	CH	CH	CH	CH	CH
MONITOR INTERVAL	IG	MONITOR SUSPENDED	IG	IG	Cancel monitor interval	CH	CH	LUBE INTERVAL	CH	CH
MONITOR SUSPENDED	IG	IG	MONITOR INTERVAL	IG	NOT ACTIVE	CH	CH	CH	CH	CH
LUBE INTERVAL	IG	IG	IG	IG	CANCELING LUBRICATION	CH	Normal lubrication	CH	LOW PRESSURE LUBRICATION	CH
LOW PRESSURE LUBRICATION	IG	IG	IG	IG	IG	Count cycle	Count cycle	CH	CH	CH
CANCELING LUBRICATION	IG	IG	IG	IG	IG	Interrupted cycle count	Count cycle	CH	CH	CH

Figure 6-16. Autocycle Session wait states

Figure 6-16 shows the wait state excerpt of the Autocycle Session class.

The three delayed events are highlighted in the rightmost columns. Care has been taken to ensure that each delayed event is managed so that it does not occur in any state where it is not helpful.

Excerpt from Autocycle Session State Table

Transitory states	Local events								
	Created	Activated	Locked out	Next cycle	Stop	Count as normal	Monitor interval canceled	Wait interval canceled	Cycle interrupted
Creating	NOT ACTIVE	CH	CH	CH	CH	CH	CH	CH	CH
Initialize	CH	WAIT INTERVAL	NOT ACTIVE	CH	CH	CH	CH	CH	CH
Normal lubrication	CH	CH	CH	CH	CH	Count cycle	CH	CH	CH
Count cycle	CH	CH	CH	WAIT INTERVAL	NOT ACTIVE	CH	CH	CH	CH
Cancel monitor interval	CH	CH	CH	CH	CH	CH	NOT ACTIVE	CH	CH
Cancel wait interval	CH	CH	CH	CH	CH	CH	CH	NOT ACTIVE	CH
Interrupted cycle count	CH	CH	CH	CH	CH	CH	CH	CH	Count cycle

Figure 6-17. Autocycle Session transitory states

Finally, Figure 6-17 shows the transitory states of the Autocycle Session class.

Reservoir State Table

Wait states	Nonlocal events			
	Low lube level	Normal lube level	Too many low lube cycles	Low injection pressure
NORMAL	LOW	IG	CH	IG
LOW	IG	NORMAL	VERY LOW	IG
VERY LOW	IG	NORMAL	CH	EMPTY
EMPTY	IG	NORMAL	CH	IG

Figure 6-18. Reservoir state table

The Reservoir state table is small enough to show entirely in Figure 6-18.

Translating the Lubrication Domain

We undertake the translation of the Lubrication domain in the same way as we translated the ATC domain—namely, we transcribe the three facets into the pycca DSL. We apply the same decision process we used for the ATC domain to determine the attributes and how references between pycca classes will work. The state model is transcribed into transition and state statements, and the action language for each state is translated into C with the help of the pycca macros to handle model-level requests.

Rather than repeat sequences that have already been seen, we will focus on pycca constructs that have not already been presented. As always, the complete model and its translation are available as part of the online materials for the book. In this section, we start with translating associations that have an association class, followed by the translation of creation events and other operations as they arise in the Lubrication domain.

Translating Association Classes

In Chapter 3, we discussed implementing associations by decomposing the link storage based on each direction of traversal of the association. This was illustrated in the case of a simple association in Figure 3-5.

Referring to Figure 6-6, R2 is formalized by the Autocycle Session class. Not only does this class have referential attributes to formalize R2, but it has descriptive attributes and a state model. Figure 6-19 shows how R2 is decomposed into two sides with the Autocycle Session class serving as an intermediary.

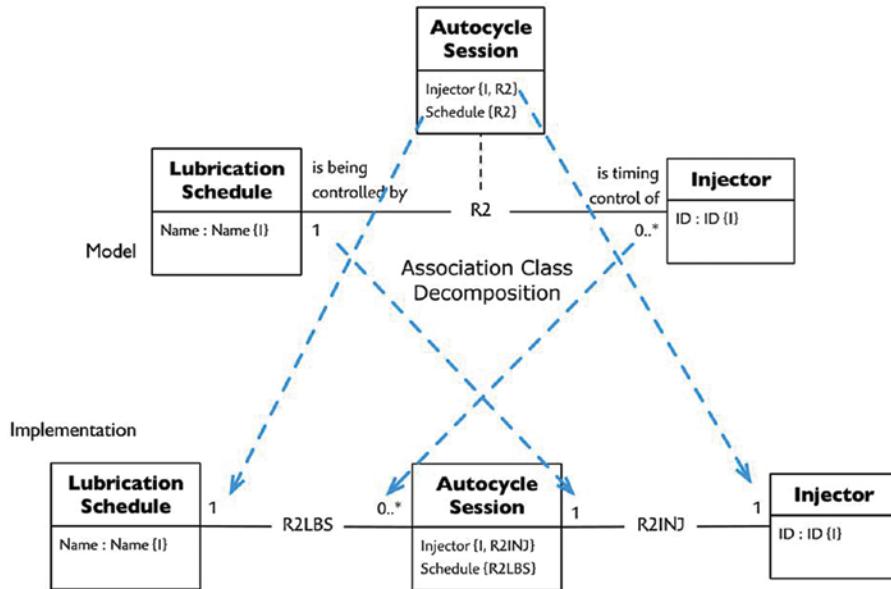


Figure 6-19. Decomposition with an association class

The R2 association has several interesting properties:

- Each Autocycle Session instance corresponds to an instance of the R2 association itself.
- Autocycle Session has two singular, unconditional references, one to Injector and one to Lubrication Schedule.
- The conditionality and multiplicity of the two sides are inverted in the decomposition. So Lubrication Schedule is related to Autocycle Session as “0..*” and Injector is related to Autocycle Session as 1. This is a consequence of the singular, unconditional reference from Autocycle Session to each participant of R2.

When translating the Lubrication domain, we use the decomposed form of R2 to determine how to set up the references. For the Autocycle Session class, this appears as follows:

```

class Autocycle_Session
    reference R2_INJ -> Injector
    reference R2_LBS -> Lubrication_Schedule

    # Other attributes and state model for Autocycle Session
End
  
```

Navigating the R2 association is also a two-step action. So, to navigate from an instance of Injector to an instance of Lubrication Schedule requires the following:

```

// Assuming inj holds a reference to an injector instance.

ClassRefVar(Lubrication_Schedule, sched) = inj->R2->R2_LBS ;
  
```

Navigating Associative Relationships

In the previous section, we showed a simple example of navigating an associative relationship. In that case, the multiplicity from the participating class to the association class was singular. In this section, we show a many-to-many associative relationship using a linked list to implement the linkage with the association class. Recall that in pycca, we treat associative relationships as being composed of two separate paths. Consequently, the translation of navigation for an associative relationship requires two parts. Consider the association shown in Figure 6-20.

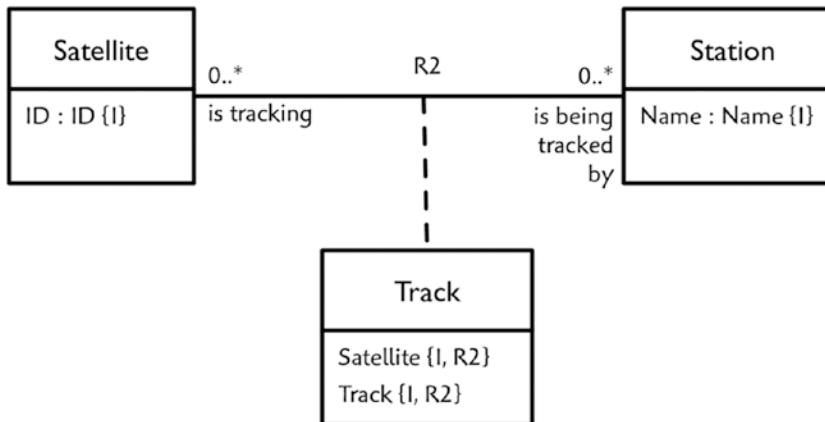


Figure 6-20. Navigating an associative relationship

The decomposition of the associative relationship results in singular references to each participant in the association class:

```

class Track
    reference R2_Satellite -> Satellite
    reference R2_Station -> Station
end
  
```

The participants have their own links to the association class. Here we have chosen linked lists in anticipation that the association is dynamic:

```

class Satellite
    attribute (char const *ID)
    reference R2_Satellite ->>l Track
end

class Station
    attribute (char const *Name)
    reference R2_Station ->>l Track
end
  
```

Note that we have appended the class name to the reference names for each participant. We need to have different reference names because pycca will use those names directly in the generated structure, and we don't want to have a naming conflict. We have chosen to append the class name because it is a convenient convention that lets us easily keep track of where the references originate or terminate.

Pycca will generate the following structures for the participants:

```
struct Satellite {
    struct mechinstance common_ ; // must be first !
    char const *ID ;
    rlink_t R2_Satellite ;
}

struct Station {
    struct mechinstance common_ ; // must be first !
    char const *Name ;
    rlink_t R2_Station ;
}
```

The preceding `rlink_t R2_XX` members serve as the terminus of a linked list and have the following structure:

```
typedef struct rlink {
    struct rlink *next ;
    struct rlink *prev ;
} rlink_t ;
```

The `Track` structure will have linked list pointers inserted into its structure for the linked list references defined by both the `Satellite` and `Station` classes:

```
struct Track {
    struct mechinstance common_ ; // must be first !
    struct Satellite *R2_Satellite ;
    struct Station *R2_Station ;
    rlink_t R2_Satellite_links ; // ①
    rlink_t R2_Station_links ; // ②
} ;
```

① `Track` instances are linked together from the `Satellite` side, and this list contains those instances of `Track` related to an instance of `Satellite`.

② Likewise for the `Station` instances. Note that the two linked lists plus the singular reference to the other participant gives us the many-to-many representation required by the association.

To navigate the association from `Satellite` to `Station`, we must first navigate to `Track` instances and from there follow the singular reference to a `Station` instance. Because the `R2` association is many-to-many, we can have many `Track` instances related to a given `Station` instance, and so we need an iteration loop as we navigate from a `Satellite` instance to the related `Track` instances.

For a given instance of `Satellite`, to print the names of all the `Stations`, we would use the following code:

```
// ... assuming self is a reference to a Satellite instance ...
rlink_t *tracklink ; // ①
PYCCA_forAllLinkedInst(self, R2_Satellite, tracklink) { // ②
    ClassRefVar(Track, tracki) = PYCCA_linkToInstRef(tracklink, Track, R2_Satellite) ; // ③
    ClassRefVar(Station, stationi) = tracki->R2_Station ; // ④
    printf("Satellite, %s, is being tracked by, %s\n",
        self->ID, stationi->Name) ;
}
```

① When using linked lists, we need a linked list pointer variable to hold our place in the iteration.

② Pycca supplies a macro to set up the iteration loop.

③ The pointers in the linked list from the Satellite instance point to the R2_Satellite_links member of the Track instance and *not* to the beginning of the Track instance. What we really want is the pointer to the beginning of the instance element, because that is what an instance reference truly is. Pycca supplies a macro to perform the address arithmetic to get us from the R2_Satellite_links member back to the Track instance reference. This is a common idiom when an object can be linked onto multiple linked lists.

④ Finally, we can follow the singular link in the Station direction.

Creation Events

The two techniques for creating a class instance are synchronous creation and asynchronous creation. In *synchronous creation*, an activity makes a direct request to the MX runtime to create an instance, and that request is fulfilled immediately. The pycca macro that supports this is PYCCA_createInstance(). Class instances created this way are placed in their default initial state (if they have a state model), but the activity of the state is *not* executed. For example, all the instances that make up the initial instance population are considered to have been created in this way (there is no runtime cost for creating the initial instance population, because pycca arranges for the initial values to be placed in memory at compile time).

For the *asynchronous* case, instance creation is triggered by signaling a special *creation event*. The activity that signals this event continues to execute to completion (as per our usual execution rules). At some time in the future, when the event is dispatched, a new class instance is created, and the event is received by the newly created instance. From the newly created instance's perspective, this is just a normal event. Consequently, the newly created instance makes a state transition, and the activity associated with the new state is executed (again per our usual execution rules of state machines).

From the modeling point of view, both techniques for instance creation have their uses. For the Lubrication domain, asynchronous creation is used to switch lubrication sessions. If you look back at Figure 6-10, at the top of the diagram the *New session* event causes a transition from a solid circle into the Creating state. The solid circle represents an *initial pseudo-state*, and the *New session* event is a creation event. The MX runtime will ensure that when the *New session* event is dispatched, an instance of Autocycle Session is created, it is placed in the initial pseudo-state, and the transition to the Creating state is taken, causing the activity to be executed.

Pycca has direct support for specifying a creation event. Next we show the required pycca statements. We have omitted much of the surrounding domain definition as well as the details of the Autocycle Session class definition to focus on the creation event:

```
# ... other parts of the Lube domain

class Autocycle_Session
    # ... attribute definitions, reference definitions, etc.

    machine
        default transition CH          # ①
        initial state Creating         # ②

        transition . - New_session -> Creating      # ③

        # ... remaining parts of the state model definition
    end
end

# ... other parts of the Lube domain
```

① As usual, we specify a default transition of CH and the ignored transitions explicitly.

② Just because a state model has creation events does not mean it cannot be created synchronously. We have to specify the default initial state when instances of Autocycle Session are synchronously created.

③ The . (period) character designates the initial pseudo-state. Here we say that the New_session event transitions from the initial pseudo-state to the Creating state, and that makes it a creation event.

It is possible for a state model to have multiple creation events, although that is not a common situation. It is also possible to signal New_session as an ordinary transitioning event, although that does not happen in the Autocycle Session state model. It is also possible to signal an ordinary transitioning event as a creation event. That results in a CH transition, and the pycca runtime will respond with a fatal error.

Signaling a creation event translates to a slightly different code sequence. In the Autocycle Session state model, it is the *Spawn new session* state that signals the *New session* event:

```
New session( in.Schedule,
    Injector: /R2/Injector.ID ) -> Autocycle_Session
// This event will create, and be delivered to,
// a new instance of Autocycle Session to replace
// this one being deleted.
```

The action language uses syntax to imply that the *New session* event is signaled directly to the Autocycle Session class. This is a convenient way to distinguish a creation event, as signals are normally directed at class *instances*. The translation of this action language follows:

```
# ... other parts of the Lube domain

class Autocycle_Session
    # ... Autocycle Session attribute definitions, reference definitions, etc.

    machine
        # ... other parts of the Autocycle Session state model definition
```

```

state Spawn_new_session(
    char const *schedule)
{
    MechEcb new_session =
        PYCCA_newCreationEventForThisClass(New_session, self) ;           // ①
    PYCCA_eventParam(new_session, Autocycle_Session, New_session, schedule) =
        rcvd_evt->schedule ;                                              // ②
    PYCCA_eventParam(new_session, Autocycle_Session, New_session, injector) =
        PYCCA_idOfRef(Injector, self->R2_INJ) ;
    PYCCA_postEvent(new_session) ;                                         // ③

    self->R2_INJ->R2 = NULL ;                                           // ④
}
# ... remaining parts of the state model definition
end
# ... other parts of the Lube domain

```

- ① This allocates an ECB and marks the event as a creation event.
- ② Because the event carries parameters, we need to fill in the values. When there are no parameters, we use the pycca macro PYCCA_generateCreation() to handle the entire operation.
- ③ Posting the event finishes the event-signaling process.
- ④ We need to clean up a reference from the Injector instance because this Autocycle_Session instance is about to be deleted. The next section discusses this automatic instance deletion.

Asynchronous Instance Deletion

Just as there are two ways to create class instances, there are also two ways to delete them. An activity may synchronously delete an instance by using the PYCCA_destroyInstance() macro. When the underlying runtime function returns, the instance no longer exists.

Referring to Figure 6-10 again, we notice that there is a transition from *Spawn new session* to a small circle and that the transition has no event label. The small circle represents a *final pseudo-state*. The transition to a final pseudo-state designates the *Spawn new session* as a final state. When an instance enters a final state, it is deleted after its activity is run. The MX runtime handles deleting the instance automatically. Pycca supports declaring any state to be a final state. For the Autocycle Session class, it appears as follows:

```

# ... other parts of the Lube domain

class Autocycle_Session
    # ... attribute definitions, reference definitions, etc.

    machine
        # ... other parts of the state model definition

```

```

final state Spawn_new_session

# ... remaining parts of the state model definition
end
end

# ... other parts of the Lube domain

```

Because the instance is deleted when a final state activity is run, outgoing transitions from a final state are not allowed, and pycca will flag an error if one is defined.

Finally, we can return to the last line of code in the `Spawn_new_session` activity:

```
self->R2_INJ->R2 = NULL ;
```

It is necessary to `NULL` out the reference from the `Injector` instance to the `Autocycle_Session` because `Spawn_new_session` is a final state and about to be deleted. We do not want the `Injector` instance referring to something that is about to go away. We don't have to deal with the references made by the `Autocycle_Session` instance because it will be deleted and therefore can't refer to anything.

When the creation event is dispatched and a new `Autocycle_Session` instance is created, the activity of the `Creating` state will set the new references properly. In a time window between the end of the `Spawn_new_session` activity and the dispatch of the `New_session` creation event, the conditionality of the references implied by the `R2` association is violated. That is, there is a time when the `Injector` instance is not controlled by any Lubrication Schedule. This does not violate our execution rules, because we have signaled the event that, when it is dispatched, will bring all the references back to a consistent state. You can think of this sequence as being part of a larger *transaction* on the data model, and any integrity rule checks are deferred until the transaction is completed, which occurs when the creation event is dispatched. The ST/MX runtime code does not actually perform any of these integrity checks, but other, more capable model execution domains might. For example, a model execution domain that uses a relational database management system to manage the domain data can easily provide transaction-based integrity checking.

Operations

Most of the algorithmic processing in a domain happens in the state activities. However, there are other means to factor processing into invocable units. For example, common code executed in multiple state activities should be factored into a single place. In this section, we show how code can be grouped into various types of *operations*.

Class Methods

A domain may define class methods to encapsulate processing performed on a particular class instance. For the Lubrication domain, several class methods were shown in Figure 6-13.

Pycca supports defining class methods as part of the specification of a class. Here we show the translation of the `Max_system_pressure` method from the `Injector` class:

```
Injector.Max_system_pressure()
```

```
ALARM. Set pressure error()
/R2/Autocycle Session.Deactivate()
```

Reacting to the lubricant pressure exceeding its maximum requires raising an alarm and invoking a method on the related Autocycle Session instance. These two actions need to be performed together and so are part of a single method.

In pycca terms, a class method is known as an `instance operation` and is defined as shown here:

```
# ... other parts of the Lube domain

class Injector
    # ... Injector attribute definitions, reference definitions, etc.

    instance operation Max_system_pressure() {
        ExternalOp(ALARM_Set_pressure_error)(PYCCA_idOfSelf) ; // 1
        ClassRefVar(Autocycle_Session, acs) = self->R2 ;           // 2
        InstOp(Autocycle_Session, Deactivate)(acs) ;               // 3
    }

    # ... remaining parts of the Injector definition
end

# ... other parts of the Lube domain
```

1 The `ExternalOp()` macro hides the naming conventions used by pycca to resolve external operation names.

2 Follow the association reference to obtain the instance of the `Autocycle_Session` that must be deactivated.

3 Because this is C, we must pass the reference to the instance explicitly. A more object-oriented language would probably do this for us. The `InstOp()` macro hides the naming conventions used by pycca for instance operations. Pycca uses a number of naming conventions to ensure that function names are unique within the generated C file.

Domain Operations

The Lubrication domain depends on the Signal I/O domain to detect when the pressure on an injector exceeds its maximum. When that happens, the *Max system pressure* method needs to be invoked for the overpressured Injector instance. To accomplish this, we need a function that can be invoked on the Lubrication domain that will find the correct instance of Injector and invoke the *Max system pressure* method.

In pycca terms, an operation that forms part of the service interface for a domain is called a **domain operation**. Domain operations provide the visible services that may be invoked on a domain at runtime. A domain starts in a well-known state defined by the initial instance population and the initial states of the active class instances. How a domain evolves over time depends on the initial configuration as well as the invocations to the service interface provided by the domain operations. You may think of the domain operations as forming an API for the domain but, unlike a conventional programming interface, a domain can come with a significant initial configuration and is not necessarily completely configured at runtime.

The following shows the domain operation definition that SIO may use to declare that the maximum pressure on an Injector has been exceeded:

```
# ... other parts of the Lube domain

domain operation
Injector_max_pressure(
    InstId_t injId)
{
    PYCCA_checkId(Injector, injId) ; // 1
    ClassRefVar(Injector, inj) = PYCCA_refOfId(Injector, injId) ; // 2
    if (IsInstInUse(inj)) { // 3
        InstOp(Injector, Max_system_pressure)(inj) ;
    }
}

# ... other parts of the Lube domain
```

① Injectors are identified by a small integer number at the domain interface. Here we use a pycca macro to make sure we are not handed an out-of-bounds identifier.

② Convert the identifier to an actual pointer reference to the requested Injector.

③ We must make sure the instance is not an empty storage slot.

External Operations

As domain operations provide the service interface for a domain, an `external` operation declares a service dependency for a domain. Each external operation that appears in an activity has a corresponding definition of its invocation interface. The following is an example of two external operation definitions:

```
# ... other parts of the Lube domain

external operation
SIO_Inject(InstId_t injectorId)
{

external operation
SIO_Stop_injecting(InstId_t injectorId)
{
```

```
# ... other parts of the Lube domain
```

The Lubrication domain will invoke these operations at the appropriate time to start and stop lube injection. How starting and stopping injection happens in the real world has been delegated to another domain, and the Lubrication domain assumes it will happen.

Notice that no code is included in the definition. Pycca will not emit code for the external operations themselves; that is typically supplied by bridge code. However, pycca will accept code as part of the external operation definition, and certain companion tools to pycca can create stubs for the external operations by using the included code.

We will have more to say about domain operations and external operations when we get to Chapter 8. There we will use the domain and external operations to bridge the Lubrication and Signal I/O domains.

Class-Based Operations

Pycca supports one more type of grouping for processing, the `class` operation. A class operation is similar to an instance operation, except that no instance reference is passed to the function. It is conceptually similar to class-based operations in conventional object-oriented programming languages.

Class operations are *not* part of the domain model (and, more to the point, they are intentionally not supported in xUML). They are strictly an implementation artifact. In a domain model, all operations on classes are provided by constructs of the action language. For example, creating and destroying an instance of a class is provided for by the action language. It is also possible in the action language to find a subset of the class instances based on the value of an expression. All the operations on classes are provided generically by the action language constructs.

When those action language constructs are translated, however, we must devise an implementation to accomplish the action language intent. To illustrate this, we will consider the Creating state of the Autocycle Session. Recall that in Figure 6-10, the Creating state is the one entered upon receiving the *New session* creation event. Its activity is shown here:

```
// Link Schedule and Injector together to create this instance
Lubrication Schedule( Name: in.Schedule ) &R2 Injector( ID: in.Injector )
Created -> me
```

The first line of the activity says to find the instance of Lubrication Schedule whose name is passed as a parameter and relate it across R2 to the instance of Injector whose ID is also given as a parameter. To do this, we need to search the instances of Lubrication Schedule to find the one of interest.

When considering how to translate this, two broadly different approaches can be taken:

- We can write a generic runtime expression evaluator that would operate across the instances of an arbitrary class and evaluate an expression that contains variable terms bound to the class attributes.
- We can write a type-specific piece of code suitable to evaluate a particular expression across the instances of a particular class.

For the types of target systems we are considering, and because we are coding in a statically typed language, we always choose the second approach. To write a generic runtime expression evaluator is a large and complicated task, and the amount of code required has to be amortized across the entire application to make it worthwhile. Unlike a database management system that must support ad hoc queries on the data set that are not known in advance, the data queries for a domain are fixed by the activities in the model and known at translation time.

Usually, the number of distinct expressions that must be evaluated is small, and writing a type-specific piece of code for each one usually results in less code than a generic solution. Some object-oriented implementation languages support type-safe generic programming (for example, C++ using templates), and in those languages a simple search may require only “a one-liner.” For C, we will have to roll up our sleeves and code each expression evaluation on its own or, as we see with pycca, perform some preprocessor macro gyrations.

In pycca, implementing a search on class instances is most easily realized as a `class` operation. For the Lubrication Schedule class, the search for a matching schedule appears as follows:

```
# ... other parts of the Lube domain

class Lubrication_Schedule
    # ... Lubrication Schedule attribute definitions, reference definitions, etc.

    class operation findByName(char const *name) : (struct Lubrication_Schedule *) { // 1
        ThisClassRefVar(ls) ; // 2
        PYCCA_selectOneStaticInstOfThisClassWhere(ls, strcmp(ls->Name, name) == 0) // 3
        return ls == ThisClassEndStorage ? NULL : ls ; // 4
    }

    # ... remaining parts of the Lubrication Schedule definition
end

# ... other parts of the Lube domain
```

1 The return type is indicated after a colon (:) character. In this case, we return a pointer to a `Lubrication_Schedule` instance.

2 We need a variable to act as an iterator for the search and to hold the result.

3 The operation is common enough that pycca provides a macro that expands to perform a linear search of the class instances. The second argument to the macro is a Boolean expression that evaluates to true if the instance matches.

4 Because the Name attribute is an identifier, we know that there can be at most one instance of Lubrication Schedule that matches the input name value. If the iterator reached the end of the storage for class instances, we use NULL to indicate we did not find a match.

The following is the translation of the Creating state activity with the invocation of the `findByName()` operation:

```
# ... other parts of the Lube domain

class Autocycle_Session
    # ... attribute definitions, reference definitions, etc.

    machine
        # ... other parts of the state model definition
        state Creating(
            char const *schedule,
            unsigned injector)
        {
            ClassRefVar(Lubrication_Schedule, ls) =
                ClassOp(Lubrication_Schedule, findByName)(rcvd_evt->schedule) ; // 1
            assert(ls != NULL) ; // 2
            ClassRefVar(Injector, inj) = PYCCA_refOfId(Injector, rcvd_evt->injector) ;
            self->R2_LBS = ls ;
            self->R2_INJ = inj ;
            inj->R2 = self ;
```

```

        PYCCA_generateToSelf(Created) ;
    }

    # ... remaining parts of the state model definition
end
end

# ... other parts of the Lube domain

```

① Again, pycca provides a macro to hide the naming conventions used.

② We insist that a Lubrication_Schedule instance is found.

The search for a matching schedule implemented by the `findByName()` class operation is a simple linear search across the array of Lubrication Schedule instances. That is what the `PYCCA_selectOneStaticInstOfThisClassWhere()` macro provides. For a small number of instances, this is probably the best approach. But what happens if the number of instances of Lubrication Schedule is substantially larger, say 1,000? If the number of instances is large and the frequency of invoking the search is high, then the linear search technique could become a performance bottleneck.

We are obliged to state our strongly held belief that performance optimizations should *always* be based on performance *measurements* and not *thought experiments* (a.k.a. guessing). That said, we can code the `findByName()` operation to use a search algorithm that displays better performance characteristics when faced with larger instance populations. For example, we can take advantage of pycca's placement of initial instances into their storage array in the order they are defined. If we define the instances of `Lubrication_Schedule` in alphabetical order by the `Name` attribute, we can use the standard C library function, `bsearch()`, to find a matching instance using a binary search. Because the search is for equality of an identifier, another alternative would be to keep a parallel hash map data structure and use a hash function to compute an index into an array of pointers to the class instances corresponding to where the `Name` attribute hashes. If the number of instances of Lubrication Schedule is static (as we have translated it here), it is possible to compute a *perfect hash function* for the schedule names. Searching is a well-researched aspect of computer science, and we want to select the appropriate algorithms based on the computational demands of the particular situation; that selection can have a large impact on the quality of the implementation.

It is important to recognize that regardless of how the search is performed, the model logic of the activity is preserved, and the model is nondestructively transformed into the implementation. The action language states that the instance must be found. How that is accomplished is strictly the purview of the implementation, and choosing one search technique over another has zero impact on the logic specified by the model.

Summary

An Automatic Lubrication System (ALS) was introduced, featuring multiple domains. A domain is an executable package of content with a set of semantics at the same level of abstraction. Organization by domain is orthogonal to the practice of modeling. A domain may be either modeled or hand coded. Domains are organized by a principle of delegation. Whatever content is excluded from consideration in one domain may be delegated to another. Ultimately, all concepts essential to a software system must be present in a domain. To make implementation possible, the chain of domain delegation must always culminate in one or more existing implemented domains.

The ALS consists of the Lubrication, Signal I/O (SIO), User Interface (UI), and Alarms domain. These run on top of the model execution domain that is implemented in C. This chapter described the topmost domain, Lubrication.

The Lubrication domain manages a set of injectors according to a programmable cyclic schedule. System pressure and safety lockouts are monitored as part of the lubrication process. Interaction among the state models within the Lubrication domain and external domains are coordinated with a class collaboration diagram.

Various state actions rely on services supplied by the SIO and Alarms domains. Additionally, the UI interacts by interacting with the Autocycle Session class.

Translation examples of yet unused constructs in pycca were shown. This included signaling creation events and factoring code into instance, domain, external, and class operations.

CHAPTER 7



Sensor and Actuator Service Domain

In the previous chapter, we presented the Lubrication domain, which is part of an Automatic Lubrication System (ALS). The Lubrication domain delegates to a Signal I/O (SIO) domain controlling and obtaining the necessary data about the lubricated machinery. In this chapter, we describe the SIO domain and show how sensing and control over the external world is accomplished.

From the point of view of the whole ALS, we are producing side effects in the outside world. Indeed, for many systems, it is just these side effects that give the system its utility and purpose. Engineers have devised many ways for happenings outside a computer system to be sensed and controlled by a computer system. Despite the large variety of techniques, there are patterns, rules, and policies that apply to handling sensors and actuators. Sensing and controlling signals form the subject matter we model for this domain.

Because SIO must directly address aspects of the physical world, it may appear that this domain would need to specify platform-specific details. Our model contains abstractions of the real-world electronics that the system hardware presents to us, yet still contains no artifacts of the software platform on which the system executes. For example, we acknowledge that transducers in the outside world produce electrical signals that are converted into digital quantities and that technology forms the basis for interfacing physical quantities to the system. Our SIO model contains abstractions of that real-world fact. But the model artifacts of those abstractions do not include any reference to the specifics of the computing platform on which the system ultimately runs. Those specifics are, of course, supplied by the translation.

The role of the SIO domain in the ALS is to supply services to the Lubrication domain that it needs to accomplish proper lubrication of the machinery. The abstractions that capture the requirements of lubricating machinery are quite different from those of sensing and controlling the external environment. We say there is a *separation of concerns* between the two domains, as shown in Figure 7-1. The Lubrication domain is concerned with sequencing computation to cause machinery to be lubricated, without worrying about how to acquire the data it needs to perform those computations. The SIO domain is concerned with conditioning and transporting control and data across the system boundary, without caring what the data and control mean. As we show in the next chapter, pycca provides mechanisms for integrating the two worlds.

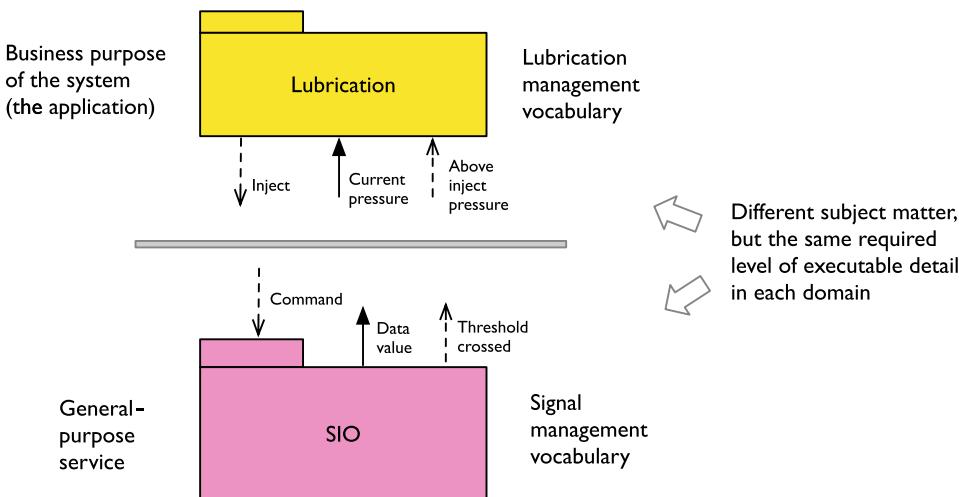


Figure 7-1. Separation of concerns

We sometimes characterize service domains such as SIO as being at a lower level. We do not use that term in the traditional sense, where higher/lower indicates a relative degree of detail. By *lower level*, we mean that the subject matter of a domain is less concerned with the business purpose of the application and instead focused on other enabling aspects of the total solution. In fact, a modeled domain cannot be more or less detailed. Every modeled domain is constructed with the same model elements: classes, states, actions, and so forth. To execute, all the elements must specify the same degree of detail. The level of detail is constant, and it is only the subject matter that varies across domains.

Separation of concerns is a tool we use to divide, conquer, and solve the larger problem at hand. By attending to only certain aspects of the larger problem, we find it easier to construct solutions. The separation has the added advantage of allowing us to construct the two domains simultaneously and of potentially reusing the SIO domain in another application that needs control over the outside world.

Of course, creating a separation means that it must at some time be filled. Again, we must emphasize that ignoring an aspect of a problem does not make it mysteriously disappear. We simply note that experience shows that “thinking inside the box” and then “connecting the boxes” together yields a more comprehensible and flexible solution. In the next chapter, we show how the Lubrication and SIO domains are connected back together via a process known as *bridging*.

Domain Overview

We do not describe all the aspects of the SIO domain in this chapter. The complete domain description and its translation is available as part of the online materials for the book. Our focus here is on translation, so we have picked a few areas of the domain that need to use features of pycca that you have not seen already.

The primary abstraction presented by the SIO domain is the I/O Point. An I/O Point is an idealized view of the way data values are passed between the system and the external world. Clients of SIO read a point to obtain a value of a sensor, and write to a point to assert control over the outside world. I/O Points are given simple small integer identifiers, such as 2 or 27, and clients use these identifiers to read or update the point value.

Clients of SIO always deal with engineering units. *Engineering units* are the common units with which we are familiar, such as meters, seconds, or kilopascals. Agreeing upon which engineering units are used by a project is vitally important, and there are some notorious examples of what happens if there is a discrepancy (the [Mars Climate Orbiter](#) is a relatively recent example). The electrical hardware that interfaces to the system always deals with its own set of units, which we call *device units*. Device units are specially encoded on a device-by-device basis for the benefit of the electronics hardware. For example, an analog-to-digital converter (ADC) reports the fraction of the measured voltage relative to a reference voltage. How the measured voltage is interpreted in terms of a physical quantity is an element of the electronics design that must specify the calibration or scaling from the ADC voltage to a meaningful physical quantity. One essential function of providing an idealized view of I/O values is to translate between engineering units and device units (and *vice versa*).

SIO does compute the correct values in device units for controlling the hardware interface but delegates that actual update and access of hardware to an external entity. This may seem strange, but delegating the access to the physical hardware controls to a *device access entity* means that the many and varied ways in which hardware controls are accessed does not become entangled into SIO. For example, some hardware controls are mapped into memory. Some are mapped onto external busses. Depending on the operating environment, special permissions may be required. Although SIO computes the right bits in the correct device units, it does depend on something else to transport those bits to the hardware control registers.

Often for the type of systems that we target here, device access code is supplied as part of a software library available from the manufacturer of the microcontroller chip. These libraries are often called *hardware access layers* and usually provide functions for common peripheral device operations. For example, a *system on a chip* (SOC) microcontroller may have supplied functions to run the on-board ADC. Such libraries are useful because they usually encode details of the device operations that can be difficult to determine solely from the chip design manual. It is necessary to supply a bridge from SIO to the device access code. Such a bridge follows the same pattern of bridging we describe in the next chapter. For hardware access, we are bridging to manually written code (written either by the chip manufacturer or by the project), and it is part of the solid ground we use to build up the rest of the system.

Converting Electrical Signals

Most systems that interact with the outside world need to measure physical quantities. Even conventional desktop computers usually measure the temperature of the processor to control the fan speed and ensure that the CPU doesn't overheat.

A common way to measure physical quantities is to use a transducer. A *transducer* is a device that produces an electrical signal, typically a voltage, that can be related to a physical property, usually by linear scaling. An ADC is typically used to interface transducer signals.

Digitizing signals is an ubiquitous operation and a distinct specialty in the computing world. We take a much simpler view of things here. Figure 7-2 shows in block form the hardware arrangement that we assume.

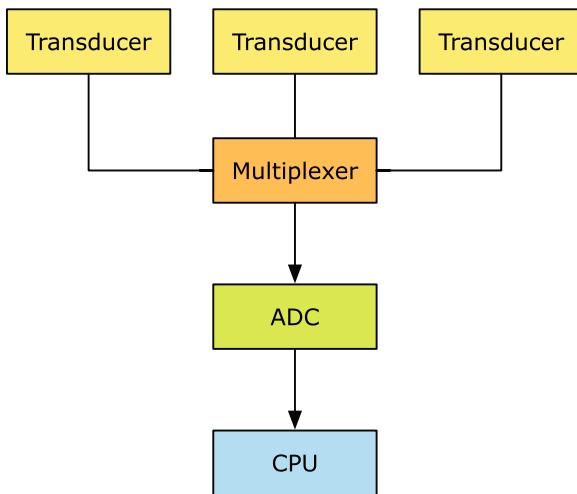


Figure 7-2. Block diagram of ADC hardware

The ADC typically has a *multiplexer* that acts as a switch. This allows many signals to be wired physically to the ADC, and one is selected via the multiplexer. It is also common that several signals need to be sampled at the same time. This might happen if the signals represent different instances of the same quantity sampled at the same rate or have some other correlation in time such as two quantities used to compute a third value. The ADC cannot convert multiple signals simultaneously, but it can convert a signal and switch the multiplexer to convert another signal fast enough that it can be effectively time shared between multiple signals. Most ADC peripherals support converting a series of inputs as a distinct mode of operation requiring no additional CPU intervention.

Modeling Signal Conversion

Figure 7-3 shows the fragment of the class diagram that deals with signal conversion.

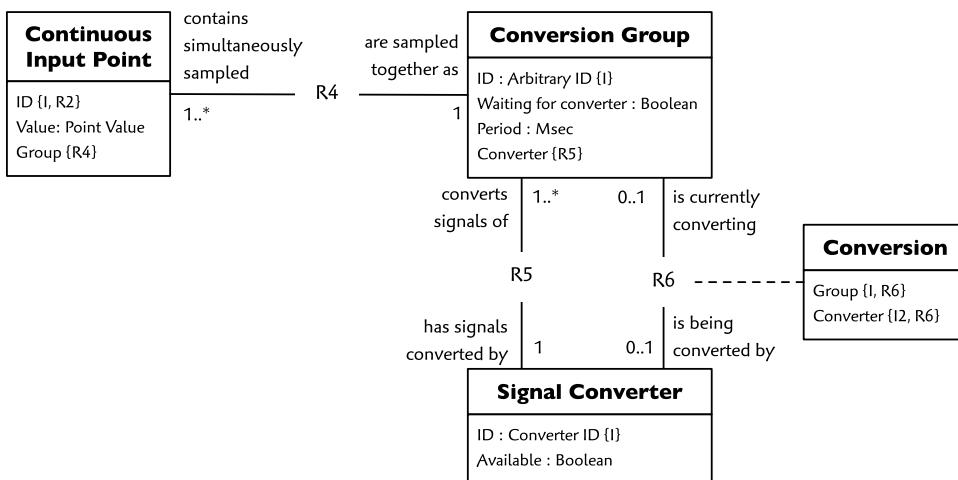


Figure 7-3. Classes dealing with signal conversion

The essential rules are as follows:

- Input points that must be sampled at the same time are grouped together (R4). A group of one is used for those points with no correlation to other points.
- Grouped input points are all wired to the same Signal Converter (R5).
- At any point in time, at most one group of signals is being converted and there are times when the Signal Converter is idle (R6).

Each Conversion Group has its own sampling period. The period is the amount of time to elapse between samples being taken. This implies that at some point in time, two conversion groups may need access to the same Signal Converter, or the Signal Converter may be in use at the same time a conversion group decides to sample its inputs.

From the modeling point of view, we must be careful about how the R6 association is managed. It is necessary to serialize the processing associated with creating and deleting instances of the R6 association. This is accomplished with an *assigner*. An assigner is a state model attached to an association that manages competition among the instances of the association. Rather than state activities creating or deleting instances of the association synchronously, they signal events to the assigner declaring their intent. The state activities of the assigner function as a single point of control to coordinate the life cycles of the association instances to ensure proper access to the Signal Converter. Because our focus here is translation, we do not discuss the general background of competitive associations and the protocols used to ensure the proper sequencing of operations. See *Executable UML: A Foundation for Model-Driven Architecture* for more about how to recognize and model competitive associations and construct assigner state models.

Figure 7-4 is a sequence diagram for one possible event sequence in which a group of points needs to be sampled. This is the sequence that would happen the first time a conversion group is requested to be sampled, assuming the corresponding Signal Converter is not being used.

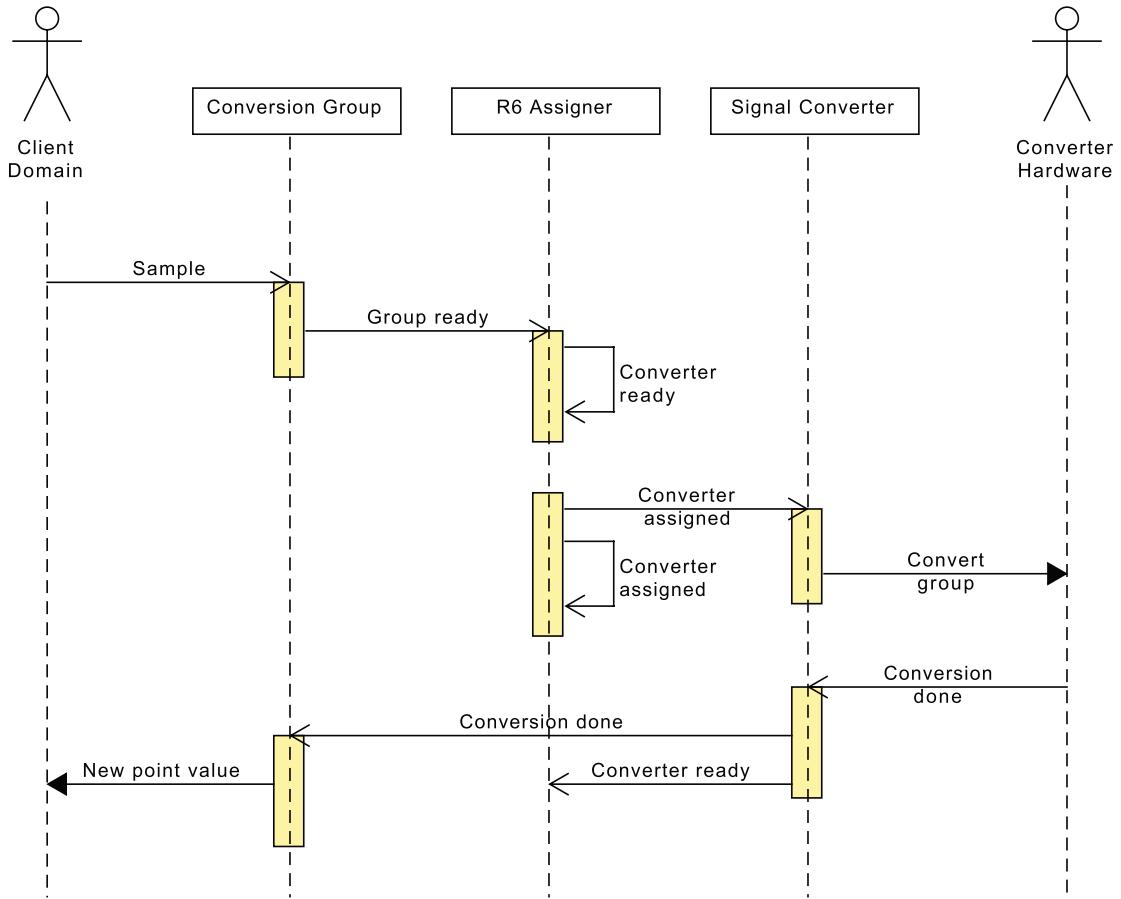


Figure 7-4. Event sequence assigning a Signal Converter

The state models for the R6 assigner along with those for Conversion Group and Signal Converter cooperate to ensure that Signal Converter access is properly serialized among the competing instances of Conversion Group. Clients can request the Conversion Group to begin sampling, which, in turn, signals the R6 assigner it is ready. Because this is the first time through, the assigner detects that the Signal Converter is available and signals *Converter ready* to itself. That transition causes the Signal Converter to be signaled with the *Converter assigned* event, which causes the Signal Converter to request the conversion from the hardware. Meanwhile, the R6 assigner signals to itself that a converter is assigned and transitions to a state in which no other conversions are allowed until the ongoing one is completed.

When the hardware finishes the conversion, it signals *Conversion done* to the Signal Converter, which reads the converted values and processes them. The Conversion Group is signaled to tell it that the sampling has been accomplished, and the assigner is signaled with the *Converter ready* event to inform it that another Conversion Group may be assigned.

Figure 7-5 shows the state model for the R6 assigner.

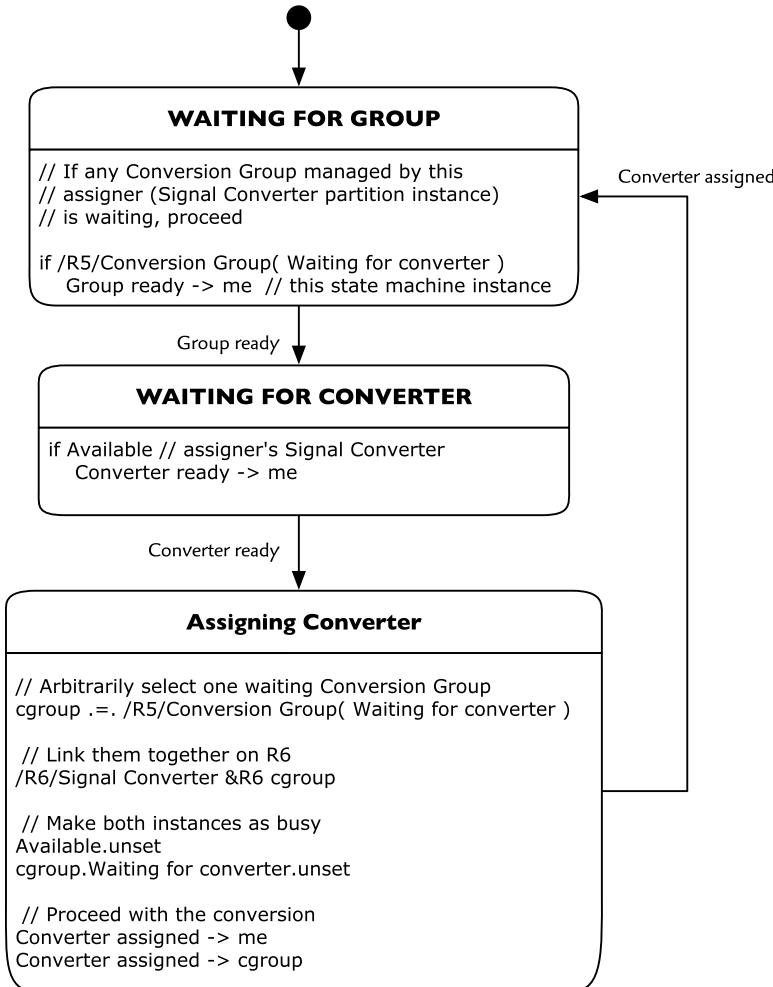


Figure 7-5. R6 assigner state model

Note in particular that for this state model, if another Conversion Group signals *Group ready* while a converter is still converting the signals of a previously assigned group, the state machine remains in the **WAITING FOR CONVERTER** state until the Signal Converter signals *Converter ready*, indicating that it is ready to perform another conversion. This ensures that two conversion groups don't try to use the signal converter at the same time. Conversely, if the conversion finishes before another group needs to be sampled, the *Converter ready* event is ignored in the **WAITING FOR GROUP** state as we remain in that state until a Conversion Group requests the signal converter. The state model defines a sequencing protocol between the Conversion Group and Signal Converter classes to ensure that access to the Signal Converter is serialized properly.

The following transition matrix is for the R6 assigner.

	Group ready	Converter ready	Converter assigned
WAITING FOR GROUP	WAITING FOR CONVERTER	IG	CH
WAITING FOR CONVERTER	IG	Assigning Converter	CH
Assigning Converter	CH	CH	WAITING FOR GROUP

Note the ignored events. Ignoring the *Converter ready* event when waiting for a Converter Group and ignoring the *Group ready* event when waiting for a Signal Converter are an essential aspect of the way the R6 assigner serializes the access to the Signal Converter by the competing Converter Groups.

Figure 7-6 shows how a Conversion Group notifies the assigner that it needs the Signal Converter.

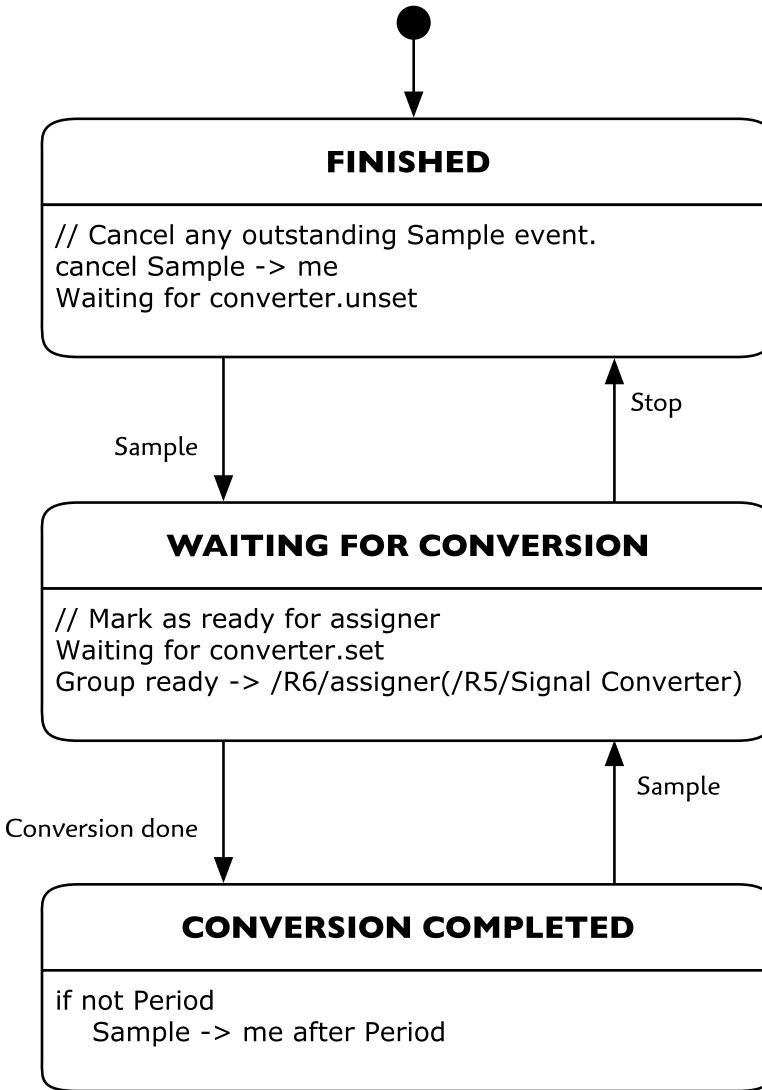


Figure 7-6. Conversion Group state model

The Conversion Group signals the assigner that it needs to perform a conversion and then waits to be told that the conversion has been done. The Sample and Stop events are used to start and stop the periodic conversion of the group of points.

Finally, Figure 7-7 shows how the Signal Converter operates.

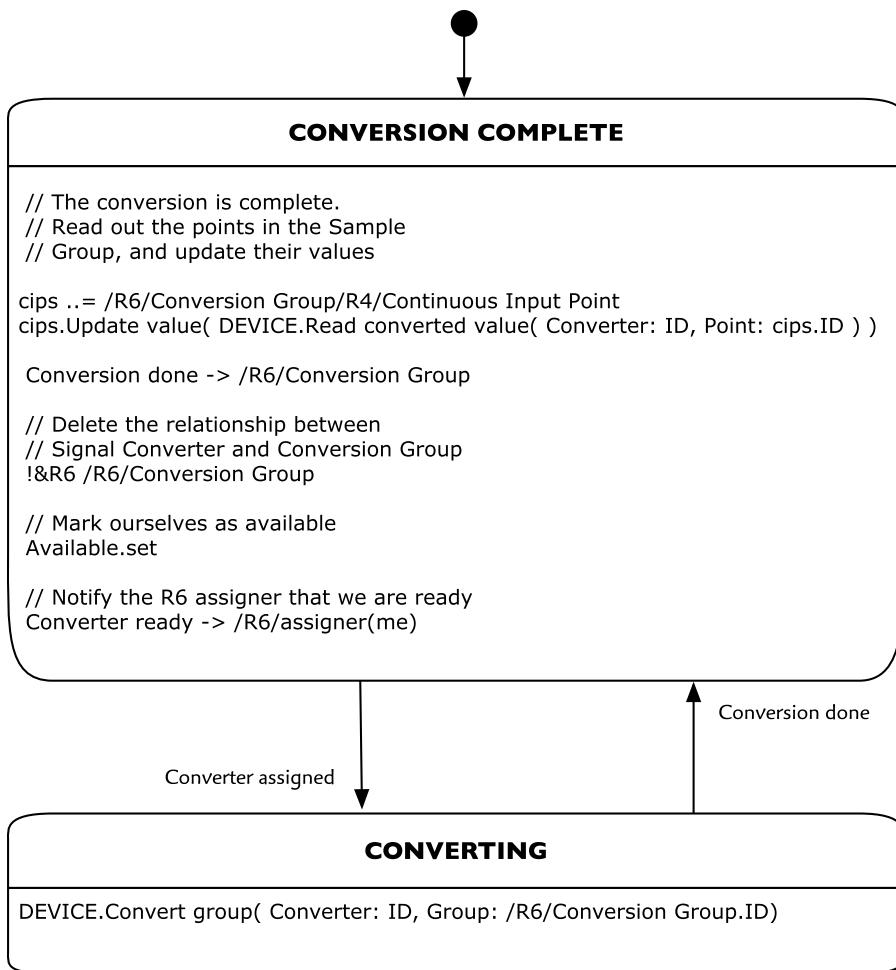


Figure 7-7. Signal Converter state model

After being assigned to a Conversion Group by the R6 assigner, the Signal Converter requests the device to perform the conversion and then waits for it to be done. Then it is necessary to retrieve the converted values and update the values of the input points. Note that the CONVERSION COMPLETE state activity deletes the instance of the R6 association because it knows when the conversion process is complete and because there is a one-to-one correspondence between Signal Converter and R6 association instances.

Because input points are physically wired by the electronics design to a particular Signal Converter (as stated by R5), there must be an instance of the R6 assigner for each Signal Converter class instance. It is not possible to allow an arbitrary Signal Converter to sample an arbitrary Conversion Group. Such an arrangement is called a *multi-assigner* because there are multiple instances of the assigner state machine. For the simpler case in which an arbitrary resource can be allocated to an arbitrary user of that resource, there needs to be only a single instance of the assigner. For a multi-assigner, one class always serves to partition the instances of the assigners among the resource users. Formally, a multi-assigner can be identified by the same identifying attributes as for the partitioning class. In this case, it is the Signal Converter class that partitions the assigner instances, and the number of instances of the R6 assigner state machine equals the number of instances of the Signal Converter class.

Implementing the Assigner

We use a pycca class to implement the R6 assigner. Although the assigner is *not* a class from the model point of view, a pycca class has all the implementation characteristics we need—namely, attributes, state models, and the ability to have multiple instances. This also demonstrates that a pycca class is an implementation construct and *not* the same as a model class. We use a pycca class to implement a model class, but we also use it to implement an assigner.

The pycca implementation follows the usual pattern we have already seen; however, the callouts point to distinct usage that is specific to implementing assigners:

```
class R6_Assigner
    reference idclass -> Signal_Converter
    machine
        default transition CH
        initial state WAITING_FOR_GROUP

        transition WAITING_FOR_GROUP - Group_ready -> WAITING_FOR_CONVERTER
        transition WAITING_FOR_GROUP - Converter_ready -> IG
        transition WAITING_FOR_CONVERTER - Group_ready -> IG
        transition WAITING_FOR_CONVERTER - Converter_ready -> Assigning_Converter
        transition Assigning_Converter - Group_ready -> IG
        transition Assigning_Converter - Converter_ready -> IG
        transition Assigning_Converter - Converter_assigned -> WAITING_FOR_GROUP

    state WAITING_FOR_GROUP () {
        ClassRefVar(Signal_Converter, sc) = self->idclass ;

        ClassRefConstSetVar(Conversion_Group, cgset) ;
        PYCCA_forAllRelated(cgset, sc, R5) { // ②
            ClassRefVar(Conversion_Group, cg) = *cgset ;
            if (cg->Waiting_for_converter) {
                PYCCA_generateToSelf(Group_ready) ;
                return ;
            }
        }
    }
    state WAITING_FOR_CONVERTER () {
```

```

ClassRefVar(Signal_Converter, sc) = self->idclass ; // ❸
if (sc->Converter_available) {
    PYCCA_generateToSelf(Converter_ready) ;
}
}

state Assigning_Converter () {
    ClassRefVar(Signal_Converter, sc) = self->idclass ;
    assert(sc->Converter_available) ;
    ClassRefConstSetVar(Conversion_Group, cgset) ;
    PYCCA_forAllRelated(cgset, sc, R5) {
        ClassRefVar(Conversion_Group, cg) = *cgset ;
        if (cg->Waiting_for_converter) { // ❹
            sc->R6 = cg ;

            sc->Converter_available = false ;
            cg->Waiting_for_converter = false ;
            PYCCA_generateToSelf(Converter_assigned) ;
            PYCCA_generate(Converter_assigned, Signal_Converter, sc, self) ;
            return ;
        }
    }
}
end
end

```

❶ Because this is a multi-assigner partitioned by Signal Converter, we need a reference to the Signal Converter instance on whose behalf this assigner instance is operating. The reference to an identifying class has the net effect of giving the assigner the same identifying attributes as Signal Converter.

❷ Iterate across the instances of Conversion Group related by R5 looking for one waiting to be sampled. Here, cgset is an iterator that is assigned successive values of pointer to the instances related across R5.

❸ There is no need to search for the Signal Converter that this instance is assigning. There is a one-to-one correlation between R6 assigner instances and Signal Converter instances.

❹ Again, we find a group that is waiting to be converted. We select the first one we find and assign the Signal Converter to it.

There is no specific policy implemented for deciding between multiple, waiting Conversion Groups. If the requirements, and hence models, specified a particular policy for resolving the contention, it would be implemented here. One possible policy might be “first come, first served,” but that is *not* what this code does. This code assigns Conversion Groups based solely on the order in which they were related to the Signal Converter. This would correspond to a requirement of assigning an arbitrary waiting Conversion Group and, because the requirements allow for any Conversion Group to be assigned, we have chosen the most convenient and efficient implementation. In general, the requirements for selecting how a resource might be allocated in the presence of contention can be complex. Here we have taken the simple approach for the benefit of the example.

Tracing Execution

As we discussed in Chapter 5, translated models executing on the ST/MX domain can generate an execution trace. The following is a trace of the event transitions involving the R6 assigner. In this scenario, two instances of Conversion Group are both connected to the same Signal Converter. The event sequence has been constructed to ensure that there is competition for access to the Signal Converter. The trace shows how the R6 assigner correctly serializes the Converter Group requests.

```

1  (nil) - Sample -> Conversion_Group.cg1: FINISHED -> WAITING_FOR_CONVERSION ①
2  Conversion_Group.cg1 - Group_ready -> R6_Assigner.r6asgn1: WAITING_FOR_GROUP -> ↵
   WAITING_FOR_CONVERTER
3  R6_Assigner.r6asgn1 - Converter_ready -> R6_Assigner.r6asgn1: WAITING_FOR_CONVERTER -> ↵
   Assigning_Converter
4  R6_Assigner.r6asgn1 - Converter_assigned -> R6_Assigner.r6asgn1: Assigning_Converter -> ↵
   WAITING_FOR_GROUP
5  R6_Assigner.r6asgn1 - Converter_assigned -> Signal_Converter.cvt1: CONVERSION_COMPLETE -> ↵
   CONVERTING
6  (nil) - Sample -> Conversion_Group.cg2: FINISHED -> WAITING_FOR_CONVERSION ②
7  Conversion_Group.cg2 - GroupReady -> R6_Assigner.r6asgn1: WAITING_FOR_GROUP -> ↵
   WAITING_FOR_CONVERTER
8  (nil) - Conversion_done -> Signal_Converter.cvt1: CONVERTING -> CONVERSION_COMPLETE ③
9  Signal_Converter.cvt1 - Conversion_done -> Conversion_Group.cg1: WAITING_FOR_CONVERSION -> ↵
   CONVERSION_COMPLETED
10 Signal_Converter.cvt1 - Converter_ready -> R6_Assigner.r6asgn1: WAITING_FOR_CONVERTER -> ↵
   Assigning_Converter
11 R6_Assigner.r6asgn1 - Converter_assigned -> R6_Assigner.r6asgn1: Assigning_Converter -> ↵
   WAITING_FOR_GROUP
12 R6_Assigner.r6asgn1 - Converter_assigned -> Signal_Converter.cvt1: CONVERSION_COMPLETE -> ↵
   CONVERTING
13 (nil) - Conversion_done -> Signal_Converter.cvt1: CONVERTING -> CONVERSION_COMPLETE ④
14 Signal_Converter.cvt1 - Conversion_done -> Conversion_Group.cg2: WAITING_FOR_CONVERSION -> ↵
   CONVERSION_COMPLETED
15 Signal_Converter.cvt1 - Converter_ready -> R6_Assigner.r6asgn1: WAITING_FOR_GROUP -> IG

```

① The first five lines of the trace show the sequence of event dispatches to start the sample conversion for Converter Group cg1. These events correspond to those shown in the top part of Figure 7-4.

② Converter Group cg2 is requested to sample *before* the hardware has completed the conversion for Converter Group cg1. The *Group ready* event sent to the R6 assigner (line 7) causes it to transition to the WAITING FOR CONVERTER state, and there it examines the Signal Converter to determine that it is not available at this time. This causes the assigner to remain in the WAITING FOR CONVERTER state.

③ The hardware has completed its work, and this initiates a sequence of event dispatches that eventually end up with the Signal Converter signaling *Converter ready* to the R6 assigner. Because a Conversion Group is ready to go, it is assigned the Signal Converter, and the next sample is initiated.

④ The second conversion is done. When the Signal Converter signals *Converter ready* to the assigner, the event is ignored (line 15), as there are no ready Conversion Groups at this time.

Limitations

Sampling values in the manner that we have shown here is limited to relatively low frequencies, on the order of a few hundred Hertz. The timing for sampling points is controlled by software, and that will necessarily create *jitter* in the timing precision. The fastest that can be sampled is 1,000 Hz, because we are using delayed signals and the delay time resolution is 1 millisecond.

In the larger data acquisition world, this is a low data rate, so our simple example should not be applied indiscriminately. It is, of course, possible to do better, but it means moving more of the work into hardware. Our view of that underlying hardware would be different from what we showed earlier. Each hardware arrangement will require some software control, and that can be modeled and implemented by using the techniques shown here. What differs is the underlying hardware reality that the software model controls. What does not change is the service interface that the SIO domain presents.

Value Thresholds

Frequently, we are not so concerned about the value of a point but rather how that value compares to a threshold. In the Lubrication domain, we saw the domain needed to have the lubrication pressure monitored to ensure that it was in the correct pressure range to be effective. To support that delegated capability, the SIO domain implements value threshold concepts and provides a way to compare values to defined limits.

Figure 7-8 shows an example of a value threshold. As point values are sampled over time, the value may rise above a specified threshold limit. When that happens, the value is deemed *out of range*. It remains out of range until it falls below the threshold limit at which time it is deemed to be *in range*.

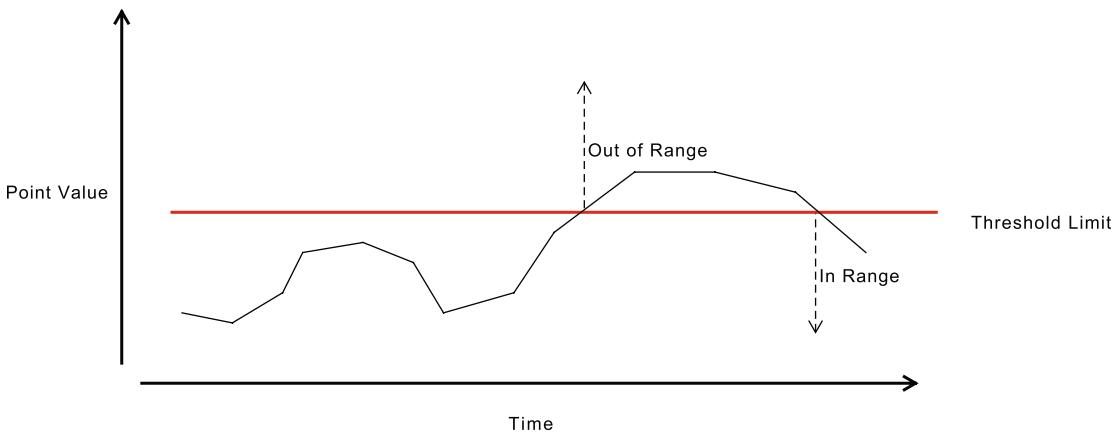


Figure 7-8. Threshold limit on a value

This diagram shows a *rising-edge* detection of the out-of-range condition. If you invert the value comparisons, you have the complementary form of *falling-edge* detection, which detects a point considered out of range when the point value falls below the threshold limit. By defining two threshold limits, one for a rising-edge comparison and one for a falling-edge comparison, then a value can be monitored to determine whether it is between upper and lower limits.

When deciding whether a value exceeds a threshold, we want to prevent oscillations near the threshold limit from repeatedly triggering the out-of-range condition followed immediately by an in-range condition. We use a simple counting filter to supply *hysteresis* to the detection. If we delay the determination that a value is out of range or in range until we have seen a certain number of consecutive excursions above or below the threshold, we can prevent minor fluctuations of the value causing a spurious detection of being out of range. If we want to see each time a value exceeds its threshold, we can set the limit number to 1. There are, of course, many other possible ways to add hysteresis to the threshold determination, but simple counting schemes are both effective and, when translated, computationally efficient.

Figure 7-9 is a class diagram that captures these ideas.

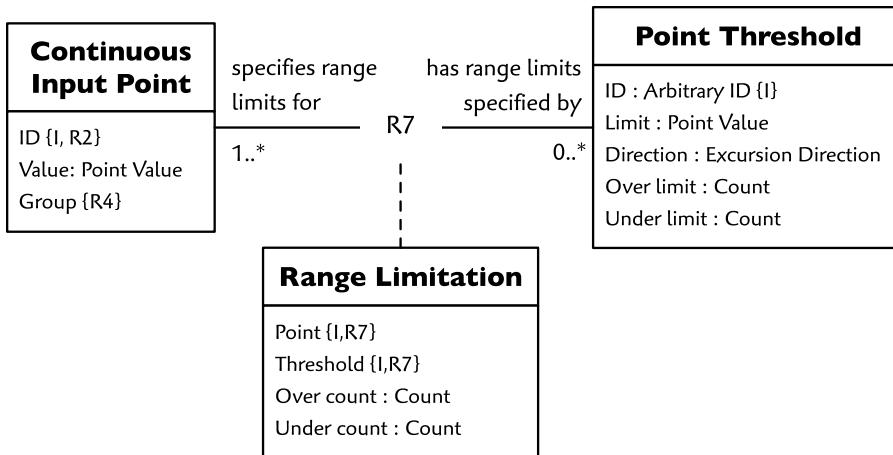


Figure 7-9. Class diagram for Range Limitation

Each input point may have multiple Point Thresholds specified for it. In our example, we use three Point Thresholds to monitor an injector's lubrication pressure to know if the lubricant pressure is sufficient, if there is excessive dissipation pressure, or if the maximum allowed pressure is exceeded. A Range Limitation is an instance of applying a Point Threshold to the value of a Continuous Input Point. We may define one Point Threshold and apply it to several input points, but we will need a separate instance of Range Limitation for each of those applications. The computations necessary to detect the threshold excursions are specified in the state model for the Range Limitation class.

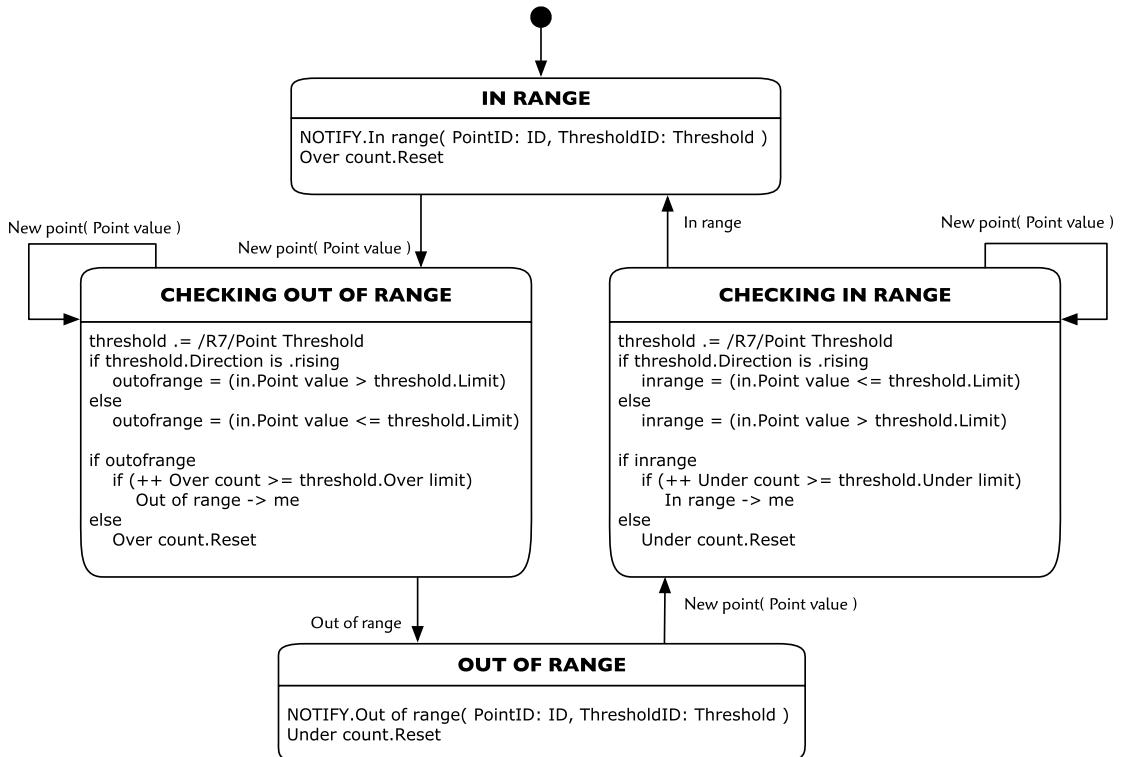


Figure 7-10. Range Limitation state model

We do not show the entire pycca implementation of the state model here because it follows the same pattern you have already seen. We do, however, show the state definition of the Checking Out Of Range state:

```

state CHECKING_OUT_OF_RANGE(
    sio_Point_Value pointValue) {
    ClassRefVar(Point_Threshold, pt) = self->R7_PT ;

    bool outRange = pt->Direction == Rising ?
        rcvd_evt->pointValue > pt->Limit :
        rcvd_evt->pointValue <= pt->Limit ;           // ①
    if (outRange) {
        if (++self->Over_count >= pt->Over_limit) {
            PYCCA_generateToSelf(Out_of_range) ;
        }
    } else {
        self->Over_count = 0 ;
    }
}

```

① Pycca defines a pointer named `rcvd_evt` to access event parameters. The variable name and parameter access technique is an unfortunate choice that will be improved in a future release.

Initial Instance Population

In this section, we present part of an initial instance population for the Signal I/O domain. This population corresponds to the needs of the Lubrication domain as it was populated in Chapter 6. We do not show the entire set of values in the population, but only those parts that pertain to converting signals and detecting value threshold limits. The complete population is available in the online materials.

Supporting the needs of the Lubrication domain population requires eleven I/O Points: three for Machinery lockouts, two for Reservoir levels, three for Injector pressure, and three for Injector solenoids. We focus here on the three I/O Points associated with the Injector pressure because they must be both sampled and compared against threshold limits.

Injector pressure is represented by a Continuous Input Point whose value represents the lubricant pressure at each injector. Each injector has a transducer for pressure that is sampled and converted to the value of the Continuous Input Point.

ID	Value	Group
inj1_pres	0	inj1_cg
inj2_pres	0	inj2_cg
inj3_pres	0	inj3_cg

The I/O Points for the injector pressure are formed into three Conversion Groups, each group containing only a single point.

ID	Waiting for converter	Period	Converter
inj1_cg	false	500	cvt1
inj2_cg	false	500	cvt1
inj3_cg	false	500	cvt1

We assume that only a single Signal Converter is attached to the system and that all the injector pressure transducers are wired to it.

ID	Converter available
cvt1	true

The Lubrication domain assumes that Signal I/O will perform threshold comparisons on the injector pressure values. Three thresholds must be determined: above the dissipation pressure, above the injection pressure, and above the maximum pressure. From the Lubrication domain population, we know that there are two Injector models. Our pressure threshold values are selected based on the different injector designs.

ID	Limit	Direction	Over limit	Under limit
ix77b_above_disp	26	Rising	2	2
ix77b_above_inj	15	Rising	2	2
ix77b_max_pres	26	Rising	1	2
ihn4_above_disp	32	Rising	2	2
ihn4_above_inj	19	Rising	2	2
ihn4_max_pres	35	Rising	1	2

With three injectors and three thresholds for each injector design type, we have nine instances of Range Limitation to detect all the threshold excursions that the Lubrication domain requires.

Point	Threshold	Over count	Under count
inj1_pres	ix77b_above_disp	0	0
inj1_pres	ix77b_above_inj	0	0
inj1_pres	ix77b_max_pres	0	0
inj2_pres	ihn4_above_disp	0	0
inj2_pres	ihn4_above_inj	0	0
inj2_pres	ihn4_max_pres	0	0
inj3_pres	ix77b_above_disp	0	0
inj3_pres	ix77b_above_inj	0	0
inj3_pres	ix77b_max_pres	0	0

Summary

In this chapter, we have shown a model and translation for the SIO domain. This domain handles interactions with the external world by sensing values and controlling actuators. We focused on two particular capabilities that are delegated to the SIO domain by the Lubrication domain:

- Sampling values of sensors
- Comparing sampled values against thresholds

For value sampling, our model included an assigner state model to manage the competition that is inherent in the physical arrangement of our electronics design.

For value thresholds, we saw how a state model attached to an association class provides the means to track multiple threshold limits applied to the same input point.

In both cases, we saw rules and policies applied to collecting and evaluating data from the external world. From the point of view of the SIO domain, the values and meaning of the data were no of concern. From the point of view of the Lubrication domain, it needs current, up-to-date data values, and the rules of how the electronics design affects acquiring data is of no concern. This separation of concerns allows us to focus on the development of a consistent set of abstractions for both domains and to potentially reuse the SIO domain in another application that has similar needs to interact with the external world.

CHAPTER 8



Integrating the Application and Service Domains

In this chapter, we discuss how to manage interactions among domains. Because we are bridging the gap from the semantics of one domain to that of another, we call this topic *bridging*. From a modeler's perspective, bridging two domains can be relatively simple. This assumes that no semantic content is missing between the two domains. For example, it would be difficult to bridge a video game application directly to the rendering hardware without having at least an intermediate draw engine. We have avoided this kind of trouble by carefully constructing the domain chart for the ALS example.

Efficient implementation of a bridge, on the other hand, can pose its own set of problems. Pycca provides support for bridging, but a certain amount of code does have to be written. Fortunately, this can be done systematically with pycca providing help to ease the process.

Before diving into implementation, as always, we need a clear and detailed vision of what must be accomplished. The first part of this chapter presents useful building blocks for visualizing and specifying the data necessary to define a bridge. We apply this to a few key interactions between the Lubrication and SIO domains. Then, in the second part of this chapter, we show how to specify this data by using pycca-generated constants in conjunction with handwritten C code. Finally, we show some of the details of how pycca supports performing model-level actions from outside a domain.

Summary of Domain Benefits

As we proceed into all the details of bridging, you may wonder why we are going to all this effort to preserve the semantic integrity of our domains. So let's first take stock of why domains are important. A key goal of domain engineering is to promote reuse of domains. If a domain is redesigned but provides the same specified services, it should have minimal, preferably zero, impact on the models of any domains with which it interacts.

For example, if we (or someone else) devise a completely different way of managing SIO, it should make no difference to the Lubrication domain. As long as the *Above injection pressure* event gets delivered, under the proper physical conditions, the Lubrication domain's needs are satisfied. The logic of the Lubrication domain isn't aware of and doesn't care about how the event is delivered.

More benefits than reusability are at stake. By keeping the models separate, it becomes possible to develop them in parallel without a lot of coordination overhead. The expertise required to build one domain is distinct from that required to build the other. This means that you can employ entirely different specialists to develop the models of each domain. Complexity is reduced because each domain model can be built without having to worry about the details of how the other works. This outcome derives from applying the principle of *separation of concerns*, as we discussed in the first part of Chapter 7.

Each Domain Is a Black Box

To preserve these benefits, the models in two bridged domains view one another as black boxes. SIO doesn't model anything about Lubrication, and Lubrication doesn't model anything about SIO. The Lubrication domain does not know which events, if any, ping around in the SIO domain. It knows nothing of SIO's internal states or actions. You can rename and reorganize the model content in one domain all you want without breaking anything in the other domain. But this opacity principle applies only to the model structures, not necessarily the data they contain! For example, an instance of the SIO Point Threshold class can have its Limit attribute set equivalent to the *Max delivery pressure* of a corresponding instance of Injector Design in the Lubrication domain. Or some instances of SIO's Continuous Input Point class may correspond to specific instances of Injector in the Lubrication domain. Depending on usage, this data may be glued together at model time, compile time, initialization time, or dynamically at normal runtime.

Figure 8-1 depicts this mystery box principle.

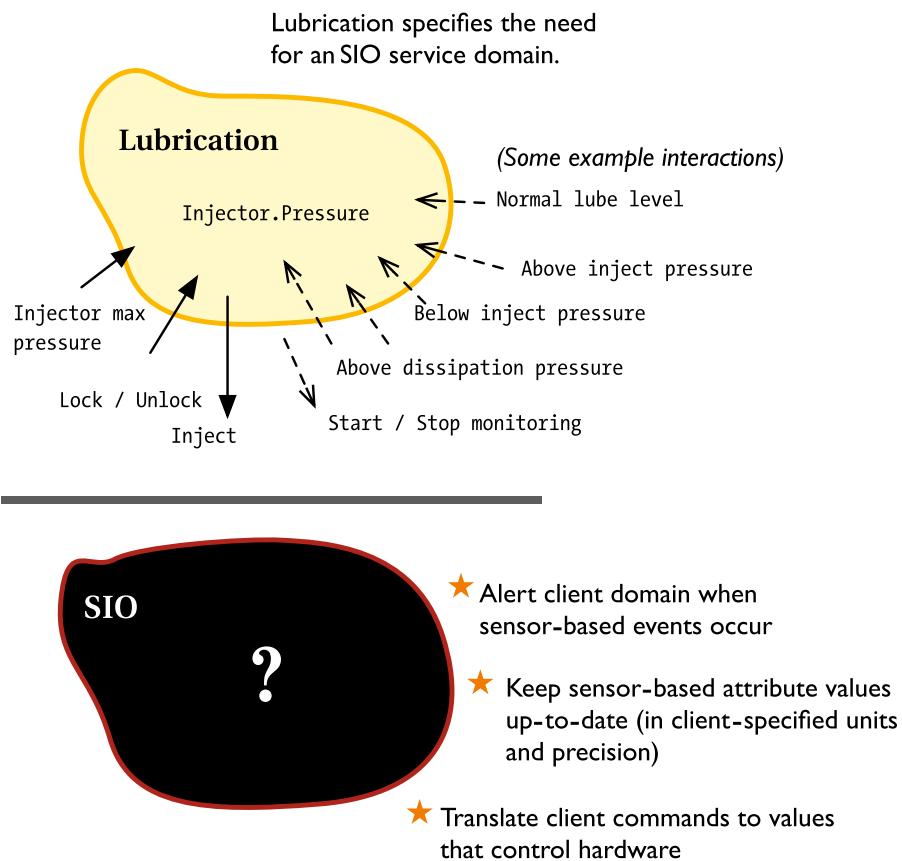


Figure 8-1. Lubrication sees SIO as a black box

We can see that the Lubrication domain assumes the existence of an SIO domain that triggers sensor-based events, method calls, and attribute values. It also assumes that SIO can manage control directed at the physical world. But Lubrication has no idea how any of this is accomplished or what structures (modeled or nonmodeled) are involved in SIO.

The Lubrication domain has an external entity named SIO that serves as a proxy for these assumptions. Keeping the name of the proxy external entity and the actual domain the same is a convenient mnemonic, but it is important to remember that the SIO external entity is *not* the same thing as the SIO domain. The SIO external entity is the Lubrication domain's view of how it has delegated functionality outside its scope to be accomplished elsewhere. The Lubrication domain uses the external entity as a proxy so that it can express its requirements for actuator and sensor service in a way that is consistent with the structure and activities of the domain. If the SIO external entity was in fact the same as the SIO domain, we would expect to be able to connect the external entity operations arising in Lubrication directly to domain operations targeted at SIO, and there would be no semantic gap to bridge. This is definitely *not* the situation we want. If there were no semantic gap between the domains, the SIO domain would have to know the details of how the Lubrication domains works. We would have sacrificed reducing complexity by separating the concerns of the domains and abandoned any chance of reusing the SIO domain in another application.

Figure 8-2 shows the view looking back from SIO's perspective. We get a similar picture.

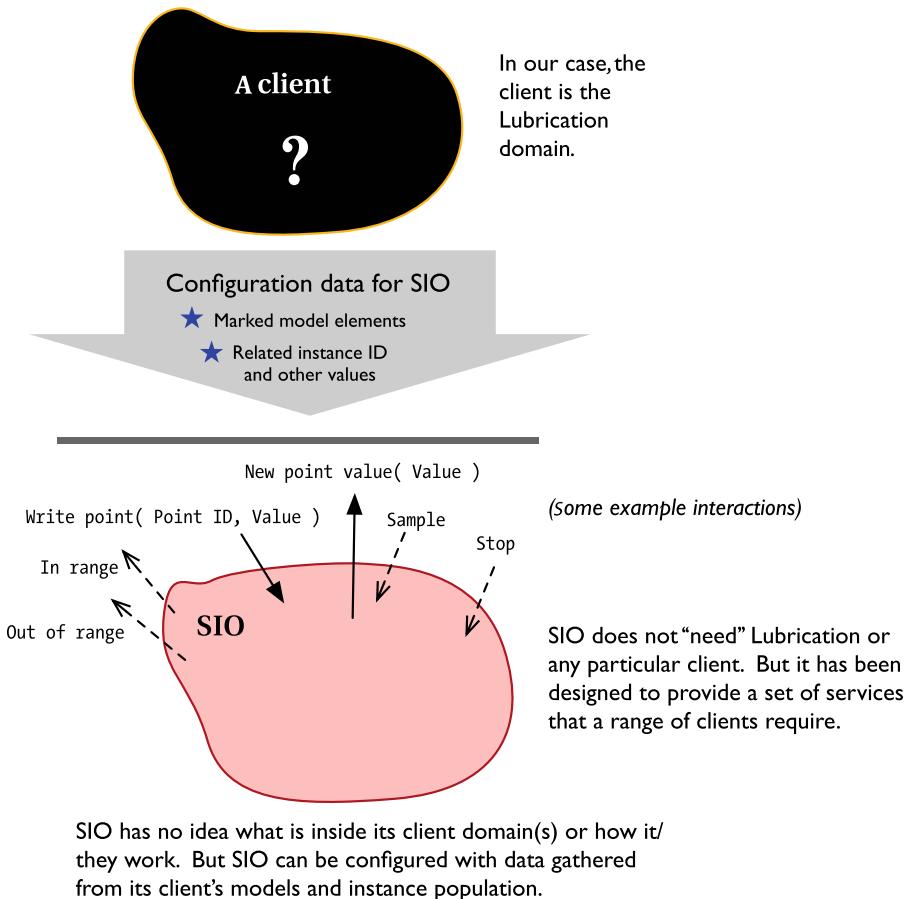


Figure 8-2. SIO sees its clients as black boxes

SIO has no idea what *Inject* means. But it does know how to write a value out to a hardware register with the *Write point* domain operation. Similarly, other SIO services are defined in a vocabulary that is consistent with the view SIO has of its subject matter. SIO is not completely unconstrained in what it does. The services provided are intended to be sufficient for a client domain such as Lubrication to interact with the physical world, and the Lubrication domain does make known the specific services it needs.

Despite the black-box view that each domain takes of the other, when we bridge the Lubrication and SIO domains together, we must provide data values to certain SIO classes that account for the specific behavior the Lubrication domain requires. Those SIO classes were parameterized by having attributes whose values control the processing details. For example, the SIO domain has attributes that specify a value range limit as part of its service to provide its clients with alerts. We provide values for these types of attributes with data gathered from the models and instance populations of the Lubrication domain, or any other client domain that is serviced by SIO.

Note also that while the Lubrication domain needs SIO in order to function, SIO doesn't really need anything from Lubrication to perform its duty. This is the nature of a client/service domain bridge. The client imposes requirements on the service, and the service is populated and bridged to fulfill them. That explains the direction of the dependency arrows on the domain chart; these arrows are sometimes misinterpreted as indicating some type of flow of control or data between domains. They are instead intended to represent the flow of requirements or dependency between the domains. The actual control or data flow between domains is determined later, as part of the bridging effort.

Marking and Mapping

Now let's take a look at what data is required to connect one domain to another across a bridge. We need a way to define how certain data and activity in one domain maps to corresponding data in the other domain. This is done using two methods: marking and mapping.

Marking is the process of identifying and classifying model elements in a client domain that can be used to configure properties in a service domain. For example, we mark attributes in the Lubrication domain whose values are sensor driven. *Mapping* is the process of defining the correspondence between marked elements in the client domain and supporting model elements in a service domain. For example, a Pressure attribute in Lubrication will correspond to the sampled and scaled value of an SIO Continuous Input Point. There must then be a way to map each instance of Injector to its counterpart instance of Continuous Input Point in SIO.

It is critical that we do this in such a way that is complete, but doesn't specify any particular implementation. That way, when we implement, we have all the marking and mapping data we need to make the system work. We also have the freedom to implement the bridge efficiently, given our particular platform technology.

Look at Figure 8-3, a snippet from the Injector state model in the Lubrication domain.

External entity operations invoked from the
Injector state model marked as “actuator controls”

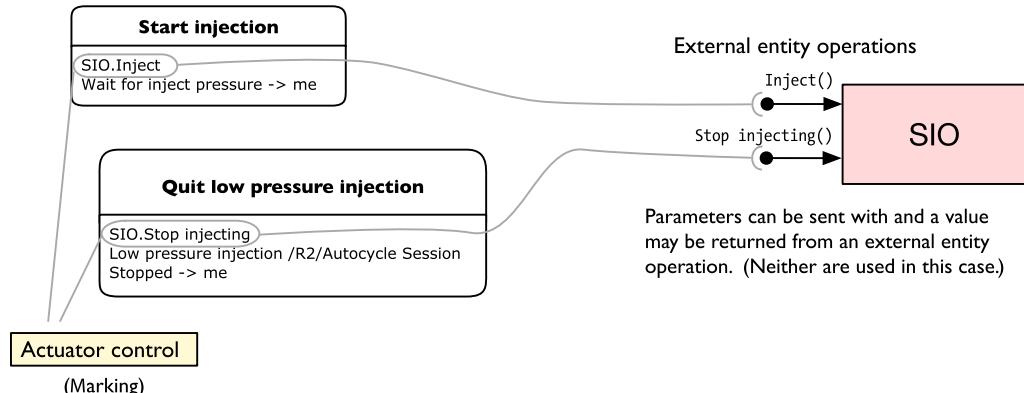


Figure 8-3. External entity operations

On the right-hand side, we see the external entity acting as a proxy for the SIO domain. We have marked two Injector state actions that are special in that they trigger control in the physical environment via some sort of actuator. So we declare them to be *Actuator controls*. They also happen to be external entity operations. They invoke operations defined on an external entity that represents the SIO domain. An external entity operation can be invoked with parameters and return a value. Neither operation in this example specifies any parameters or expects any return value. Our action language presumes that the ID of the calling instance is implicitly available via any external entity operation, so it is not necessary to explicitly supply the Injector ID.

Figure 8-4 shows what the SIO *domain* must do in response to the operations on the SIO *external entity*.

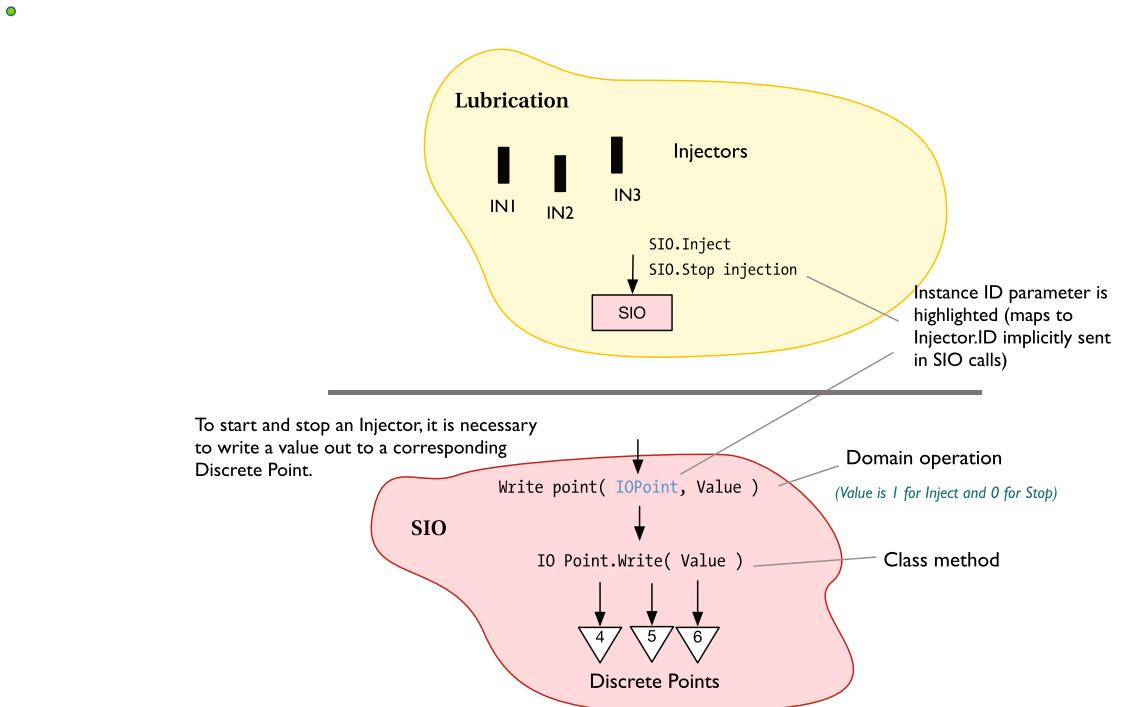


Figure 8-4. A domain operation

The SIO domain provides the *Write point* domain operation. This operation takes an I/O Point parameter, which is the value of an identifier of the target I/O Point and a Value to write. If 1 is written, injection is started, and if 0 is written, injection is stopped. Note that the Injector and I/O Point parameters have been highlighted to indicate that each carries an ID value corresponding to an instance. They are handled a bit differently than the Value parameter, as you will see.

So our task is to map the marked external entity operations in the Lubrication domain onto the domain operation in the SIO domain. To accomplish this, we will use something called *half tables*. You put two of them together to create a bridge table. Figure 8-5 shows an example of how it works.

1 Choose mapping:

External Entity Operation ↔ Domain Operation

2 Identify required half tables

External Entity Operation (without parameters)

Domain	EE	Operation
Lubrication	SIO	Inject
Lubrication	SIO	Stop injecting

Domain Operation (with one parameter value)

Domain	Operation	Parameter	Value
SIO	Write point	Point ID	1
SIO	Write point	Point ID	0

3 Join together and fill out

We exclude the Point ID parameter in this table since it refers to an instance.

Domain	EE	Operation	Domain	Operation	Parameter	Value
Lubrication	SIO	Inject	SIO	Write point	Value	1
Lubrication	SIO	Stop injecting	SIO	Write point	Value	0

- ★ This table is populated at model time (before building or running the system).
- ★ It tells us which method and parameter value to use when an Inject/Stop injecting bridge operation is invoked.

Figure 8-5. Inject operation half tables

The first step is to map an external entity operation in the Lubrication domain to a domain operation in the SIO domain. For the moment, we won't consider how to identify the instances involved because, just now, we are concerned only with how the domain models connect together. We then construct the half table for each side. Joining the two halves together yields a bridge table. With the bridge table in hand, we can fill in the values on each side corresponding to the two domains. The result shows exactly how the semantics of Inject and Stop Injecting in the Lubrication domain are made manifest by writing specific values to an I/O Point.

But we're not done yet. The preceding table shows how the semantic gap is bridged, but operations happen on instances. We know that the *Write point* operation is called when the Lubrication domain invokes an *Inject* or *Stop injecting* operation. But the method must be invoked on some instance of I/O Point.

To determine the instances involved in the operations, a further mapping is required, this time between instances of Injector and instances of I/O Point. Again, we use half tables, as shown in Figure 8-6.

1 Choose mapping:

Instance  Instance

2 Identify required half tables (use the same one on each side)

Instance (with single attribute identifier)

Domain	Class	ID Attribute	ID Value



3 Join together and fill out...

Domain	Class	ID Attr	ID Value	Domain	Class	ID Attr	ID Value
Lubrication	Injector	ID	IN1	SIO	IO Point	ID	IOP4
Lubrication	Injector	ID	IN2	SIO	IO Point	ID	IOP5
Lubrication	Injector	ID	IN3	SIO	IO Point	ID	IOP6

- ★ ... at initialization for static populations
- ★ ... during runtime for dynamic populations
(requires additional mapping of create and delete actions across domains)

Figure 8-6. Inject instance half tables

We are making a correspondence between an instance of an Injector and an instance of an I/O Point. The Injector class has a single identifying attribute named ID, and the same is true of the I/O Point class. So the half tables, coincidentally, have the same column headings. To obtain the bridge table, we join the two instance half tables.

The instance populations of Injector and I/O Point are static during runtime. During the running of the system, we do not create or delete Injector instances, nor do we create or delete I/O Point instances. Consequently, we can fill in the table when we are ready to build the system. If either or both populations were dynamic, we would need to map create and delete actions in each domain to one another so that the values in the instance mapping table could be maintained during runtime.

There is one final detail about the bridge that must be specified. The *Inject* and *Stop injecting* external entity operations have a formal parameter named Injector whose value determines which Injector class instance is to be started or stopped. Similarly, the *Write point* operation in SIO has a formal parameter named I/O Point whose value determines to which I/O Point class instance the write operation pertains (in addition to the parameter that specifies the value to write). We must say where we get the argument values for these formal parameters. We want to say that the Injector parameter in the Lubrication operations represents an instance of the Injector class and that this maps to the I/O Point parameter that represents a

corresponding instance of the I/O Point class. Again, we specify a bridge table by using half tables, mapping an ID parameter in an external entity operation to an ID parameter in a domain operation. Figure 8-7 shows the resulting bridge table.



The diagram shows a double-headed arrow indicating a mapping between the 'External Entity ID Parameter' (in blue) and the 'Domain Operation ID Parameter' (in pink).

External entity operation ID parameter				Domain operation ID parameter				
Domain	EE	Operation	ID Param	Class	Domain	Operation	ID Param	Class
Lubrication	SIO	Inject	Injector	Injector	SIO	Write Point	IO Point	IO Point
Lubrication	SIO	Stop injecting	Injector	Injector	SIO	Write Point	IO Point	IO Point

Here we associate ID parameters with classes. The instance-to-instance mapping can then be consulted to look up an IO Point.ID value for the Point ID parameter based on the value of Injector provided by the Inject/Stop injecting operations.

Figure 8-7. Mapping ID parameters

These three bridge tables precisely define the way the semantic gap for starting and stopping injection between the Lubrication domain and the SIO domain is realized. Tracing through the bridge tables, we can see the complete logic of the bridge mapping.

- When the Inject external entity operation is invoked in the Lubrication domain, the inject operation bridge table, Figure 8-5, states that we must realize the Lubrication domain's intent by having the bridge code invoke the *Write point* domain operation in the SIO domain and that passing 1 as the Value argument in that operation corresponds to starting the injection.
- The ID parameter bridge table, Figure 8-7, states that the Injector parameter value, given as an argument in the Inject invocation, is an identifier of an Injector class instance and that the *Write point* operation requires the identifier of a corresponding I/O Point class instance.
- The instance bridge table, Figure 8-6, provides the correspondence between the value of an Injector identifier, as obtained from the Inject operation ID parameter and the value of an I/O Point identifier that is supplied to the *Write point* ID parameter.

In this way, the bridge can determine not only what domain operation in SIO satisfies the needs of the Lubrication domain, but precisely what values have to be passed to the *Write point* operation when it is invoked in the bridge code.

You may notice that the columns of the half tables always refer to generic model elements (operations, instances, and so forth). Mapping across domains involves the general idea of creating a correspondence between one kind of model element in a domain to another (or possibly the same) type of element in another domain.

After we build our tables, we are in an excellent position to move forward with our pycca implementation. Without them, things can get rather confusing. But before jumping into implementing the bridge, we show other bridge tables to demonstrate the variety of mappings that can arise, particularly when asynchronous operations such as event signaling are involved.

Start and Stop Monitoring Pressure

Figure 8-8 shows two asynchronous signals, *Start* and *Stop monitoring*, which are directed at the SIO external entity. They are asynchronous because Lubrication wants to signal SIO and expects feedback in the future, but needs to go on about its business in the meantime.

Asynchronous signals to an external entity.

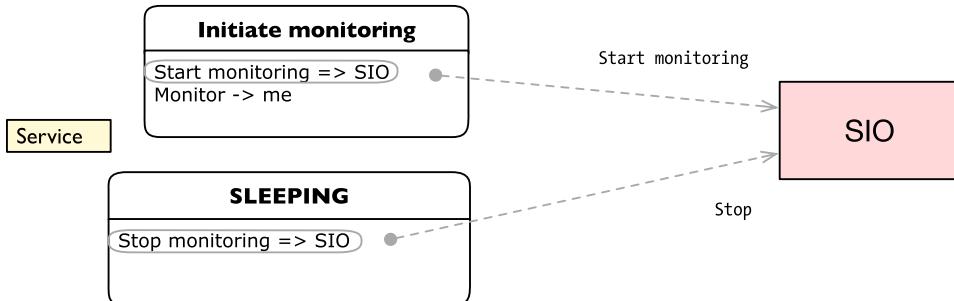


Figure 8-8. Signals to an external entity

It's a bit difficult to classify these as anything other than services we need, so we'll just mark them as *Service*. These signals correspond to the *Sample* and *Stop* events in SIO, as shown in Figure 8-9.

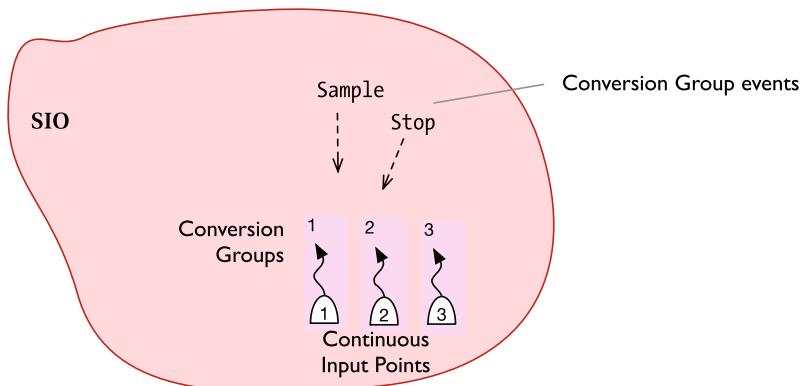
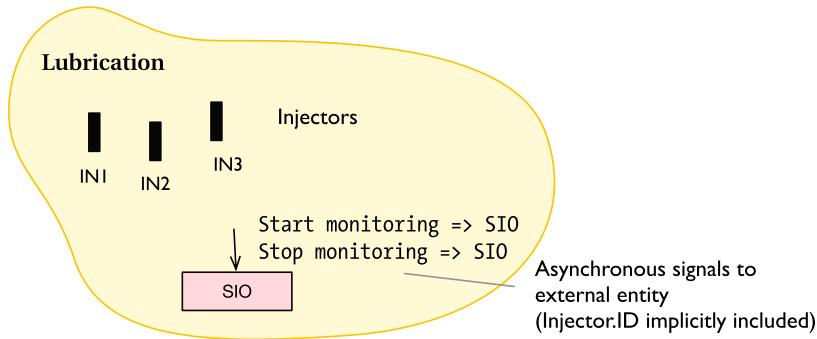
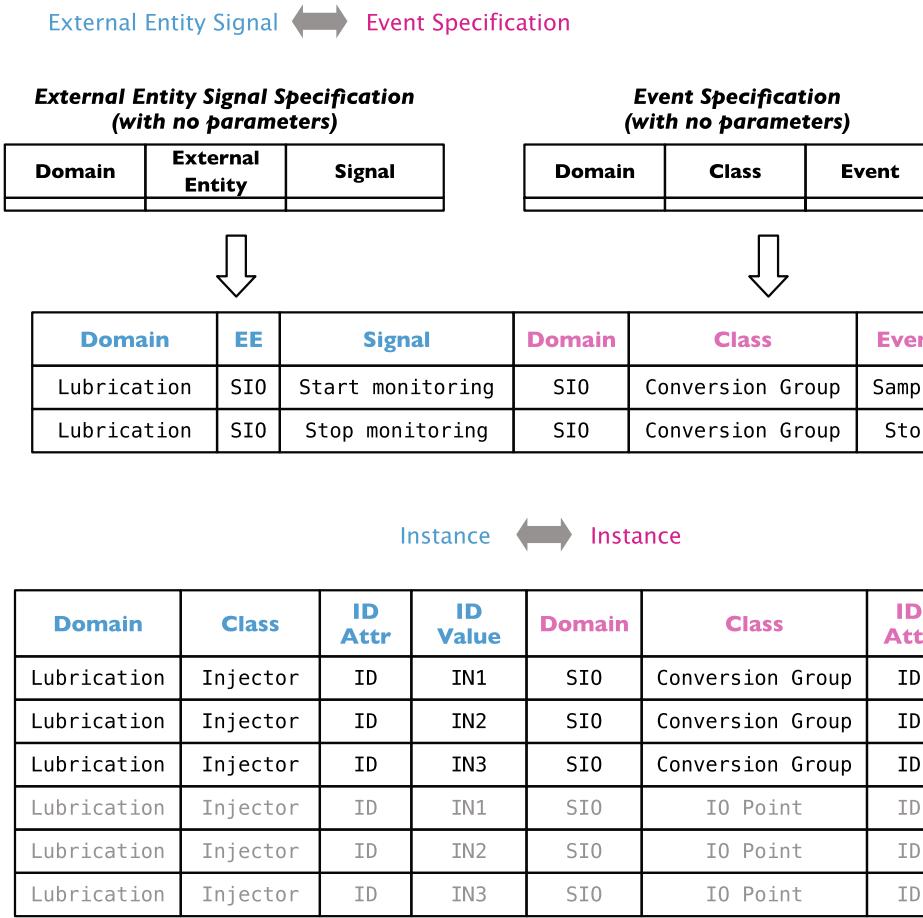


Figure 8-9. Events in SIO

In this case, we are able to leave the Injector.ID implicit in the sent signal. So neither signal carries an explicit ID parameter. It is always assumed that the source of a signal is available to the model execution architecture even if it isn't explicitly sent.

Now let's build the tables. Figure 8-10 shows the result.

Monitor start/stop half tables



Each Injector instance maps to an instance of Conversion Group.

Figure 8-10. Monitor start stop bridge tables

First we map an external entity signal specification to an *event specification*. We use the term *specification* here to be clear that we are not talking about a particular signal flying around during runtime, but rather the signature or specification of the signal/event structure.

In the second table, we map instances of Injector to SIO instances of Conversion Group. But we have already seen this table when we mapped instances for the injection control. The table heading is the same, so we can just add rows for the instance mapping from Injectors to Conversion Groups. We have now specified all the information necessary to work out which event to trigger in the SIO domain and to which instance it should be addressed.

Update Pressure

Bridging is not limited to actions. If we look at the Injector class, we see that there is one special attribute, Pressure. It is special in that the Lubrication domain assumes that it can read the attribute and obtain the current lubricant pressure of the actual physical Injector. This implies that SIO is delegated the responsibility to keep the pressure value up-to-date. In Figure 8-11, we mark it as a *sensor* attribute. In a domain such as Lubrication, many such attributes are often scattered around. The SIO notion of a sensor is realized as a Continuous Input Point. For each pressure value in Lubrication, there is a Continuous Input Point instance in SIO. After the raw device value has been converted and scaled to a meaningful pressure value, it must be made available somehow when an Injector reads its pressure. We'll get to the *somewhat* later in the chapter, but for now it is enough to say that the attribute values are mapped together.

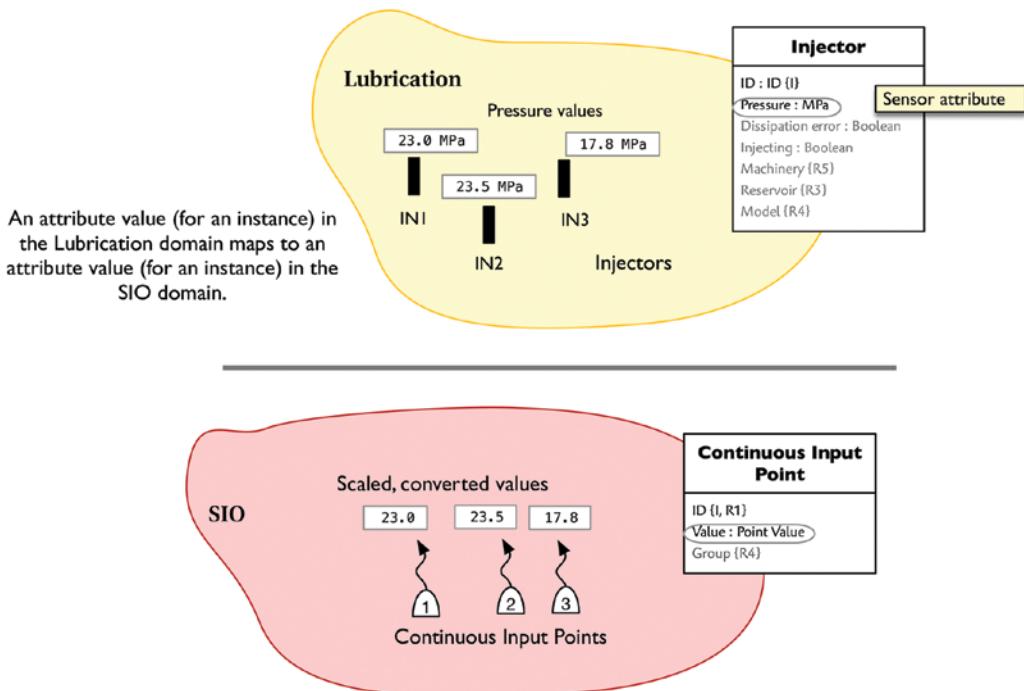


Figure 8-11. Marking a sensor attribute

Figure 8-12 shows the specification of the bridge tables.

The Injector.Pressure attribute maps to the Continuous Input Point.Value attribute.

attribute		
Domain	Class	Attribute

Attribute Attribute



Domain	Class	Attribute	Domain	Class	Attribute
Lubrication	Injector	Pressure	SIO	Continuous Input Point	Value

Instance Instance

Each Injector instance maps to a Continuous Input Point instance.

Domain	Class	ID Attr	ID Value	Domain	Class	ID Attr	ID Value
Lubrication	Injector	ID	IN1	SIO	Continuous Input Point	ID	IOP1
Lubrication	Injector	ID	IN2	SIO	Continuous Input Point	ID	IOP2
Lubrication	Injector	ID	IN3	SIO	Continuous Input Point	ID	IOP3
Lubrication	Injector	ID	IN1	SIO	Conversion Group	ID	CG1
Lubrication	Injector	ID	IN2	SIO	Conversion Group	ID	CG2
Lubrication	Injector	ID	IN3	SIO	Conversion Group	ID	CG3
Lubrication	Injector	ID	IN1	SIO	IO Point	ID	IOP4
Lubrication	Injector	ID	IN2	SIO	IO Point	ID	IOP5
Lubrication	Injector	ID	IN3	SIO	IO Point	ID	IOP6

Figure 8-12. Attribute mapping bridge tables

In the instance-to-instance table, we map the instances of Injector to corresponding instances of Continuous Input Point. This table also has the same heading as you have seen before. We can just extend that table with the new instance mappings for the Continuous Input Points associated to the injector lubricant pressure. Note that multiple SIO model elements are required to capture the services required by a single Injector. This is not an unusual circumstance.

The correlation between the Injector.Pressure attribute and the Continuous Input Point.Value is straightforward. If we had other sensor-based attributes, such as Temperature or Vibration, we would add rows to the table, filling in the appropriate class and attribute names on the left half, and filling in the appropriate Continuous Input Point ID Value on the right half.

We now know which data is mapped across the bridge, but we haven't yet made any implementation decisions about the mechanism used to transport the data. Are the values mapped to the same memory location? Are values pushed or pulled between domains? This can be determined when the bridge is implemented. From a model perspective, the requirement is that when an Injector.Pressure value is read, it must be up-to-date and reflect the actual lubricant pressure seen at the real-world injector. From a bridge perspective, we must ensure that the Injector.Pressure value is the same as its corresponding Continuous Input Point.Value attribute value. From an implementation perspective, there are numerous means, each with different performance trade-offs, that accomplish the goal.

Injector Pressure Alerts

This last example is a bit more complex, but we will use the same process. We mark elements of the Lubrication domain and then figure out how to map them onto corresponding elements of SIO.

During the period when injector pressure is monitored, the Lubrication domain expects SIO to notify an Injector if one of four conditions occurs:

1. The pressure is above the minimum required for injecting lubricant.
2. The pressure is below the minimum required for injecting lubricant.
3. The pressure is above the minimum allowed when dissipating (that is, when lubricant is not being injected).
4. The pressure is greater than the allowed maximum.

Injectors are characterized by their associated Injector Designs. Three attributes of Injector Design can be marked as *Threshold attributes*. These markings show the Lubrication domain's intent that the attributes be used as part of the required pressure alerts for the injector pressure. Our task is to realize this marking as elements of SIO. *Max delivery pressure*, *Max system pressure*, and *Max dissipation pressure* must correspond to instances of Point Threshold whose *Limit* value is equal to the Injector Design attribute values. Figure 8-13 shows the correspondence.

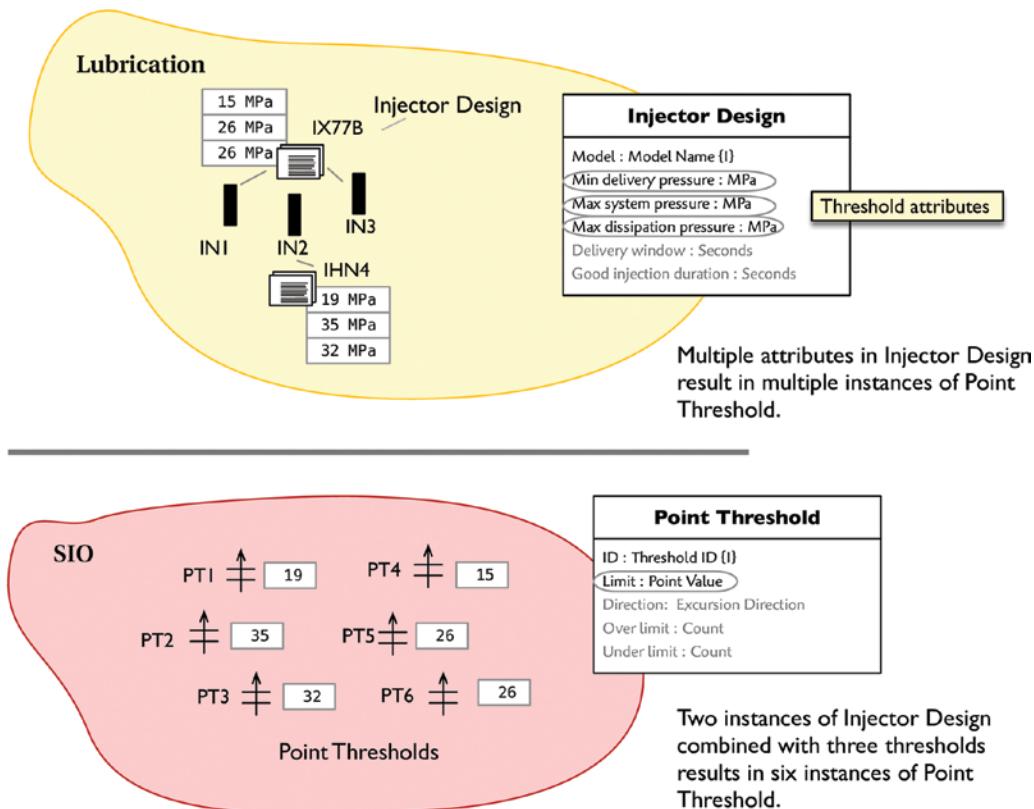


Figure 8-13. Marking threshold attributes

Looking again at the Injector model, we see that three events and a class method correspond to the preceding conditions, and these must be triggered by SIO at the appropriate times. We can think of these as being marked as *Alert events* and an *Alert method*.

The SIO domain provides the capability to compare values of Continuous Input Points against limits provided by a Point Threshold and determine whether a point value is in range or out of range with respect to the Point Threshold. This comparison operation is captured in the life cycle of the Range Limitation class. The association between Continuous Input Points and Point Thresholds, as mediated by the Range Limitation class, allows a point to be compared against multiple thresholds and a threshold to be applied to multiple points. So to configure the SIO domain to monitor for pressure alerts means that we must carefully populate the Point Threshold and Range Limitation instances to match the expectations of the Injectors. We already have seen, as part of updating the Injector.Pressure value, that three Continuous Input Points are populated in SIO that correspond to the injector pressure. The values of those points are compared against limits to decide whether there is a pressure alert. Figure 8-14 shows the correspondence between Lubrication alert events and SIO range limits.

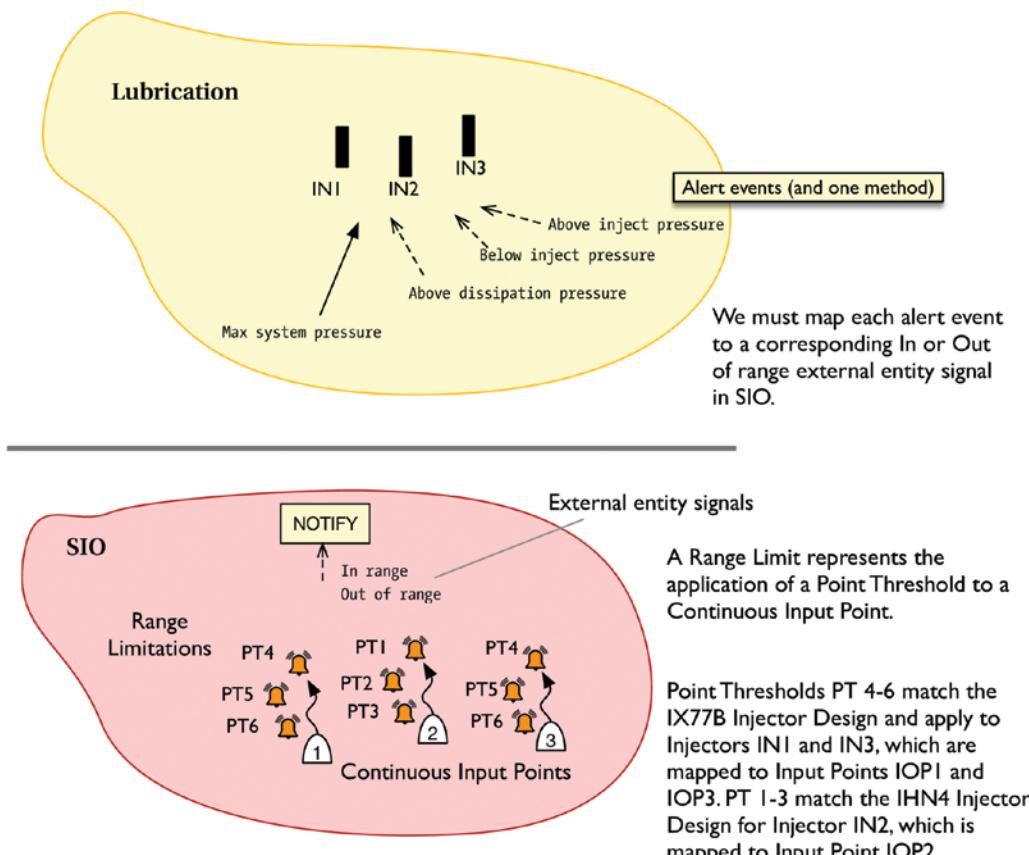


Figure 8-14. Marking alert events

Knowing which Continuous Input Points refer to injector pressure and knowing which Point Thresholds we need to apply to satisfy the pressure alerts, we can populate the instances of Range Limitation. Consider a single Injector, IN2, which happens to be a model IHN4 injector. Its related Injector Design specifies the following values:

- Min delivery pressure \Rightarrow 19 MPa
- Max system pressure \Rightarrow 35 MPa
- Max dissipation pressure \Rightarrow 32 MPa

This means that there are three corresponding instances of Point Threshold called PT1, PT2, and PT3. We already know from the bridge that updates the `Injector.Pressure` attribute, IN2 maps to the Continuous Input Point IOP2. Three instances of Range Limitation must be created in SIO to monitor IOP2, one for each Point Threshold that reflects the marked *Threshold attributes* of the Injector Design. These are identified in SIO as PT1-IOP2, PT2-IOP2, and PT3-IOP2. In total, we have nine instances of Range Limitation for the three Continuous Input Points, and the three distinct Point Thresholds corresponding to each Injector Design.

For each newly acquired point value, each associated instance of Range Limitation compares the value to its corresponding threshold limit. When the threshold is crossed, an *In range* or *Out of range* signal is sent to the NOTIFY external entity in SIO. The task is to ensure that this signal triggers the appropriate event or method in Lubrication and directs it to the correct instance of Injector. Because the Lubrication domain expects the alerts to be delivered as either a domain operation or as an event signaled to an Injector instance, we'll need two bridge tables to accomplish the mapping, as shown in Figure 8-15. One maps to the *Injector max pressure* domain operation, and the other maps to our alert events.

External entity signal and attribute value  Domain operation

External entity signal X attribute value					Domain operation	
Domain	EE	Signal	Parameter	Value	Domain	Operation
SIO	NOTIFY	Out of range	Threshold ID	PT5	Lubrication	Injector max pressure
SIO	NOTIFY	Out of range	Threshold ID	PT2	Lubrication	Injector max pressure

Domain	EE	Signal	Parameter	Value	Domain	Operation
SIO	NOTIFY	Out of range	Threshold ID	PT5	Lubrication	Injector max pressure
SIO	NOTIFY	Out of range	Threshold ID	PT2	Lubrication	Injector max pressure

External entity signal and attribute value  Event specification

External entity signal X attribute value					Event specification		
Domain	EE	Signal	Parameter	Value	Domain	Class	Event Spec
SIO	NOTIFY	Out of range	Threshold ID	PT4	Lubrication	Injector	Above inject pressure
SIO	NOTIFY	Out of range	Threshold ID	PT1	Lubrication	Injector	Above inject pressure
SIO	NOTIFY	In range	Threshold ID	PT4	Lubrication	Injector	Below inject pressure
SIO	NOTIFY	In range	Threshold ID	PT1	Lubrication	Injector	Below inject pressure
SIO	NOTIFY	Out of range	Threshold ID	PT6	Lubrication	Injector	Above dissipation pressure
SIO	NOTIFY	Out of range	Threshold ID	PT3	Lubrication	Injector	Above dissipation pressure

Domain	EE	Signal	Parameter	Value	Domain	Class	Event Spec
SIO	NOTIFY	Out of range	Threshold ID	PT4	Lubrication	Injector	Above inject pressure
SIO	NOTIFY	Out of range	Threshold ID	PT1	Lubrication	Injector	Above inject pressure
SIO	NOTIFY	In range	Threshold ID	PT4	Lubrication	Injector	Below inject pressure
SIO	NOTIFY	In range	Threshold ID	PT1	Lubrication	Injector	Below inject pressure
SIO	NOTIFY	Out of range	Threshold ID	PT6	Lubrication	Injector	Above dissipation pressure
SIO	NOTIFY	Out of range	Threshold ID	PT3	Lubrication	Injector	Above dissipation pressure

Figure 8-15. Mapping range limits

We are doing something a bit different in these tables. Because we are handling an external entity signal that originates in SIO, we are mapping from SIO to Lubrication this time. The table column order makes no difference, but it is a bit easier to explain from the direction of the signal to its destination. Also in this case, the half tables is a combination of two SIO model elements: an external entity signal and a parameter value. That's because any Continuous Input Point may have multiple thresholds defined for it, and we need a parameter to distinguish which threshold is being signaled. Each of the six thresholds can be signaled as either *In range* or *Out of range*. Notice also that of the twelve possible entries in the two tables, only eight are populated with values. This is because the Lubrication domain is not interested in the cases where the

maximum injector pressure limit goes *In range* nor when the dissipation pressure limit is *In range*. There are no alert events or alert methods for these cases. Consequently, when those external entity signals are generated, they are not found in the bridge tables, and no bridge action is taken. Going back to our example, consider when the value of Continuous Input Point, IOP2, exceeds the limit for Point Threshold, PT2. This would be detected by the PT2-IOP2 instance of Range Limitation. If the *Out of range* signal is sent to the NOTIFY external entity, and the Threshold ID is given as PT2, then our table shows that the bridge invokes the *Injector max pressure* domain operation. Alternatively, if the Threshold ID value had been PT1, consulting the first table would fail to find a match; but consulting the second table, we find that we must signal the *Above inject pressure* event to an Injector.

Proceeding as before, we must build half tables that correlate the Continuous Input Point instances in SIO to Injector instances in Lubrication. The Continuous Input Point identifiers here are the same ones that are used for updating the Injector.Pressure value. It is that same value against which the threshold limit is compared. Figure 8-16 shows the populated bridge table.

Instance ↔ Instance

Domain	Class	ID Attr	ID Value	Domain	Class	ID Attr	ID Value
SIO	Continuous Input Point	ID	IOP1	Lubrication	Injector	ID	IN1
SIO	Continuous Input Point	ID	IOP2	Lubrication	Injector	ID	IN2
SIO	Continuous Input Point	ID	IOP3	Lubrication	Injector	ID	IN3

Figure 8-16. Mapping input points

Finally, we must specify from where the ID parameters come. When we invoke the *Injector max pressure* domain operation, the external entity ID parameter is a Continuous Input Point ID, and the domain operation ID parameter is an Injector ID. Similarly, if we are signaling an alert event out of the bridge, the instance to which the signal is sent is an Injector. Figure 8-17 shows the identifier mappings.

External entity ID parameter ↔ Domain operation ID parameter

Domain	EE	Signal	ID Param	Class	Domain	Operation	ID Param	Class
SIO	NOTIFY	Out of range	Point ID	Continuous Input Point	Lubrication	Injector max pressure	Injector ID	Injector

External entity ID parameter ↔ Event specification ID attribute

Domain	EE	Signal	ID Param	Class	Domain	ID Param	Class
SIO	NOTIFY	Out of range	Point ID	Continuous Input Point	Lubrication	ID	Injector
SIO	NOTIFY	In range	Point ID	Continuous Input Point	Lubrication	ID	Injector

Figure 8-17. Mapping identifier parameters

This last example of constructing the bridge tables is certainly more complicated than the others. It requires populating several different classes in SIO to configure it appropriately to meet the Lubrication domain's needs. Once that is done, the bridge tables are built, following the techniques we have shown. It serves as a good reminder that bridging is an abstract undertaking, requiring that we make correspondences between multiple domains; we need to apply the marking and mapping techniques diligently to avoid getting tangled up.

Implementing Bridges in Pyccca

In the previous section, we showed how to bridge the assumptions and dependencies of one domain onto the services provided by another domain. In this section, we turn our attention to creating the code necessary to implement the bridge between the domains.

In our example, the Lubrication domain uses the SIO external entity as a proxy to express how it has delegated actions. It invokes operations on the SIO external entity at the correct point in its processing when those actions need to take place. We distinguished between external entity operations as synchronous and asynchronous. This distinction is important to the Lubrication domain, as it expresses the domain's expectation for fulfillment of the service. When the *Inject* or *Stop injecting* operations are invoked, the Lubrication domain assumes that, however it may happen, by the time the operation has completed, the injector solenoids have been engaged or disengaged from applying lubricant. Contrast that with the assumptions the Lubrication domain has when the *Start monitoring* and *Stop monitoring* operations are invoked. In that case, the Lubrication domain assumes that it can continue with its processing, and events will arrive later to indicate the results of monitoring the injector pressure.

For pyccca-generated domains, each external entity operation, whether synchronous or asynchronous in its nature, is converted into an external declaration of an ordinary C function. When the operation is invoked, it executes as any other ordinary C function, potentially accepting arguments and returning a value. The model-level concepts of synchronous vs. asynchronous become lost in the conversion to an ordinary function but are still present in the interactions of the bridge.

To bridge pyccca domains, it is necessary to supply a definition for the external entity functions. Pyccca uses a naming convention for the external entity functions:

```
eop_<domain name>_<external operation name>
```

- <domain name> is the name of the domain in which the external entity was defined.
- <external operation name> is the name given to the external entity operation when it was declared by the external operation declaration in the pyccca source.

The bridge implementation consists of a function, named as just described, for each of the external entity operations defined by a domain. Then, when the domain code files along with the bridge code files are compiled and linked, all the external symbol references are resolved, and we obtain a runnable program.

Pyccca Facilities for Implementing Bridge Code

In this section, we discuss the help that pyccca provides in constructing bridges.

The bridging code for pyccca-generated domains is manually written. We do not have any great expectations that bridge code can be reused in the same way that we wish to reuse domains. Bridges are specific to the usage of particular domains, populated with specific instances for a given application context. As you saw previously, the population of the bridge tables depends on the initial population of class instances in both Lubrication and SIO. For our example, the data values of SIO attributes have been chosen specifically to match corresponding attribute values in Lubrication. Reusing the SIO domain in a different

application requires a different instance population and different bridge tables. Because we expect little reuse of bridge code, manually writing such code is the most direct way to glue domains together. Although there is no support in pycca for generating the bridge code itself, pycca does provide other essential support.

The Domain Portal

As we saw previously in the bridge tables that map an external entity operation onto model elements of a service domain, there are several types of actions we might wish to perform in the service domain. Sometimes it is as simple as invoking a domain operation on the service domain. Other times, we may need to signal an event to a class instance or update an attribute value in the service domain.

When pycca generates a domain, the only symbols visible outside the domain are the domain operation names. For each `domain operation` statement, pycca generates an ordinary C function of external scope whose name is of the form `<do-main name>_<domain operation name>`. All other C identifiers in the generated code file have file static scope, and so their symbols are *not* available outside the code file for the domain. The domain is well encapsulated, and there is little probability of a symbol name conflict when linking multiple domains into an application. Typically, the external entity functions for a domain are placed in a separate C source file, but other organizations are possible. For simple cases, such as ours, we can place the external entity functions for both domains in a single source file.

To help create the external entity functions, pycca supplies a means to accomplish simpler, common model-level actions from outside the domain. Upon request, pycca creates a *portal* into the domain. The portal consists of an initialized variable along with a set of C functions that operate over the data values in the portal variable to perform common model-level operations. The name of the portal variable follows the convention of `<domain name>_portal`, and this name is also of external scope. The C functions of the portal code support the following operations:

- Creating and destroying class instances
- Reading and updating instance attribute values
- Signaling events
- Signaling and canceling delayed events

The ability to access attributes and signal events eliminates the need to create domain operations that perform only simple, common, model-level operations. Any bridge operation that can be accomplished with the portal functions does not require the domain to provide a domain operation. Domains must provide domain operations when the required model-level operations are more complicated, such as traversing relationships or searching for instances based on attribute values.

The pycca approach is a compromise between opening the domain to arbitrary processing from outside and having each model-level operation required by a bridging operation implemented as a domain operation. The portal breaks the encapsulation of a domain, but in a limited way. In practice, the compromise works well for many cases and achieves the goal of enhancing domain reuse by minimizing the number of explicit domain operations that might have to be created when a domain is used in a particular application context.

Identifying Domain Elements

When pycca is requested to generate a portal for the domain, it also includes in the generated header file for the domain a set of C preprocessor *define* statements that constitute a numerical encoding for the model-level elements of the domain. The definitions include the following:

- A number to uniquely identify each class in the domain
- The total number of classes in the domain

- A number to uniquely identify each attribute within a class
- The total number of attributes of a class
- A number to uniquely identify each instance within a class
- The total number of instances of a class
- The numerical encoding of the state model events for a class

This encoding gives symbolic names to a set of ordinary integers. The values are used as arguments to the portal functions to specify which class or attribute is intended and to identify any instances that are part of the initial instance population. For example, to send an event to an instance, you must supply the class number, instance number, and event number to the portal function. This is done with the *define* statements supplied by pycca in the generated header file for the domain. We show how these symbols are used in implementing bridge mappings and how the portal itself operates.

Lubrication Domain External Entity Functions

Four external entity functions are defined by the Lubrication domain. Two control the lubricant injection, and two monitor injector lubricant pressure.

Implementing Injection Control Functions

We now show the implementation of the external entity functions for starting and stopping lubricant injection. Figure 8-6 showed the bridge tables that map between Lubrication domain Injector instances and SIO domain I/O Point instances. We could design the external entity function by directly encoding the table as shown. Looking closely, however, we see that the only two columns that vary are the ID Value on the Lubrication domain side and the ID Value on the SIO domain side. The instance bridge table maps a Lubrication domain Injector ID attribute value to an SIO I/O Point ID attribute value. This suggests an optimization that yields a smaller table. We need only implement a mapping for the varying parts of the bridge tables. We further observe that every Injector instance in the Lubrication domain appears in the bridge table. This means that we can implement the bridge table search as a simple array-indexing operation using the numerical encoding of instances supplied by pycca. The pycca encoding for instance identifiers consists of zero-based consecutive integers that are intended to be usable as array indices. The table to map Injectors to I/O Points that control the injector solenoids can be reduced to the following:

```
static sio_Point_ID const injToPointMap[LUBE_INJECTOR_INST_COUNT] = {
    [LUBE_INJECTOR_IN1_INST_ID] = SIO_IO_POINT_IOP1_INST_ID,      // ❶
    [LUBE_INJECTOR_IN2_INST_ID] = SIO_IO_POINT_IOP2_INST_ID,
    [LUBE_INJECTOR_IN3_INST_ID] = SIO_IO_POINT_IOP3_INST_ID,
};
```

❶ Note the use of C99 *designated initializers* in the array definition. This allows us to use the symbolic name of the injector instance number without regard for its value, and the ordering of the elements of the array in memory are placed correctly by the compiler. Along with making the mapping clearer, this technique avoids many potential errors, especially because the values for the index symbols are automatically generated.

So, the bridge table from Figure 8-6 is *implemented* as an initialized C array, indexed by the Injector instance identifier generated by pycca and whose element values are the SIO I/O Point instance numbers corresponding to the Discrete Points controlling the injector solenoids. Notice that the symbol names generated by pycca contain enough additional information to ensure that the symbols do not conflict with those from other domains. Also notice that we have chosen names for the initial instances that reflect their usage in the application and this text. Careful name selection can help avoid confusion when putting together bridge-mapping data.

Figure 8-5 showed the bridge table that maps the *Inject* and *Stop injecting* external entity operation from the Lubrication domain onto the *Write point* domain operation of the SIO domain. Again, if we examine the table closely, we see that the only two columns that vary are Operation on the Lubrication domain side and Value on the SIO domain side. This suggests another optimization. We implement the bridge mapping as a C function that performs the instance mapping and parameterizes the I/O Point value depending on whether we are starting or stopping the injection:

```
static void
controlInjector(
    InstId_t injectorId,
    bool starting)
{
    assert(injectorId < LUBE_INJECTOR_INST_COUNT) ;
    sio_Write_point(injToPointMap[injectorId], starting ? 1 : 0) ;
}
```

This function maps an Injector instance to an I/O Point instance, using the injToPointMap array, and invokes the SIO domain operation to write a value to the point. The starting argument is true if we want to start injecting, and false otherwise. Starting and stopping are mapped to 1 and 0, respectively, as the required values of the Discrete Point to start and stop lubricant injection.

The external entity functions need only invoke the common controlInjector function with the appropriate value for the starting argument:

```
void
eop_lube_SIO.Inject(
    InstId_t injectorId)
{
    controlInjector(injectorId, true) ;
}

void
eop_lube_SIO.Stop_injecting(
    InstId_t injectorId)
{
    controlInjector(injectorId, false) ;
}
```

Once distilled down to its essentials, the bridge to implement starting and stopping injection is quite small. However, we want to emphasize that the path to this small implementation starts with designing the bridge, using the ideas discussed in the first part of this chapter. Larger and more complicated bridges will require larger and more complicated external entity function implementations, but those functions can be derived using the design process we have shown.

Implementing Injector Pressure Monitoring Functions

The two other external entity functions for the Lubrication domain involve monitoring the injection pressure. The Lubrication domain invokes *Start monitoring* when it has decided that the injection lubricant pressure needs to be monitored. It invokes *Stop monitoring* when the Injector is at a point in its life cycle where monitoring is no longer needed. The design of the bridge is shown in Figure 8-10. In this case, the Lubrication domain maps an asynchronous signaling operation onto signaling an event to an instance in the SIO domain. The instance mapping is from Injector instances in the Lubrication domain to Conversion Group instances in the SIO domain.

Using the same reasoning as in the previous external entity function, we can construct an initialized C array variable to perform the instance mapping:

```
static sio_Point_ID const presToConvGrpMap[LUBE_INJECTOR_INST_COUNT] = {
    [LUBE_INJECTOR_IN1_INST_ID] = SIO_CONVERSION_GROUP_INJ1(CG_INST_ID),
    [LUBE_INJECTOR_IN2_INST_ID] = SIO_CONVERSION_GROUP_INJ2(CG_INST_ID),
    [LUBE_INJECTOR_IN3_INST_ID] = SIO_CONVERSION_GROUP_INJ3(CG_INST_ID),
};
```

Again, because the set of Injector instances in the implementation of the mapping bridge table is the complete set of instances for the domain, we can perform the search for the corresponding Conversion Group by using array indexing.

Examining the bridge table in Figure 8-10 that maps the external entity signal to an event, we see that the arrangement is similar to that for starting and stopping injection. In this case, the varying parts of the bridge table are the Signal column from the Lubrication domain and the Event column from the SIO domain. We exploit this arrangement by constructing a function where the SIO event, encoded as a number by pycca, is a parameter:

```
static void
signalConversionGroup(
    InstId_t injectorId,
    EventCode event)
{
    assert(injectorId < LUBE_INJECTOR_INST_COUNT) ;

    int pcode = pycca_generate_event(&sio_portal,
        SIO_CONVERSION_GROUP_CLASS_ID,
        presToConvGrpMap[injectorId],
        NormalEvent,
        event,
        NULL) ;

    assert(pcode == 0) ;
    (void)pcode ; // ①
}
```

① This eliminates compiler warnings about pcode being unused when the assertions are removed.

The controlConversionGroup function uses the pycca portal function `pycca_generate_event` to signal an event to a Conversion Group instance from outside the SIO domain. The arguments to `pycca_generate_event` are as follows:

- `sio_portal`, which tells the function which domain is involved.
- `SIO_CONVERSION_GROUP_CLASS_ID`, which gives the class of the signaled instance. This symbol is generated by pycca.
- The Conversion Group instance number obtained from the instance mapping array shown previously. The initializer values in this array are symbols generated by pycca.
- The type of the event to signal.
- The number of the event to signal, passed in as a parameter.
- A pointer to the event parameters. In this case, there are none.

Using the preceding function, we can code the Lubrication domain external entity functions for starting and stopping pressure monitoring.

```
void
eop_lube_SIO_Start_monitoring(
    InstId_t injectorId)
{
    signalConversionGroup(injectorId, SIO_CONVERSION_GROUP_SAMPLE_EVENT_ID) ;
}

void
eop_lube_SIO_Stop_monitoring(
    InstId_t injectorId)
{
    signalConversionGroup(injectorId, SIO_CONVERSION_GROUP_STOP_EVENT_ID) ;
}
```

SIO Domain External Entity Function

In this section, we show the external entity functions for the SIO domain. We show only four of the functions here. The remaining function not covered here is shown in the online materials for the book. Its implementation pattern is similar to ones you have already seen.

Updating Injector Pressure Attribute

After the Lubrication domain has invoked *Start monitoring*, its expectation is that the value of the Pressure attribute of the Injector instances always contains the latest measured value. You saw how monitoring injector pressure was mapped to an event to an instance of Conversion Group in the SIO domain. Each time a Continuous Input Point is sampled, it is scaled to engineering units, and the *New point value* operation of the NOTIFY external entity is invoked. The asynchronous nature of the bridge from the Lubrication domain perspective now shows up as an external entity operation invoked by the SIO domain.

The external entity function for *New point value* is responsible for updating, in the Lubrication domain, the value of the Pressure attribute for the Injector instance corresponding to the Continuous Input Point. Figure 8-12 shows the bridge table. From the point of view of the bridge, Figure 8-12 extends the bridge table used in other parts of the bridge. However, from the point of view of our *implementation*, we are unable to reuse the instance mapping created for starting and stopping injection. This is because we simplified the instance mapping bridge table to exclude parts that did not vary when the mapping was used for controlling injection. That “optimization” allowed us to use array indexing as the mapping search mechanism, but

prevents us from using the mapping array for any other external entity functions. It is a trade-off we accept to simplify the instance-mapping search, but it requires us to build a different instance-mapping implementation for this external entity function.

Unlike our previous maps, here we are mapping from I/O Point in the SIO domain onto Injectors in the Lubrication domain. There may many more I/O Points in the SIO domain than there are Injectors in the Lubrication domain, so our previous strategy of using a simple array index doesn't work well. For this mapping, we choose to build and search a table that explicitly maps I/O Point identifiers corresponding to the Injector pressure values to the Injector instances' IDs in the Lubrication domain.

To accomplish the mapping, we need a data structure to hold the necessary data:

```
typedef struct {
    InstId_t fromInst ;
    InstId_t toInst ;
} BridgeIDMap ;
```

Given an array of such structures, we code a simple linear search to find the corresponding ID value:

```
static BridgeIDMap const *
mapIOPoint(
    BridgeIDMap const *mapping,
    int numMappings,
    InstId_t from)
{
    for ( ; numMappings > 0 ; numMappings--, mapping++) {
        if (mapping->fromInst == from) {
            return mapping ;
        }
    }

    return NULL ; // ①
}
```

① We use NULL to indicate that the mapping failed.

We have chosen a simple search implementation. Our instance populations are small, and so our mapping tables are also small. Larger populations would warrant a more sophisticated search technique such as a hash table.

The mapping from I/O Point instances that correspond to Injector pressure values to the Lubrication domain Injector instances is held in an initialized array variable:

```
static BridgeIDMap const presToInjMap[LUBE_INJECTOR_INST_COUNT] = {
    {
        .fromInst = SIO_IO_POINT_IOP1_INST_ID,
        .toInst = LUBE_INJECTOR_IN1_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP2_INST_ID,
        .toInst = LUBE_INJECTOR_IN2_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP3_INST_ID,
```

```

        .toInst = LUBE_INJECTOR_IN3_INST_ID,
    },
} ;

```

Because we are searching the array for the identifier of an I/O Point, the order of the array elements is arbitrary.

The code for the SIO external entity function uses a portal function to update the Injector pressure attribute in the Lubrication domain:

```

<<sio external operations>>= void
eop_sio_NOTIFY_New_point_value(
    sio_Point_ID point,
    sio_Point_Value value)
{
    assert(point < SIO_IO_POINT_INST_COUNT) ;

    BridgeIDMap const *pointMap =
        mapIOPoint(presToInjMap, COUNTOF(presToInjMap), point) ; // ❶

    assert(pointMap != NULL) ; // ❷
    if (pointMap == NULL) {
        return ;
    }

    int pcode = pycca_update_attr(&lube_portal,
        LUBE_INJECTOR_CLASS_ID,
        pointMap->toInst,
        LUBE_INJECTOR_PRESSURE_ATTR_ID,
        &value,
        sizeof(value)) ;

    assert(pcode > 0) ;
    (void)pcode ;
}

```

❶ The COUNTOF macro computes the number of elements in its argument array.

❷ We use assert to catch bad mappings during development. There should be none. However, when the assertions are gone in a release build, we have decided to protect ourselves against dereferencing the NULL pointer. Just because we expect no failures in the mapping does not mean one won't happen in reality.

❸ This portal function updates the value of an attribute. In this case, the attribute is the Pressure for an Injector instance in the Lube domain. Note the use of pycca-generated symbols as argument values to the portal function.

You can think of this bridge implementation as a *push* strategy for transporting the injector pressure values across to the Lubrication domain. The Pressure attribute of each Injector instance is updated at the same frequency that the corresponding Conversion Group is sampled. So, the activities of the Lubrication domain always pick up the latest pressure value when it is needed. You can contrast this technique with a *pull* technique, in which the pressure values for the Injectors would be read from the Signal IO domain when needed.

Signaling Pressure Alerts

The second part of the lubricant pressure monitoring bridge is to alert an Injector in the Lubrication domain when the lubricant pressure falls outside specified range boundaries. When SIO obtains a newly sampled value for a Continuous Input Point, it checks to see whether that point has one or more Range Limitations associated with it. The Range Limitation instances compare the point value against its Point Threshold limit and invoke the external entity operations of *Out of range* or *In range*, depending on the outcome of the comparison.

Figure 8-15 shows the bridge table. To external entity functions must be supplied. The bridge mapping of the external entity signal has one of three outcomes:

- A class method is invoked on an Injector.
- An event is signaled to an Injector.
- The threshold notification is not needed by Lubrication, and nothing happens.

The choice of three outcomes adds complexity to the external entity function implementations, but following our previous pattern, we factor out a small piece of code to perform the signaling operation. We use a pycca portal function and take the injector instance and event number as parameters:

```
static void
signalInjector(
    InstId_t injectorId,
    EventCode event)
{
    assert(injectorId < LUBE_INJECTOR_INST_COUNT) ;

    int pcode = pycca_generate_event(&lube_portal,
        LUBE_INJECTOR_CLASS_ID,
        injectorId,
        NormalEvent,
        event,
        NULL) ;

    assert(pcode == 0) ;
    (void)pcode ;
}
```

The external entity function to notify Injectors when a pressure threshold is out of range tests the threshold ID to determine the manner of notification:

```
void
eop_sio_NOTIFY_Out_of_range(
    sio_Point_ID point,
    sio_Threshold_ID threshold)
{
    assert(point < SIO_IO_POINT_INST_COUNT) ;
    assert(threshold < SIO_POINT_THRESHOLD_INST_COUNT) ;

    BridgeIDMap const *pointMap = mapIOPoint(presToInjMap, COUNTOF(presToInjMap), point) ;

    assert(pointMap != NULL) ;
    if (pointMap == NULL) {
        return ;
    }
```

```

switch (threshold) {
case SIO_POINT_THRESHOLD_IX77B_ABOVE_INJ_INST_ID:// fall through
case SIO_POINT_THRESHOLD_IHN4_ABOVE_INJ_INST_ID:
    signalInjector(pointMap->toInst,
                  LUBE_INJECTOR_ABOVE_INJECT_PRESSURE_EVENT_ID) ;
    break ;

case SIO_POINT_THRESHOLD_IX77B_ABOVE_DISP_INST_ID:// fall through
case SIO_POINT_THRESHOLD_IHN4_ABOVE_DISP_INST_ID:
    signalInjector(pointMap->toInst,
                  LUBE_INJECTOR_ABOVE_DISSIPATION_PRESSURE_EVENT_ID) ;
    break ;

case SIO_POINT_THRESHOLD_IX77B_MAX_PRES_INST_ID:// fall through
case SIO_POINT_THRESHOLD_IHN4_MAX_PRES_INST_ID:
    lube.Injector_max_pressure(pointMap->toInst)    ;
    break ;

/*
 * N.B. no default case.
 * Unexpected Range Limitation instances are silently ignored.
 */
}

}

```

Like the preceding external entity functions, the *In range* external entity function also tests the threshold ID. Because there is no *Below dissipation pressure* event and no *Below max pressure* method for an Injector, some of the Range Limitation threshold values have no mapping in the bridge. The only threshold for which *In range* notifications are meaningful is the lubrication injection pressure:

```

void
eop_sio_NOTIFY_In_range(
    sio_Point_ID point,
    sio_Threshold_ID threshold)
{
    assert(point < SIO_IO_POINT_INST_COUNT) ;
    assert(threshold < SIO_POINT_THRESHOLD_INST_COUNT) ;

    BridgeIDMap const *pointMap = mapIOPoint(presToInjMap, COUNTOF(presToInjMap), point) ;

    assert(pointMap != NULL) ;
    if (pointMap == NULL) {
        return ;
    }

    switch (threshold) {
case SIO_POINT_THRESHOLD_IX77B_ABOVE_INJ_INST_ID:// fall through
case SIO_POINT_THRESHOLD_IHN4_ABOVE_INJ_INST_ID:

```

```

    signalInjector(pointMap->toInst,
        LUBE_INJECTOR_BELOW_INJECT_PRESSURE_EVENT_ID) ; // ❶
    break ;
}

/*
 * N.B. no default case.
 * Unexpected Range Limitation instances are silently ignored.
 */
}
}

```

❶ Notice that we signal *Below inject pressure* when the minimum injection pressure threshold returns to being *In range*.

Both of these external entity functions could have been implemented differently. The implementation given is convenient for a small number of Point Threshold instances and makes clear which thresholds cause which action, either signaling an event or invoking a domain operation. But say there were 100 Point Threshold instances. A switch statement implementation would not scale well, and we would probably choose to encode the Point Threshold mapping in data and write code to search the data for the correct action to perform. Again we stress that the logic of the bridge and its mapping of semantics between domains stands apart from the implementation of the bridge code itself.

How the Portal Works

We have shown how the Lubrication and SIO domains can be bridged together by supplying code for the external entity functions of each domain. The code for the external entity functions mapped class instances between the domains and invoked either a domain operation or a pyccca portal function to fulfill the expectations of the bridge. What remains is to show how the pyccca portal functions themselves work.

The portal functions are data driven by the values contained in the portal variable generated by pyccca. The portal variable has the following type:

```

struct pyccca_domain_portal {
    struct pyccca_class_portal const *classes ;
    ClassId_t numClasses ;
} ;

```

The portal variable contains a pointer to an array of class descriptive information and a count providing the number of elements in the class description array. For the Lubrication domain, the portal variable generated by pyccca is as follows:

```

struct pyccca_domain_portal const lube_portal = {
    .classes = lube_class_portal,
    .numClasses = 6
} ;

```

As we stated before, pyccca also emits a series of preprocessor *defines* that encode, as consecutive zero-based integers, a set of identifiers for the instances of a class. These identifiers can be used directly as an array index into the class instance storage array. Pyccca also generates definitions that serve as identifiers for the classes and attributes of the domain. All of these preprocessor symbols follow naming conventions to make the symbol values unique. You saw examples of the symbol names in the preceding code. The following data types are also placed in the generated header file of the domain:

```
typedef unsigned short ClassId_t ;
typedef unsigned short InstId_t ;
typedef unsigned short AttrId_t ;
typedef unsigned short AttrOffset_t ;
typedef unsigned short AttrSize_t ;
```

The class information for the portal has the following structure:

```
struct pycca_class_portal {
    void *storage ;
    struct pycca_attr_portal const *attrs ;
    struct mechclass const *mechClass ;
    AttrId_t numAttrs ;
    InstId_t numInsts ;
    size_t instSize ;
    size_t instOffset ;
    bool isConst ;
    bool hasCommon ;
    StateCode initialState ;
} ;
```

The members of the portal class description are as follows:

<code>storage</code>	A pointer to the storage array for the class instances.
<code>attrs</code>	A pointer to an array of attribute descriptors.
<code>mechClass</code>	A pointer to the descriptor for the class.
<code>numAttrs</code>	The number of attributes of the class. This is the number of elements pointed to by the <code>attrs</code> member.
<code>numInst</code>	The maximum number of instances of the class.
<code>instSize</code>	The number of bytes occupied by an instance of the class.
<code>instOffset</code>	The offset in bytes from the beginning of the storage for an instance, where the instance data begins. This member is nonzero only for union-based superclasses. Because the subclass is stored directly contained within the superclass for union-based generalizations, this value gives the offset to the subclass storage.
<code>isConst</code>	A Boolean value that indicates whether the instance storage has been placed in read-only memory.
<code>hasCommon</code>	A Boolean value that indicates whether the class structure definition contains the <code>common_member</code> . For classes with static instance populations and no state model, pycca does not define the <code>common_member</code> for the class structure and saves the memory that would otherwise be filled with NULL pointers.
<code>initialState</code>	The state number for the initial state of an instance. For classes that do not have a state model, the value is ignored.

We show part of the class descriptive information, in this case for the Injector class:

```
static struct pycca_class_portal const lube_class_portal[] = {
    // .... omitted class descriptive information

{
    .storage = Injector_storage,
    .attrs = Injector_attr_portal,
    .mechClass = &Injector_class,
    .numAttrs = 3,
    .numInsts = 3,
    .instSize = sizeof(struct Injector),
    .instOffset = 0,
    .isConst = 0,
    .hasCommon = 1,
    .initialState = Injector_INITIAL_STATE
},
    // .... omitted class descriptive information
} ;
```

Each attribute of the class is described by the following data structure:

```
struct pycca_attr_portal {
    AttrOffset_t offset ;
    AttrSize_t size ;
} ;
```

The portal functions work primarily by using the pycca-generated symbols as indices into arrays generated by pycca when the portal is requested. The data elements in the arrays contain pointer values to internal domain data that can be passed as arguments to the runtime code. This arrangement provides a level of indirection to prevent exposing the domain internals and provides some control over the operations that may be performed on a domain externally. Because this is C, such protections are not firmly enforced by the compiler, and knowing the structure of the portal data means that it can be abused. We assume some level of good intentions and professionalism.

We show one example of a portal function, `pycca_update_attr`. All portal functions return an `int` value. A return value less than zero indicates an error, whereas a return value greater than or equal to zero indicates success. The `pycca_update_attr` function updates the value of an attribute in the given instance of the given class. Here, a successful return value indicates the number of bytes copied into the attribute:

```
int
pycca_update_attr(
    struct pycca_domain_portal const *portal,
    ClassId_t class,
    InstId_t inst,
    AttrId_t attr,
    void const *src,
    AttrSize_t size)
{
    int result ;
```

```

if (class < portal->numClasses) {
    struct pycca_class_portal const *classes = portal->classes + class ;      // ①
    if (inst < classes->numInsts) {
        MechInstance instRef = (MechInstance)((char *)classes->storage +
            classes->instSize * inst + classes->instOffset) ;                  // ②
        if (attr < classes->numAttrs) {
            struct pycca_attr_portal const *attrs = classes->attrs + attr ;   // ③
            if (classes->hasCommon == false || instRef->alloc != 0) {           // ④
                if (!classes->isConst) {
                    void *dst = (void *)((char *)instRef + attrs->offset) ;
                    result = size < attrs->size ? size : attrs->size ;          // ⑤
                    memcpy(dst, src, result) ;
                } else {
                    result = PYCCA_PORTAL_NO_UPDATE ;
                }
            } else {
                result = PYCCA_PORTAL_UNALLOC ;
            }
        } else {
            result = PYCCA_PORTAL_NO_ATTR ;
        }
    } else {
        result = PYCCA_PORTAL_NO_INST ;
    }
} else {
    result = PYCCA_PORTAL_NO_CLASS ;
}

return result ;
}

```

① Index to the class descriptor.

② Compute a pointer to the class instance.

③ Index into the attribute descriptor.

④ Check whether the instance storage slot is in use.

⑤ Check whether we are trying to write to constant memory.

⑥ Copy only the lesser of the size of the attribute storage and the size of the updated value storage. Note that this works properly only for processors that store multibyte quantities in little endian order.

Most of the code is spent validating input arguments in a way that allows us to give specific error codes on failure. The successful path through the code uses the `class`, `inst`, and `attr` arguments as indices into the description data contained in the `portal` variable. Finally, the value is copied into the proper memory location.

The other portal functions follow a similar pattern. Argument values are validated and used as indices into descriptive data. The descriptive data is used to compute a pointer to a class instance. Given an instance pointer, the usual runtime functions may be invoked on it.

Summary

In this chapter, we showed how to bridge the semantic gap between two domains by using our example Lubrication and SIO domains. Bridges were specified through marking and mapping. Marking designates domain elements in the client that have a correspondence in a service domain. Marking is also used to identify and populate class instances in the service domain. The service domain instance population is configured to meet the requirements of its client. Mapping makes the correspondence between domain elements explicit by recording data in bridge tables. Bridge tables are composed of half tables. Each domain contributes columns to its half table to describe the elements of the domain that are involved in the bridge. By joining half tables, we generate a bridge table that is then populated with rows for the instances and parameters having a correspondence in the bridge.

The bridge code is implemented by manually writing code for each external entity function. Pycca supplies a portal into a domain that is used in the external entity functions to perform common fundamental model-level operations such as signaling an event. Pycca also supplies a symbolic encoding of model-level elements such as classes, attributes, and instances. These symbols are useful in coding the external entity functions and are used as arguments to the portal functions. Finally, we showed how the portal and its functions work. The portal consists of an initialized variable and a set of C functions. The structure of the portal variable was shown along with some of the initializers for the Lubrication domain. Finally, we showed the C code for the portal function that updates the value of an attribute.

CHAPTER 9



Event Polymorphism

In programming, *polymorphism* is the idea that a function interface remains fixed, but the choice of a particular implementation of that function is done at runtime. Many types of polymorphism are defined in modern programming languages. Here we are referring to simple dynamic polymorphism. At the model level, polymorphism arises in the signaling of events to subclasses of a generalization.

Generalization and Set Partitioning

Polymorphic events are applicable only in the context of a generalization relationship in which each subclass has a state model and some events are shared across those state models. We must reiterate that the xUML interpretation of a generalization relationship is more restrictive than the many forms expressed in conventional UML. In xUML, the only generalizations permitted are those tagged as *complete* and *disjoint*. The *complete* tag means that every instance in the superclass has a corresponding instance in a subclass. The *disjoint* tag means that the corresponding instance is in exactly one subclass. So, in a generalization in which Aircraft is either Rotary Wing or Fixed Wing (a helicopter or an airplane), a given Aircraft must be one or the other, and not both, and not neither. Put another way, xUML generalization represents set partitioning rather than type inheritance. The properties of a disjoint set partitioning are pertinent to our discussion. Because it is understood that all xUML generalizations carry the same set partitioning constraints, we usually omit the tags to reduce diagram clutter.

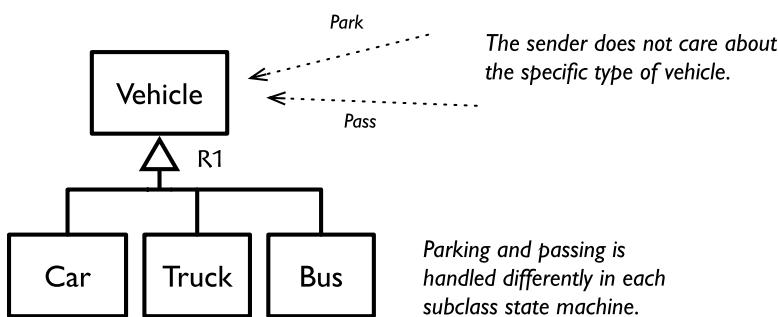


Figure 9-1. A simple generalization

Routing Polymorphic Events

Consider the generalization shown in Figure 9-1.

We assume that each of the three subclasses of Vehicle has a state model and that there is a common set of events to which each state model responds. Let's say the common events are Park and Pass (pass another vehicle). If a state activity of a class outside the generalization wishes to signal a Park event to either Car, Truck, or Bus, that activity would need to locate a particular instance of Car, Truck, or Bus. Usually, it navigates a relationship to an instance of Vehicle and then continues across R1 to find which particular instance is currently related to that instance of Vehicle. Having found the related subclass instance, we can proceed as we would for any event and signal the class instance.

Conceptually, determining which subclass is related to a particular instance of a superclass is not difficult. You just traverse the relationship from the superclass to each of its subclasses. An empty instance reference is obtained on navigation to a subclass to which an instance of Vehicle is *not* related. The properties of a generalization relationship guarantee that exactly one related instance will be found among the subclasses.

This solution has two notable problems. Navigating from the superclass to the subclass sprinkles a large amount of generic and almost identical action language throughout the state activities of the subclass signalers. Also, the action language is fragile, because adding or removing a subclass from the generalization requires reworking the superclass-to-subclass queries for each event signaled to one of the subclasses.

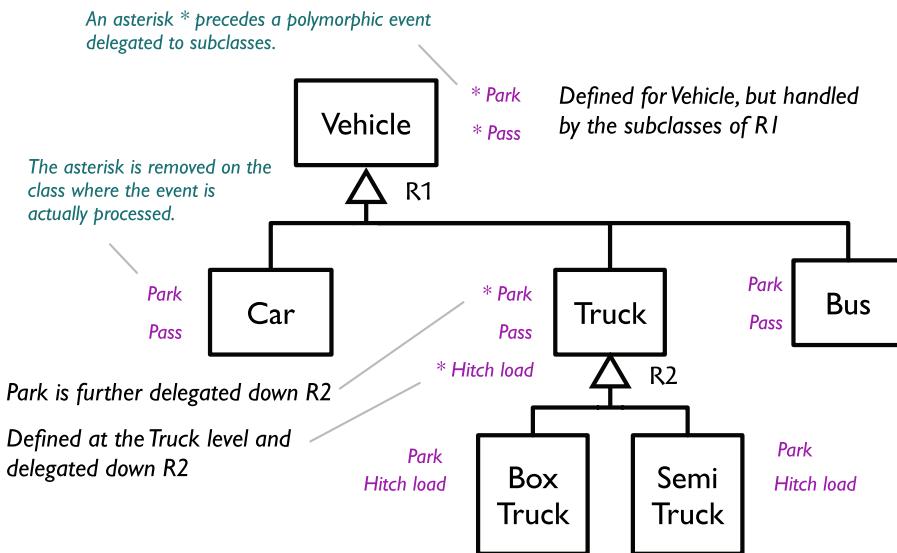
Because the model enumerates the subclasses of a generalization and the events to which any state model responds, we would prefer to address the event to the superclass instance and then let the model execution domain route the event to the appropriate subclass instance. To accomplish this, we designate *polymorphic events* that are signaled to a superclass but handled in one of the subclasses of the generalization.

A polymorphic event gives the signaling state activity the illusion that the superclass is signaled. However, the model execution domain performs the required work, at runtime, to find the currently related subclass instance, map the polymorphic event onto the event set of the related subclass instance, and signal the event. Because nothing is happening in the dispatch of polymorphic events that could not otherwise be handled in the action language of a state activity, polymorphic events can be considered, strictly speaking, an optimization, albeit a convenient and significant one. You can also think of event polymorphism as delegating to the MX domain the task of dispatching events across a generalization when those events have the same meaning in all the subclasses.

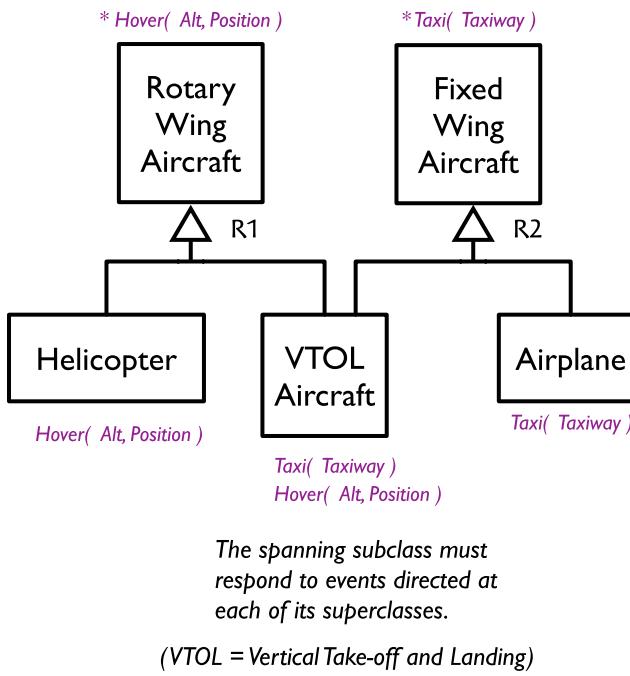
Routing for each Form of Generalization

The basic generalization form illustrated with the Vehicle scenario just described is the most common, but it is not the whole story. Generalization relationships can be composed into more complex forms.

Repeated specialization is characterized by a subclass in one generalization acting as a superclass in a different generalization. Figure 9-2 shows an example of this type of generalization. Polymorphic events may be defined on both superclasses. Those defined at the topmost superclass are delegated to the first level of subclasses. The subclass that is also a superclass of the next-lower generalization may further delegate those same events to its subclasses. It may also define new polymorphic events that are delegated to the second level of subclasses and not visible above the second-level superclass.

**Figure 9-2.** A repeated generalization

Multiple generalization features a class that participates as a subclass in more than one independent generalization. Figure 9-3 shows this case. Our interpretation of generalization does not allow a subclass to participate in multiple generalizations that ultimately have a common superclass ancestor. Such an arrangement violates the disjoint set interpretation of a generalization relationship. The spanning subclass must respond to any polymorphic events directed at each of its superclasses.

**Figure 9-3.** A multiple generalization

Compound generalization is a pattern in which a single class plays the role of a superclass in more than one generalization relationship. Figure 9-4 shows this case. An event signaled to the superclass must be dispatched to all generalizations for which the class serves as a superclass.

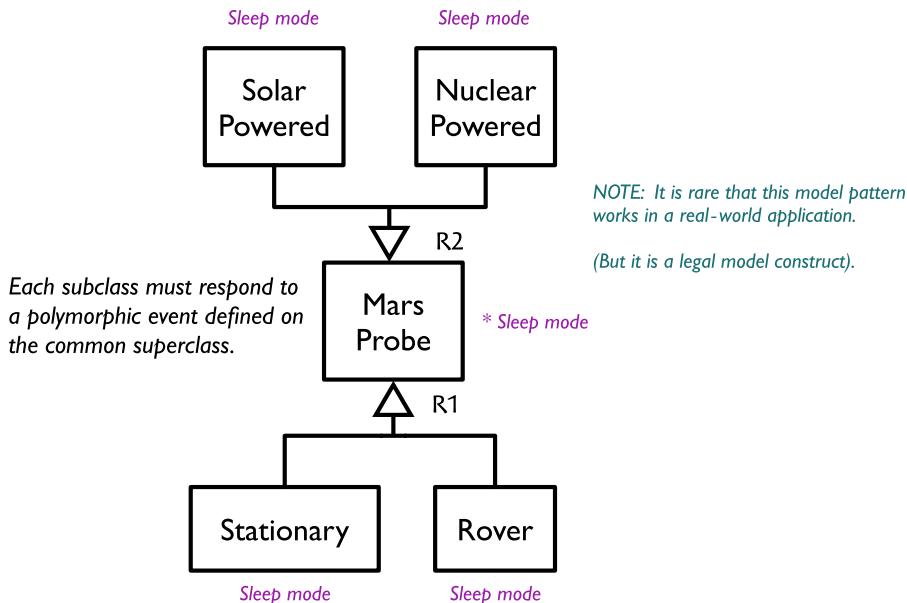


Figure 9-4. A compound generalization

We should note that many of the composed forms involving multiple generalizations may have dubious uses in modeling real-world problems. In particular, compound generalizations impose a strict set of constraints that real-world problems often do not meet. But our concern here is translation, and we must be prepared to render polymorphic event dispatch correctly across any composition of generalizations we are given.

Other implications of polymorphic events impact how they are translated:

- A polymorphic event defined on a superclass does not affect the behavior of that particular superclass. A superclass may have a state model that responds to ordinary, nonpolymorphic events.
- Classes that are repeatedly specialized and intermediate in a generalization hierarchy may or may not have a state model. If they don't have a state model, any polymorphic events are passed along to the subordinate generalization. If they do have a state model, then some, all, or none of the polymorphic events may be consumed by that state model. Any polymorphic events not consumed by the intermediate state model are passed to the subordinate generalization. Any polymorphic events that are consumed are not available to the subordinate generalization.
- A leaf subclass (a subclass that is *not* subject to further specialization) must process any events delegated to it or defined on it. If any polymorphic events have been delegated along the generalization, the leaf subclass must have a state model. Ultimately, all polymorphic events must be mapped to nonpolymorphic events and be accounted for in a state model. This is another way of saying that there is nothing special about a polymorphic event after it has been delegated to a leaf subclass.

Torpedo Launch Example

To illustrate how polymorphic events arise in models and to show how they are translated, we present a small example. This example is an excerpt of a submarine simulation that focuses on the behavior of torpedoes. As with all our examples, we do not imply that this is how a real submarine torpedo would be controlled. The model shown is strictly for pedagogical purposes.

The problem we consider is how torpedoes are stored, made ready, and eventually launched. Torpedoes are initially organized in storage racks. A torpedo is deployed for potential firing by taking it from its storage rack and loading it into a torpedo tube.

Two types of torpedo design determine how torpedoes are launched. One type of torpedo has a motor designed to propel the torpedo out of an open tube. The other torpedo type is launched using an air system that propels both torpedo and water out of the tube.

One additional complication exists. Under rare conditions, it is necessary to recall a torpedo such that there is no attempt to deploy or explode it. A torpedo could be recalled because of a late-discovered design flaw, or the need to reconfigure the torpedo for some reason. If a torpedo is sitting in its storage rack, it is easily recalled for maintenance or disposal. If it is already loaded into a tube when recalled, we must prevent it from firing and remove it from the tube. If the torpedo is already launched, we must be able to disarm a torpedo in flight before it explodes.

As always, we use a class model to capture the relevant abstractions, and it is shown in Figure 9-5. It is helpful to understand the high-level behavior intended for the classes before diving into the details of the individual state models, so we provide an overview of the class diagram.

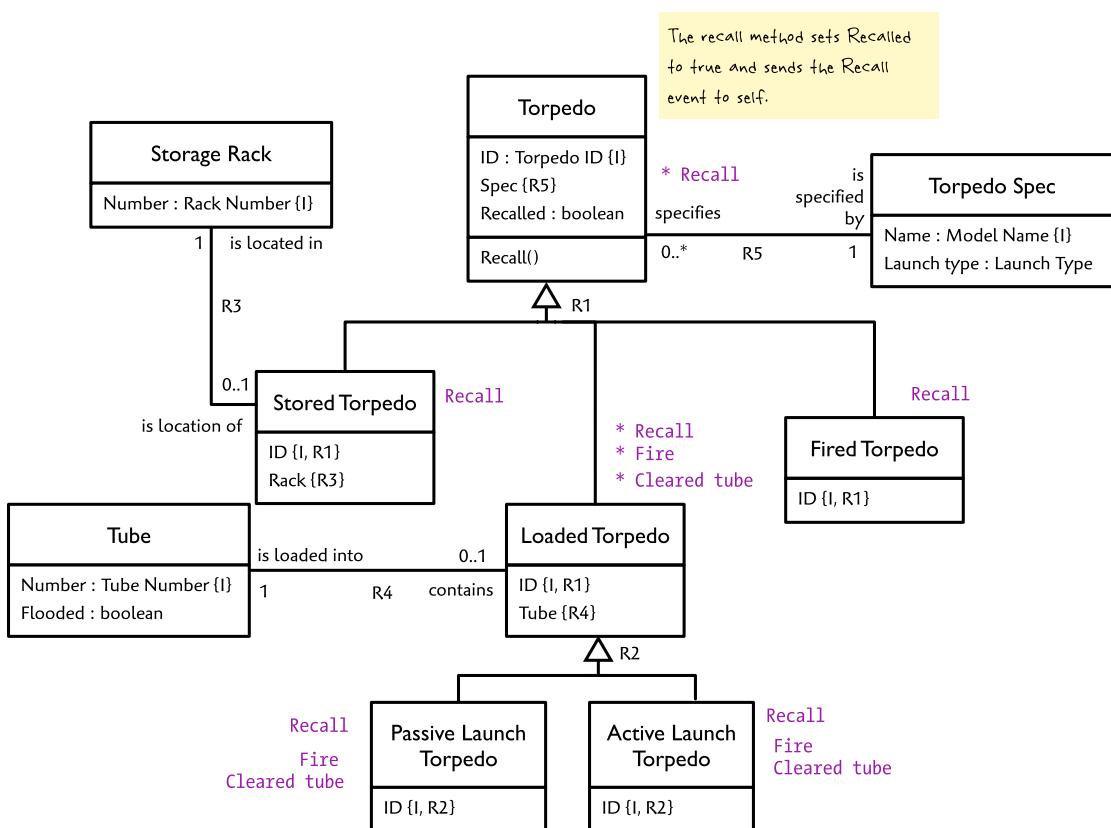


Figure 9-5. Torpedo class model

The R1 generalization captures the three conditions of a Torpedo as being either a Stored, Loaded, or Fired Torpedo. The R5 association between Torpedo and Torpedo Spec provides the *Launch type* attribute that tells us how the Torpedo is to be launched. The attribute can have a value of Active or Passive.

Initially, a Torpedo is created as a Stored Torpedo and assigned to a Storage Rack. A Load event will cause a Stored Torpedo to migrate to being an instance of Loaded Torpedo. The *Torpedo Spec.Launch type* attribute is consulted to determine whether the Passive or Active Launch Torpedo subclass must be instantiated. The Storage Rack association is discarded, and R4 is established to register the Tube into which the Torpedo is loaded.

When the Fire event is received by a Loaded Torpedo, it will escape its Tube and migrate to an instance of Fired Torpedo. The R4 association is discarded because the Torpedo is no longer in a Tube.

Note that the polymorphic events have been annotated on the class diagram. This is purely for illustration and is not considered part of the class diagram. At the top, the Recall event is shown next to Torpedo. The asterisk (*) symbol is an informal way of indicating that this event is polymorphic and may be signaled to an instance of Torpedo, but is processed in one of the subclasses.

As you can see, the Recall event is noted next to each Torpedo subclass. In Executable UML, every subclass must define a way to handle the polymorphic event of the superclass in a given generalization relationship. In other words, if an instance of Torpedo receives a Recall event, a response to that event must be defined regardless of which subclass characterizes the instance of Torpedo at a given moment.

An * remains on the Recall event for the Loaded Torpedo class, indicating that we intend to further delegate the event to all of the subclasses on R2. The * is not present on the Recall event shown next to the Passive and Active Launch Torpedo subclasses. These subclasses are leaves in the generalization and so must have state models that react to the event.

Similarly the *Fire* and *Cleared tube* events directed at Loaded Torpedo are polymorphic and delegated to the subclasses on R2. These events represent new polymorphic events introduced on the R2 generalization and have no effect on the other subclasses of R1. Regardless of whether an instance of Loaded Torpedo is Passive or Active, for example, it must define a response for both the *Fire* and *Cleared tube* events. Considering polymorphic events defined for both R1 and R2, Passive Launch Torpedo and Active Launch Torpedo must respond to Recall, *Fire*, and *Cleared tube*.

Now let's see how event polymorphism appears in the state models. We have decided to build a separate state diagram for each of the leaf subclasses. We start with the Stored Torpedo state model shown in Figure 9-6.

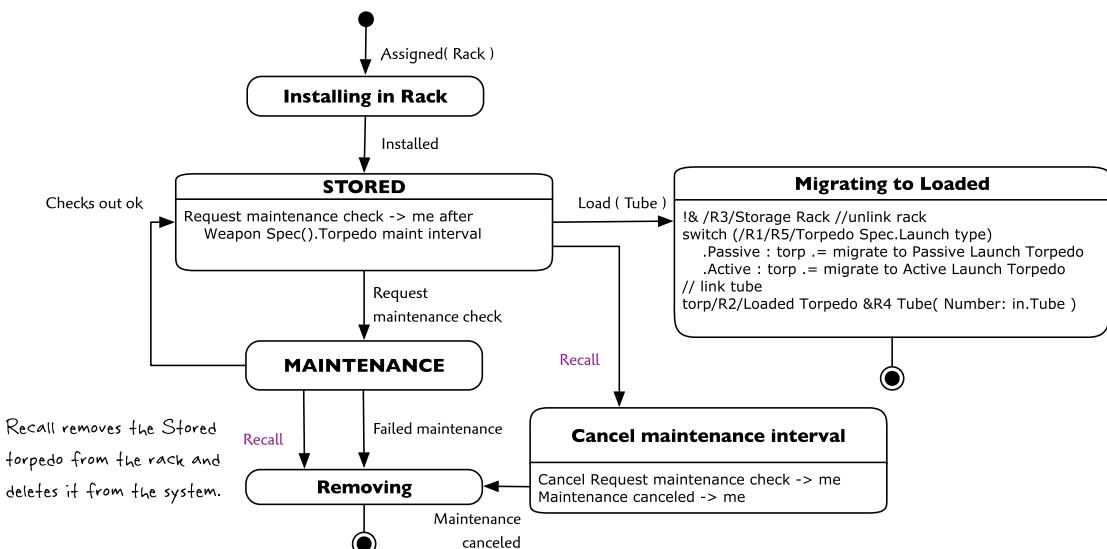


Figure 9-6. Stored Torpedo state model

This state model depicts the life cycle of a Stored Torpedo from the time it is initially created via assignment to a Storage Rack to the point at which it is either recalled or loaded into a torpedo Tube.

When the Load(*Tube*) event is received, reclassification into a new instance of Loaded Tube occurs in the Migrating to Loaded state. Here the association to the Storage Rack is broken, the appropriate instances of Passive or Active Torpedo are created, along with an instance of Loaded Torpedo linked together along R2. Finally, the Tube specified in the event *in.Tube* is selected and linked to the Loaded Torpedo instance. From there, the final state symbol indicates that the Stored Torpedo instance is deleted.

The polymorphic event Recall appears as an ordinary event that triggers a transition in the STORED and MAINTENANCE states. It is safely ignored in the other states, which are all transitory (exited on self-directed events).

There is no Loaded Torpedo state model, by choice, and instead a state model has been defined on each of its subclasses. Let's start with the Passive Launch Torpedo shown in Figure 9-7.

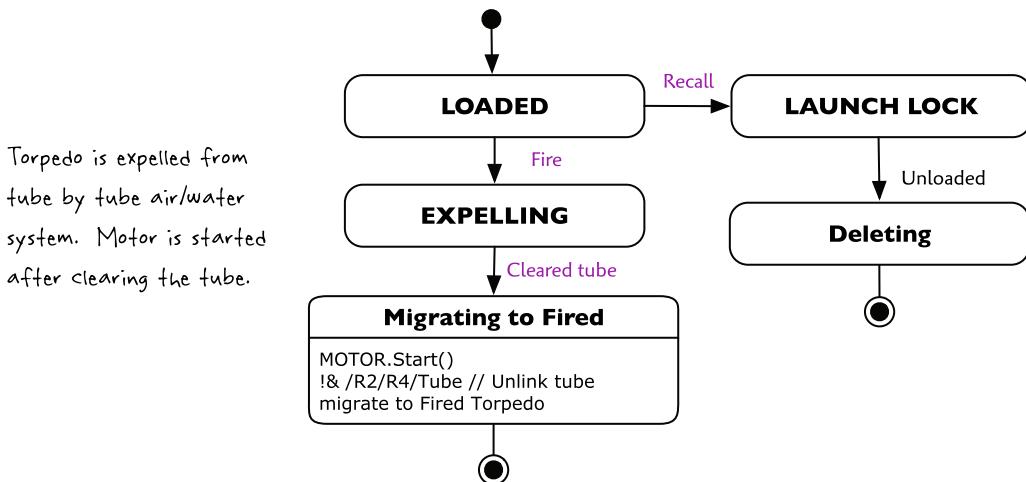


Figure 9-7. Passive Launch Torpedo state model

As the Stored Torpedo instance expires, a newly created instance of Passive Launch Torpedo is created in the LOADED state. The link to the Tube instance has already been established, and the Torpedo waits to be fired. The Fire event has been designated as polymorphic, so it can be signaled to any instance of Loaded Torpedo without regard to whether it is Passive or Active.

When a Fire event is signaled to an instance of Loaded Torpedo that is specialized as an instance of Passive Launch Torpedo, the event is delivered to the subclass instance in the Passive Launch Torpedo state model. This will cause a transition to the EXPELLING state, where the torpedo is pushed out of its Tube.

When the *Cleared tube* event (also polymorphic) is received, we know that the Torpedo is outside the submarine. It transitions to the Migrating to Fired state, where it disassociates itself from its Tube and becomes an instance of Fired Torpedo.

If the Recall event occurs while in the LOADED state, a software lock is placed on the torpedo such that it will ignore any subsequent Fire events. Otherwise, the event is ignored in the EXPELLING state because it is kind of hard to recall a Torpedo halfway out of the Tube! No worries, Figure 9-8 shows that action *can* be taken in the Fired Torpedo state model.

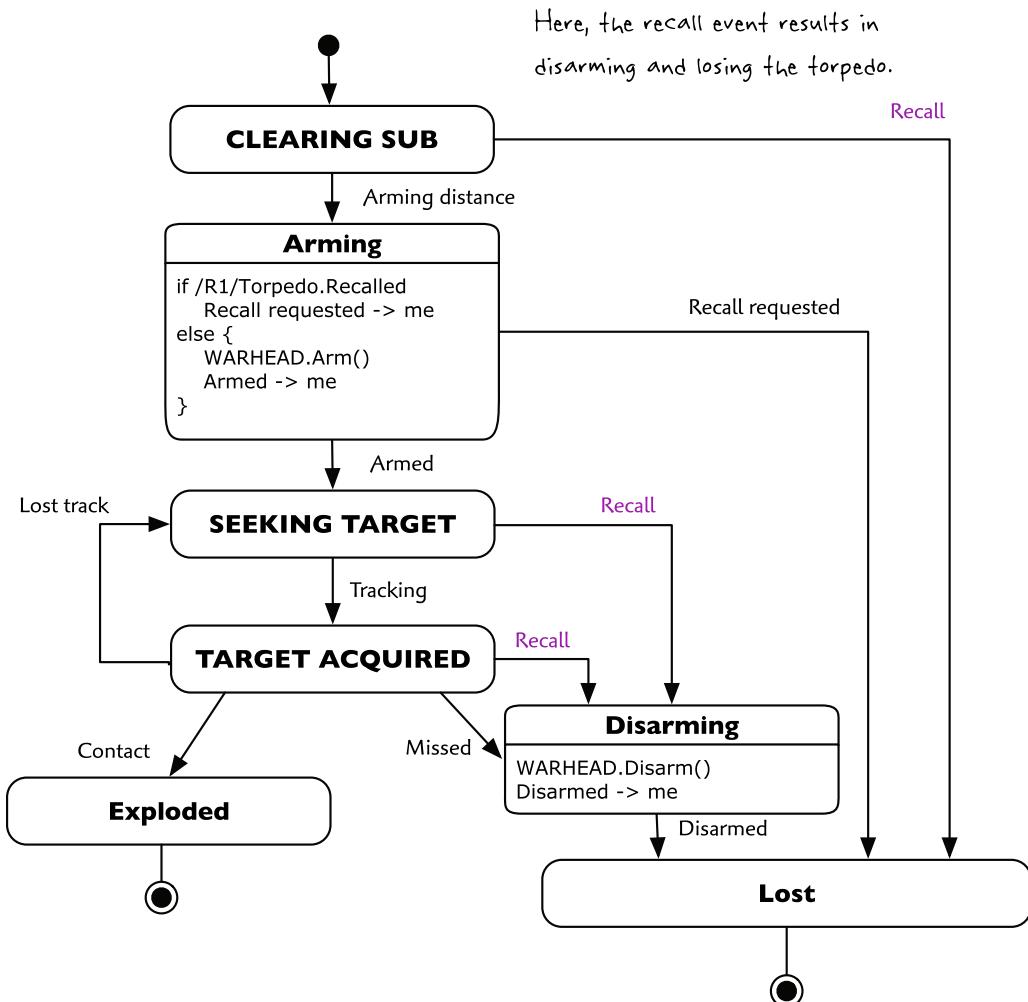


Figure 9-8. Fired Torpedo state model

So now we have a Torpedo that has left the sub on its way, presumably to a target. At this point, the *Recall* event will prevent the Torpedo from being armed. At a safe distance from the sub, the *Arming distance* event will be detected.

There is a danger, however, that a *Recall* event would have been ignored during the Passive or Active Launch Torpedo EXPELLING or ESCAPING TUBE states. And once ignored, an event is lost forever. This is why a class method named *Recall()* has been defined on the *Torpedo* class. To recall a torpedo, this method is called. It does two things. First, it sets the *Torpedo.Recalled* attribute to true. Second, it fires off the *Recall* polymorphic event directed at itself, which is then delegated appropriately. This means that even though the event may be discarded, the active recall status is remembered in the attribute value, which can be checked and cleared in a subsequent state.

In the Arming state, the *Torpedo.Recalled* value is consulted before arming the Torpedo, just in case the *Recall* event occurred in an earlier state, where no productive action could have been taken.

The state model for the Active Launch Torpedo subclass is similar to that of the Passive Launch Torpedo, with the exception of the ESCAPING TUBE state. It is shown in Figure 9-9.

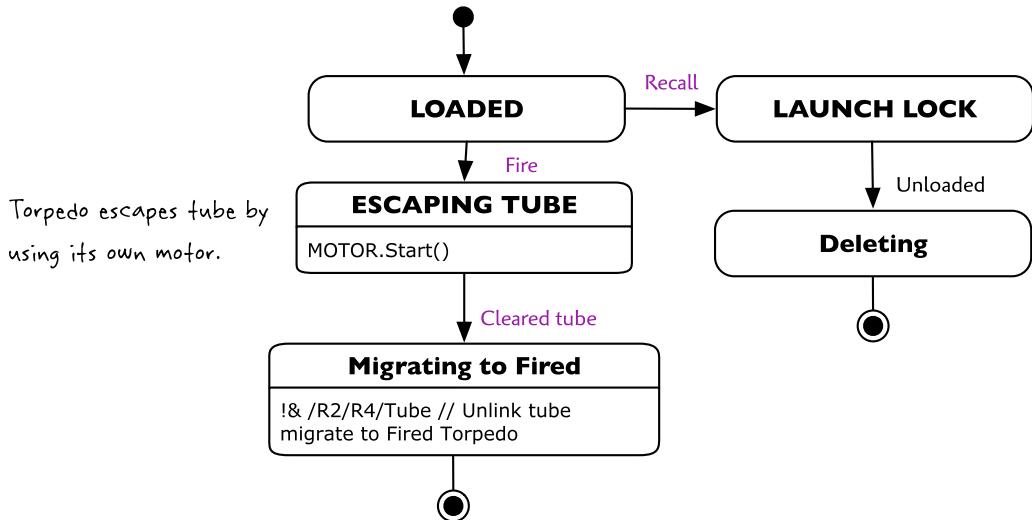


Figure 9-9. Active Torpedo state model

The important thing to note here is that the polymorphic events *Recall*, *Fire*, and *Cleared tube* are all processed in the state models of the subclasses. We reiterate that there is nothing special about polymorphic events when they are eventually consumed. We have delegated to the model execution domain only the tasks of navigating the generalization from the superclass instance to the subclass instance, and mapping the event onto the event set of the receiving instance. After the event is received, it acts as any other event (it causes a transition, is ignored, or creates an error situation). The delegation of the runtime dispatch to the Model Execution domain allows us to signal the polymorphic event to the superclass instance without knowing which corresponding subclass is instantiated. As this example shows, polymorphic events, along with migrating subclasses, can be used to specify complicated multimodal behavior by using only the state models of individual classes.

Translating Polymorphic Events with Pyccua

In this section, we show how the torpedo launch model is translated using pyccua. As usual, we show only those parts of the translation that involve constructs you have not already seen and refer you to the fully worked-out example available as part of the online materials.

First, we must specify which events are polymorphic. Two classes define and delegate polymorphic events, *Torpedo* and *Loaded Torpedo*:

```

class Torpedo
    attribute (Torpedo_ID ID)
    attribute (bool Recalled) default {false}
    reference R5 -> Torpedo_Spec

    polymorphic event
        Recall
    end

```

```

subtype R1 reference          # ②
  Stored_Torpedo
  Loaded_Torpedo
  Fired_Torpedo
end

instance operation
Recall()
{
  self->Recalled = true ;
  PYCCA_generatePolymorphic(Recall, Torpedo, self, self) ;    // ③
}

population dynamic           # ④
slots 20
end

```

① The `polymorphic event` statement introduces the names of the polymorphic events that may be signaled to the `Torpedo` class. Note that we use an asterisk (*) in the preceding code as an informal annotation on the class diagram to illustrate how polymorphic events are mapped across the generalization. Pycca does not use that notation. Events in the consuming state model need only have the same name as the ones declared as polymorphic.

② Here we declare `R1` to use pointer references to implement the generalization. This choice is arbitrary, and we use the `reference` implementation here to demonstrate its use. Polymorphic events would have worked the same if we had used the `union` style of implementation as we did in other models.

③ The instance operation shows how polymorphic events (with no event parameters) are signaled. Note that you have to use a specific macro, `PYCCA_generatePolymorphic()` to signal polymorphic events.

④ Because the number of `Torpedoes` can vary, we declare the population to be `dynamic` and allocate enough space for 20 instances. Remember, all storage is allocated at compile time, and here we state that we have at most 20 torpedoes in our system. Also note that because we have used the `reference` implementation in this case, storage will be allocated for each `Torpedo` subclass. With the `reference` implementation of a generalization, the subclass instances are stored separately from the superclass instances.

The second class to define polymorphic events is the `Loaded Torpedo` class:

```

class Loaded_Torpedo
  reference R1 -> Torpedo      # ①
  reference R4 -> Tube

  polymorphic event            # ②
    Fire
    Cleared_tube
end

```

```

subtype R2 reference
    Passive_Launch_Torpedo
    Active_Launch_Torpedo
end

population dynamic          # ❸
slots 8

```

end

❶ As a subclass in a generalization implemented using a reference, we will need a reference to our related superclass instance, as we need to traverse R1 from subclass to superclass. This back reference is necessary only if we have to navigate R1 from the subclass to the superclass. In this example, we need to perform that navigation. If the activities of Loaded Torpedo never navigate R1 to Torpedo, this reference can be omitted.

❷ Two new polymorphic events are introduced. They are polymorphic across all the generalizations in which Loaded Torpedo participates, which happens to be only R2 in this case. This means that Fire and Cleared_tube must be handled by the state models of Passive Launch Torpedo and Active Launch Torpedo.

❸ The instances of Loaded Torpedo are also dynamic. We have allocated space for only eight Loaded Torpedo instances. Assuming that our submarine has eight torpedo tubes, four forward and four aft, there can never be more than eight torpedoes loaded at a time on a submarine. Because some torpedoes are stored and others have been fired at any given point in time, the storage space allocated here need not be the same as for the Torpedo class instances.

We show the state model for Stored Torpedo implemented in pycca. We have omitted the C code for the state activities so we can focus on the structure of the state model itself:

```

class Stored_Torpedo
    reference R1 -> Torpedo

    reference R3 -> Storage_Rack
    machine
        state Installing_in_Rack(uint8_t rack) {
            // C code for the state activity.
        }
        transition . - Assigned -> Installing_in_Rack          # ❶
        transition Installing_in_Rack - Installed -> STORED

        state STORED() {
            // C code for the state activity.
        }
        transition STORED - Load -> Migrating_to_Loaded
        transition STORED - Recall -> Cancel_maintenance_interval # ❷
        transition STORED - Request_maintenance_check -> MAINTENANCE

        state MAINTENANCE() {
            // C code for the state activity.
        }

```

```

transition MAINTENANCE - Checks_out_ok -> STORED
transition MAINTENANCE - Failed_maintenance -> Removing
transition MAINTENANCE - Recall -> Removing          # ❸
state Removing() {
    // C code for the state activity.
}
final state Removing                                # ❹

state Migrating_to_Loaded(Tube_number Number) {
    // C code for the state activity.
}
final state Migrating_to_Loaded                      # ❺
state Cancel_maintenance_interval() {
    // C code for the state activity.
}
transition Cancel_maintenance_interval - Maintenance_canceled -> Removing
end

population dynamic
slots 20
end

```

❶ Stored Torpedo instances are created asynchronously (via a creation event) using the Assigned event. The period character denotes a transition from the initial pseudo-state.

❷, ❸ The Recall event shows up as any other event would. Pycca will recognize that Stored Torpedo is a subclass of Torpedo along R1 and that *Recall* was defined as polymorphic along R1. So when *Recall* is signaled as a polymorphic event to Torpedo, it is dispatched as a normal event to Stored Torpedo. Exactly how polymorphic event dispatch happens is discussed in a separate section.

❹, ❺ Both the Removing and Migrating to Loaded states are shown as final states in the state model. When a Stored Torpedo is recalled, it is removed from the system and no longer exists. When a Stored Torpedo is loaded, we create a new instance of Loaded Torpedo and therefore must delete the instance of Stored Torpedo to maintain the R1 generalization constraints.

Finally, we show the state activity for the Migrating to Loaded state of Stored Torpedo. This example shows how the subclass migration from a Stored Torpedo to a Loaded Torpedo is implemented in pycca. Let's first refer back to the action language for the state activity:

```

!& /R3/Storage Rack //unlink rack
switch (/R1/R5/Torpedo Spec.Launch type)
.Pассив : torp .= migrate to Passive Launch Torpedo
.Актив : torp .= migrate to Active Launch Torpedo
// link tube
torp/R2/Loaded Torpedo &R4 Tube( Number: in.Tube )

```

Because we have chosen to implement both R1 and R2 by using reference generalizations, we need to manage the instance creation and deletion explicitly. Migrating implies that an instance of the new subclass is created and related back to the superclass and that the instance of the old subclass is deleted. This processing would be different if we had chosen to implement the generalizations by using the union mechanism. Because with the union implementation, subclass storage is included as part of the

superclass storage, migration simply modifies things in place without creating and deleting instances. Both generalization implementation mechanisms have their uses and trade-offs.

```

state Migrating_to_Loaded(Tube_number Number) {
    // Since this is a final state, the Stored Torpedo instance is deleted
    // automatically at the end of this activity and since there were no
    // references in Storage Rack back to the Stored Torpedo, we don't have to
    // deal with the R3 association.

    // Get a reference to the superclass instance.
    ClassRefVar(Torpedo, torp) = self->R1 ;
    // Create an instance of Loaded Torpedo
    ClassRefVar(Loaded_Torpedo, ltorp) = PYCCA_newInstance(Loaded_Torpedo) ;
    // Relate the Loaded Torpedo to the Torpedo across R1.
    PYCCA_relateSubtypeByRef(torp, Torpedo, R1, ltorp, Loaded_Torpedo) ; // ❶
    ltorp->R1 = torp ;
    // Relate the Loaded Torpedo to the Tube.
    ltorp->R4 = PYCCA_refOfId(Tube, rcvd_evt->Number) ;

    // Now handle R2
    switch (torp->R5->Launch_type) { // ❷
        case Active: {
            // For Active Launch Torpedos, create a new instance.
            ClassRefVar(Active_Launch_Torpedo, altorp) =
                PYCCA_newInstance(Active_Launch_Torpedo) ;
            // Relate the Active Launch Torpedo across R2.
            PYCCA_relateSubtypeByRef(ltorp, Loaded_Torpedo, R2, altorp,
                Active_Launch_Torpedo) ; // ❸
            altorp->R2 = ltorp ;
        }
        break ;
        case Passive: {
            // For Passive Launch Torpedos, create a new instance.
            ClassRefVar(Passive_Launch_Torpedo, pltorp) =
                PYCCA_newInstance(Passive_Launch_Torpedo) ;
            // Relate the Passive Launch Torpedo across R2.
            PYCCA_relateSubtypeByRef(ltorp, Loaded_Torpedo, R2, pltorp,
                Passive_Launch_Torpedo) ;
            pltorp->R2 = ltorp ;
        }
        break ;
    }
}

```

❶ Relating the superclass and subclass across R1 is a two-step undertaking. First, we update the pointer in the Torpedo superclass to the newly created subclass instance, including the type of subclass (Loaded Torpedo) to which the superclass points. Second, we must set up the reference from the subclass to the Torpedo superclass.

❷ The Launch type attribute of Torpedo Spec determines which subclass we must create for R2.

❸ The same process is used to relate superclass and subclass instances across R2.

How Polymorphic Events Are Signaled

In Chapter 5, we showed how normal events are signaled and dispatched. Signaling requires obtaining an event control block (ECB), filling it in with the proper data, and posting the ECB to the event queue. The same basic set of operations are required to signal a polymorphic event.

Pycca provides a macro to interface to the runtime code. For example, sending the Fire event to a Loaded Torpedo might appear in a state activity as follows:

```
PYCCA_generatePolymorphic(Fire, Loaded_Torpedo, ltorp, self) ;
```

This signals the Fire polymorphic event to the ltorp instance of the Loaded Torpedo class.

If the polymorphic event has parameters, you would need to obtain the ECB, fill in the parameters, and then post the event in the same way as is done for normal events. The only difference is that there is a macro that obtains an ECB that has been marked as polymorphic. For some hypothetical event, *Reload*, that takes a parameter named *When*, signaling the event might appear as follows:

```
MechEcb polyecb = PYCCA_newPolymorphicEvent(Reload, Torpedo, torp, self) ;
PYCCA_eventParam(polyecb, Torpedo, When) = 20 ;
PYCCA_postEvent(polyecb) ;
```

Pycca encodes the polymorphic event numbers as zero-based consecutive integers. This numeric encoding is distinct from that used for normal events. Some classes will have both a set of normal events and a set of polymorphic events. To distinguish the different event sets, an enumeration is defined, and each ECB carries a value of this enumeration to indicate what must be done to dispatch each different event type:

```
typedef enum {
    NormalEvent,
    PolymorphicEvent,
    CreationEvent
} MechEventType ;
```

How Polymorphic Events Are Dispatched

In Chapter 5, you saw how a normal, nonpolymorphic event is signaled and later dispatched. To signal, an ECB is filled out and queued. The dispatch of a normal event uses the ECB and the object dispatch block of the receiving instance's class to compute the transition to a new state and invoke the associated activity. Similarly, dispatching a polymorphic event requires data from the ECB combined with data from a polymorphic dispatch block (PDB).

Additionally, the dispatch of a polymorphic event involves navigating the generalization relationship from the superclass instance to find the related subclass instance. Recall that pycca uses two strategies for storing generalization relationship information:

- Generalizations stored as references include, in the superclass instance structure, a pointer to the subclass. Because each subclass has a different data type, the superclass pointer is typed as *MechInstance* (as a pointer to a generic instance).
- Generalizations stored as unions include, in the superclass instance structure, a union of the data structures for all the subclasses of the generalization.

This difference impacts the way the related subclass instance reference is computed. An enumeration is used to discriminate the two cases:

```
typedef enum {
    PolyReference,
    PolyUnion
} PolyStorageType ;
```

Regardless of the way the generalization is stored, pycca defines a member in the superclass structure that encodes the type of the subclass to which the superclass instance is currently related. Pycca generates this encoding as consecutive integers starting at zero, which we use as array indices.

After an event is removed from the event queue, the `eventType` member of the ECB determines whether the event to dispatch is polymorphic. If it is, we can find the polymorphic dispatch block by following the `targetInst` member of the ECB, which points to the instance receiving the polymorphic event. Like all instances with dynamic behavior, its `instClass` member holds the value of a pointer to its class information. For classes that have defined polymorphic events, the class information contains a non-NULL value for the `pdb` member that points to a polymorphic dispatch block:

```
typedef struct polydispatchblock {
    DispatchCount eventCount ;
    DispatchCount hierCount ;
    HierarchyDispatch hierarchy ;
} const *PolyDispatchBlock ;
```

<code>eventCount</code>	The number of polymorphic events defined for or delegated to this class and handled in a subclass.
<code>hierCount</code>	The number of generalization hierarchies in which the class participates. This value is usually 1, but superclasses in a compound generalization will have a <code>hierCount</code> greater than 1.
<code>hierarchy</code>	A pointer to an array of hierarchy dispatch blocks. The array has <code>hierCount</code> number of elements.

If the superclass participates in a compound generalization, multiple elements will be in the `hierarchy` array. The polymorphic event is dispatched down each generalization relationship in which the superclass participates, so the dispatch code iterates over the `hierarchy` array. Commonly, there is only a single generalization.

Two operations are required to dispatch a polymorphic event. First, we must navigate the generalization from the superclass to the subclass. Second, we need to map the polymorphic event onto the event set of the subclass. The hierarchy dispatch block provides the information needed for both of these operations:

```
typedef struct hierarchydispatch {
    PolyStorageType refStorage ;
    AttributeOffset subCodeOffset ;
    AttributeOffset subInstOffset ;
    DispatchCount subtypeCount ;
    struct polyeventmap const *eventMap ;
} const *HierarchyDispatch ;
```

refStorage	Records whether the generalization relationship is stored in reference form or in union form.
subCodeOffset	Holds the byte offset from the beginning of the instance structure to where the type encoding for the currently related subclass is held. (The encoding is the structure element typed SubtypeCode described in Chapter 5.)
subInstOffset	Holds the byte offset from the beginning of the instance structure to the location of either a pointer to a subclass instance or to the union of the subclass structures.
subtypeCount	Holds the number of distinct subclasses that exist for this generalization relationship. This value is used for runtime checks.
eventMap	A pointer to the polymorphic event mapping for the hierarchy. This mapping is indexed in row order by subtype code and in column order by polymorphic event number. Each entry in the mapping array is of the following type.

```
typedef struct polyeventmap {  
    EventCode event ;  
    MechEventType eventType ;  
} const *PolyEventMap ;
```

To compute a reference to the subclass instance, the subInstOffset member is added to the beginning of the superclass instance pointer. Depending on the value of the refStorage member, the resulting address is either a pointer to the subclass instances (union storage) or a pointer to a member that is in turn a pointer to the instance (reference storage).

The type of the subclass is determined by using the subCodeOffset to compute the location in the superclass where the encoded subclass type is stored. Using the subclass code value and the polymorphic event number, we can index into the eventMap. To make the data type of the event map the same for all classes, we store it as a one-dimensional array and then compute the index by using the subtype code as a row number and the polymorphic event number as a column number. This is the same tactic we used when dispatching normal events from a transition table.

The event mapping consists of both an event and an eventType. Because polymorphic events can pass down multiple levels in a repeated specialization hierarchy, a polymorphic event may map to yet another polymorphic event, and so we need to know the event type. Given the event type, the event number, and the subclass instance pointer, we can dispatch the event recursively. Figure 9-10 shows the data involved in dispatching a polymorphic event.

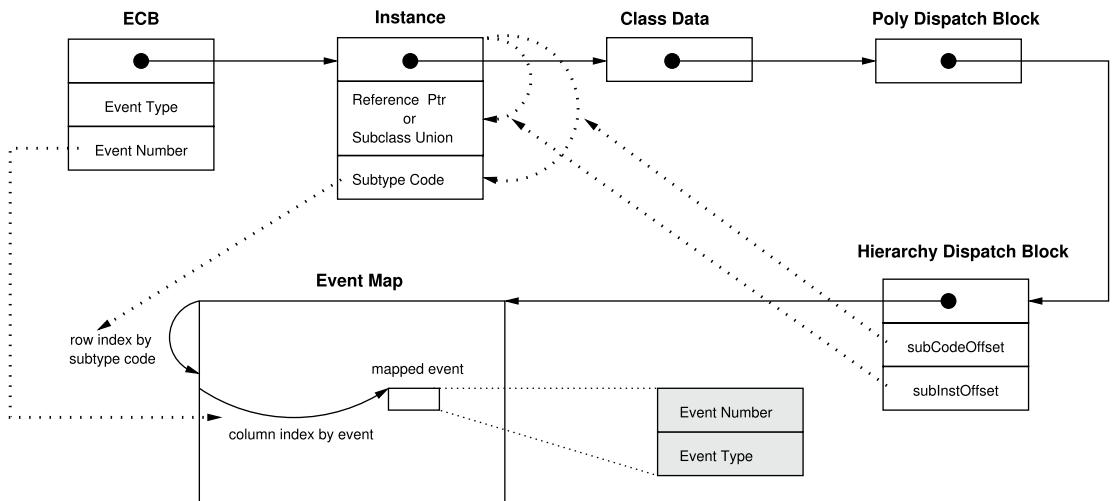


Figure 9-10. Data used in dispatching a polymorphic event

Figure 9-11 is an illustration of a scenario in which the polymorphic Fire event is signaled to a Loaded Torpedo instance.

Signaling a Polymorphic Event

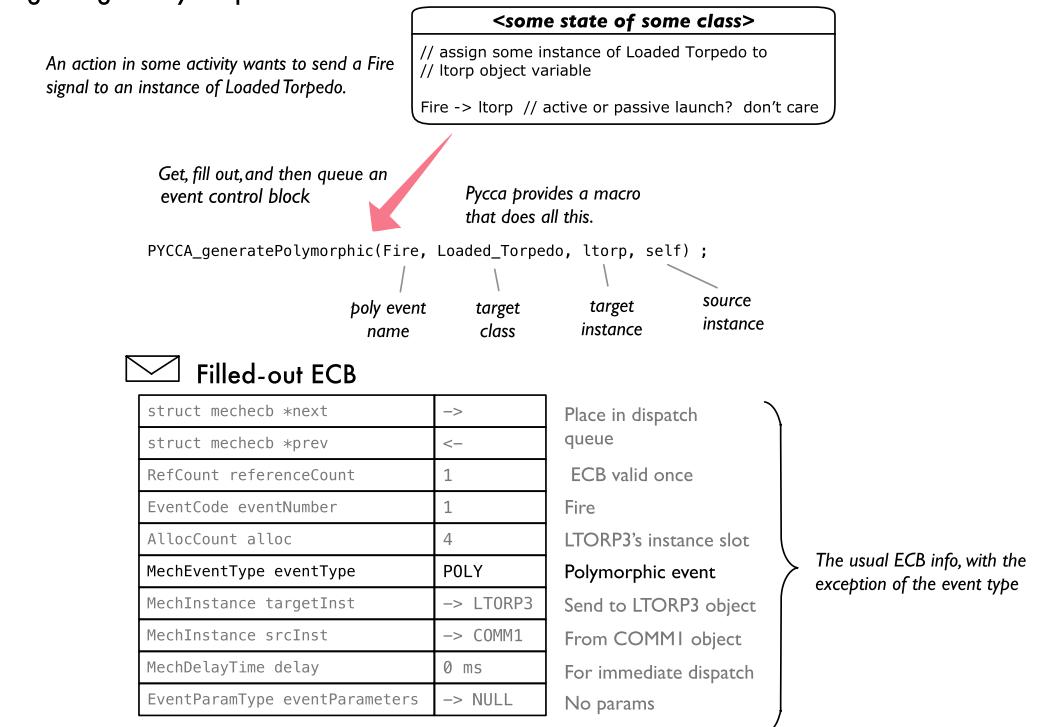


Figure 9-11. Signaling the Fire polymorphic event

An ECB has been filled out and queued. The only element that distinguishes this from any normal event ECB is the eventType field filled out as a PolyEvent. Once dequeued for dispatch, the polymorphic dispatch block must be found. Figure 9-12 illustrates how the PDB is located.

Finding what we need to dispatch a polymorphic event

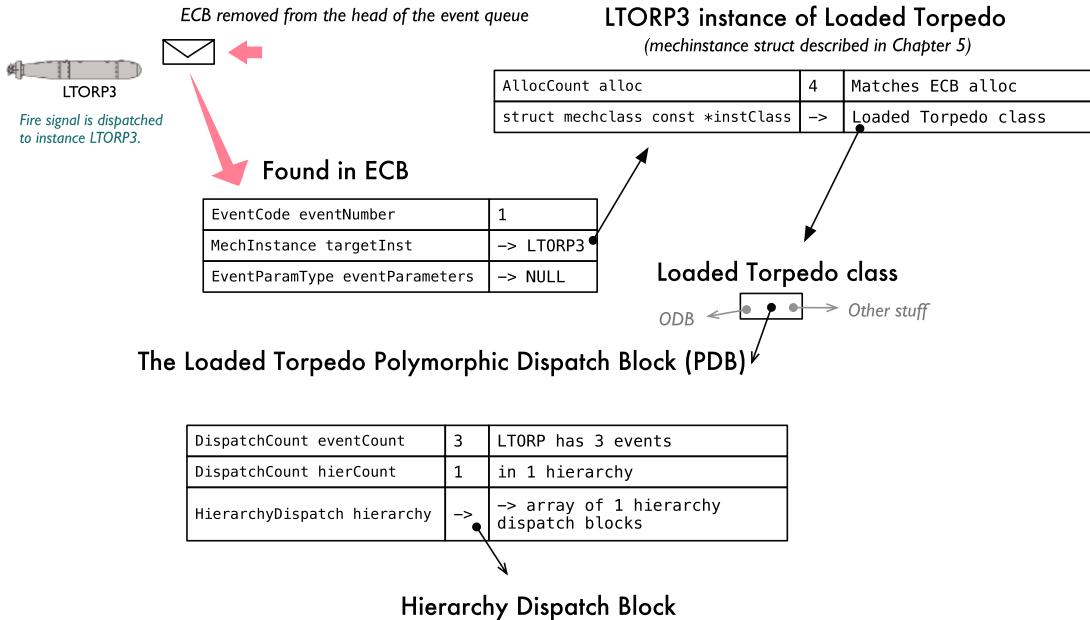


Figure 9-12. Locating the PDB

Finally, Figure 9-13 shows how the polymorphic event map is indexed to determine which normal event to send.

Dispatching a polymorphic event

Hierarchy Dispatch Block

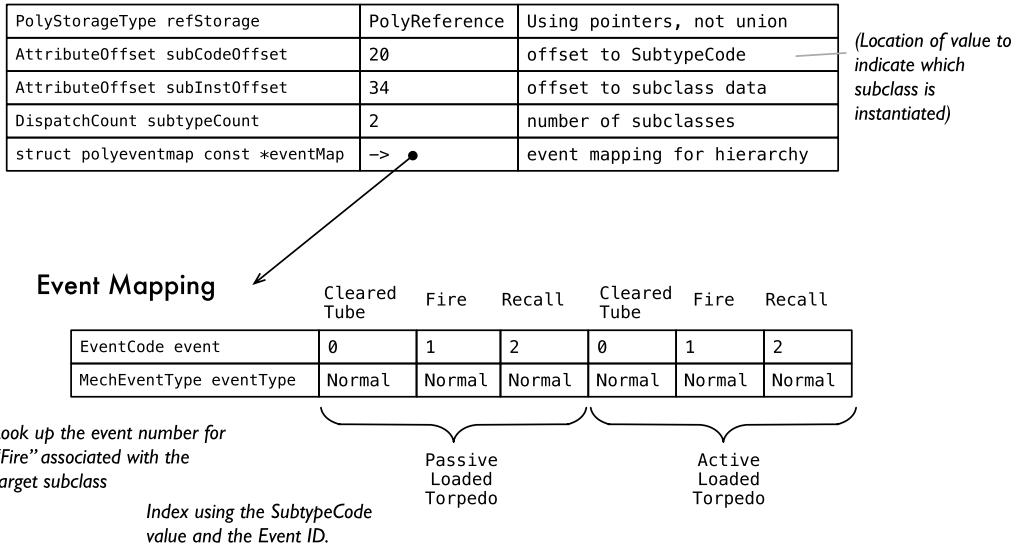


Figure 9-13. Indexing into the event map

Returning to our torpedo launch example, here is the code generated by pycca to support the polymorphic event dispatch for the Torpedo and Loaded Torpedo classes:

```
/*
 * PDB for Class, "Torpedo"
 */
static struct polyeventmap const Torpedo_R1_pem[] = {
    {.event = 6, .eventType = NormalEvent},
    {.event = 2, .eventType = PolymorphicEvent},           // ①
    {.event = 6, .eventType = NormalEvent}
} ;
static struct hierarchydispatch const Torpedo_hdb[] = {
    .refStorage = PolyReference,
    .subCodeOffset = offsetof(struct Torpedo, R1_code),
    .subInstOffset = offsetof(struct Torpedo, R1),
    .subtypeCount = 3,                                     // ②
    .eventMap = Torpedo_R1_pem}
} ;
static struct polydispatchblock const Torpedo_pdb = {
    .eventCount = 1,
    .hierCount = 1,
    .hierarchy = Torpedo_hdb
} ;
```

```

/*
 * PDB for Class, "Loaded_Torpedo"
 */
static struct polyeventmap const Loaded_Torpedo_R2_pem[] = {
    {.event = 0, .eventType = NormalEvent},
    {.event = 1, .eventType = NormalEvent},
    {.event = 2, .eventType = NormalEvent},
    {.event = 0, .eventType = NormalEvent},
    {.event = 1, .eventType = NormalEvent},
    {.event = 2, .eventType = NormalEvent}
} ;
static struct hierarchydispatch const Loaded_Torpedo_hdb[] = {
    .refStorage = PolyReference,
    .subCodeOffset = offsetof(struct Loaded_Torpedo, R2_code),
    .subInstOffset = offsetof(struct Loaded_Torpedo, R2),
    .subtypeCount = 2,
    .eventMap = Loaded_Torpedo_R2_pem
} ;
static struct polydispatchblock const Loaded_Torpedo_pdb = {
    .eventCount = 3,
    .hierCount = 1,
    .hierarchy = Loaded_Torpedo_hdb
} ;

/*
 * Class Structure for, "Torpedo"
 */
static struct mechclass const Torpedo_class = {
    .iab = &Torpedo_iab,
    .odb = NULL,
    .pdb = &Torpedo_pdb
} ; // ❸

/*
 * Class Structure for, "Loaded_Torpedo"
 */
static struct mechclass const Loaded_Torpedo_class = {
    .iab = &Loaded_Torpedo_iab,
    .odb = NULL,
    .pdb = &Loaded_Torpedo_pdb
} ;

```

❶ Here is a case where a polymorphic event is mapped to another polymorphic event. In this case, it is the Recall event from R1 being propagated down the R2 generalization. It maps to a polymorphic event on Loaded Torpedo and to a normal event in both the Stored and Fired Torpedo subclasses.

❷ The number of elements in the event map (Torpedo_R1_pem) is always equal to the subtypeCount value times the eventCount value from the PDB (Torpedo_pdb).

❸ Here the PDB shows up as part of the class information. Notice that neither Torpedo nor Loaded Torpedo responds to normal events, and so pycca sets their odb members to NULL.

In keeping with our theme to expose all the details of how a translated application runs, we show the code that performs polymorphic event dispatch. At this point, the ECB has been removed from the event queue, and it has been determined that a polymorphic event must be dispatched:

```

static void
dispatchPolyEvent(
    MechEcb ecb)
{
    PolyDispatchBlock pdb = ecb->instOrClass.targetInst->instClass->pdb ;

    assert(pdb != NULL) ;
    assert(ecb->eventNumber < pdb->eventCount) ;
    assert(pdb->hierCount > 0) ;
    /*
     * Each generalization hierarchy that originates at the supertype has an
     * event generated down that hierarchy to one of the subtypes.
     */
    HierarchyDispatch hd = pdb->hierarchy ;
    for (unsigned hnum = 0 ; hnum < pdb->hierCount ; ++hnum) {
        /*
         * The most common case is to dispatch along a single hierarchy. In any
         * case, we can modify in place the input ECB on the last dispatched
         * event.
         */
        MechEcb newEcb ;
        if (hnum == pdb->hierCount - 1) {
            newEcb = ecb ;
        } else {
            newEcb = mechEventAlloc() ;
            /*
             * We set the source as the original sender.
             */
            newEcb->srcInst = ecb->srcInst ;
            /*
             * Copy event parameters.
             */
            newEcb->eventParameters = ecb->eventParameters ;
        }
        SubtypeCode type = *(SubtypeCode *)
            ((char *)ecb->instOrClass.targetInst + hd->subCodeOffset) ; // ❶

        assert(type < hd->subtypeCount) ;
        PolyEventMap pem = hd->eventMap +
            (type * pdb->eventCount + ecb->eventNumber) ; // ❷
        #ifdef MECH_SM_TRACE
        /*
         * Trace the transition.
         */
        tracePolyEvent(ecb->eventNumber, ecb->srcInst,
            ecb->instOrClass.targetInst, type, hnum,
            pem->event, pem->eventType) ;
        #endif /* MECH_SM_TRACE */
    }
}

```

```

newEcb->eventNumber = pem->event ;
newEcb->eventType = pem->eventType ;

void *subTypeRef = (char *)ecb->instOrClass.targetInst + hd->subInstOffset ;
newEcb->instOrClass.targetInst = hd->refStorage == PolyReference ?
/*
 * When the generalization is implemented via a pointer, we need an
 * extra level of indirection to fetch the address of the subtype.
 */
*(MechInstance *)subTypeRef :
/*
 * When the generalization is implemented by a union, we need only
 * point to the address of the subtype as it is contained in the
 * supertype.
 */
(MechInstance)subTypeRef ;

if (newEcb->eventType == NormalEvent) {
    newEcb->alloc = newEcb->instOrClass.targetInst->alloc ;
    assert(newEcb->alloc != 0) ;
}

mechDispatch(newEcb) ;
++hd ;
}
}

```

❶ This bit of address arithmetic computes the location of the member of the instance structure that stores the encoded value of the currently related subclass and fetches the value located there. For example, if we were dispatching a polymorphic event to an instance of Torpedo, this expression would compute the address of the `R1_code` member and fetch the value from that location.

❷ This expression indexes into the polymorphic event map. Because the map is stored as a one-dimensional array, we have to undertake the row/column indexing ourselves. The `eventCount` tells us the number of elements that are in a row. Pycca ensures that the elements in the polymorphic event map are stored in the proper order so this indexing expression fetches the correct event mapping.

Again, distilled to its essence, the code to dispatch a polymorphic event is quite short. It derives most of its “intelligence” from the values of the data structures provided by pycca.

Summary

In this chapter, we showed how polymorphism at the model level is specified and implemented. Polymorphic events may be defined for the superclass in a generalization relationship. At runtime, polymorphic events signaled to superclass instances are dispatched across the generalization to state

machines in the subclasses. We showed how to specify polymorphic events to pycca, the data pycca generates to support the polymorphic event dispatch, and exactly how the ST/MX domain performs such event dispatch.

All xUML generalizations represent a set partitioning rather than inheritance. For each superclass, there is exactly one subclass instance, and this instance is in exactly one subclass.

Generalizations may be interconnected to yield the following forms:

- *Repeated specialization*: A subclass of one generalization is also a superclass of another generalization.
- *Multiple generalization*: A single subclass participates in more than one generalization with a superclass in each.
- *Compound generalization*: A single class plays the role of a superclass in more than one generalization.

It is important that polymorphic events are sensibly dispatched in each form.

A superclass may delegate events so that they are handled in each subclass rather than by a state model on the superclass. A subclass may then either handle an event or, if it is also the superclass of another generalization, further delegate it. Each delegated event must be handled by a subclass at the current level or at a deeper level of specialization. A received event is always processed in a class.

Each polymorphic event is defined on a superclass and designated in the pycca model script. An event delegated to a subclass may be used by that subclass, just like any other event in the subclass's state machine section of the model script.

Distinct pycca macros are provided to support the signaling of a polymorphic event with and without parameters.

Polymorphic event dispatch is handled differently, based on whether the generalization was implemented using the union or reference code pattern. The event control block used for normal events is also used for dispatching polymorphic events. The key difference is that an additional polymorphic dispatch block is referenced to navigate the generalization's dispatch hierarchy and identify the corresponding subclass event that is processed.

CHAPTER 10



Pycca and Other Platforms

In this chapter, we discuss the design and implementation of the pycca program itself. Pycca is designed as a language processor that reads DSL statements to populate a platform model and generates code by using a template system that queries the populated platform model. It is implemented in the Tcl language.

We also present some size and execution speed measurements of the ALS system on a representative microcontroller platform to show that the resulting memory usage and execution speed are appropriate for our targeted platform.

To conclude the chapter, we present a brief overview of a target platform using Berkeley DB, a key/value data store engine, to store domain data rather than keeping all the data in memory.

Design of the Pycca Program

There remains one area in our translation approach that we have not discussed. We have shown input to pycca and output from it, but we have not discussed pycca itself as a program. Space does not allow a complete description of the pycca implementation. The source code and documentation for pycca are freely available from this book's website. In this section, we discuss several aspects of the design and implementation of the pycca program itself.

Pycca has three major design elements:

- Platform model
- Domain-specific language processing
- Template-driven code generation

Platform Model

It should come as no surprise that there is a model underlying pycca operations. In the following discussion, we use the same names for things that are at two levels of abstraction. Let's define some terms to keep things clear. When we speak of the *executable model*, we are referring to the model of logic that is to be translated. When we speak of the *platform model*, we are referring to the model of the implementation technology platform onto which we are translating.

The platform model for pycca is specific to the targeted implementation technology. This is not a general model of modeling itself (a metamodel). The pycca platform model does not set the rules for how to create an executable model. Rather, it gives the rules for how our C implementation will be formed. The platform model reflects the technology choices we have made for the specific implementation that pycca generates. Targeting a different platform or making different choices about the implementation details on our platform would be reflected in platform model differences.

Figure 10-1 shows a fragment of the platform model for pycca.

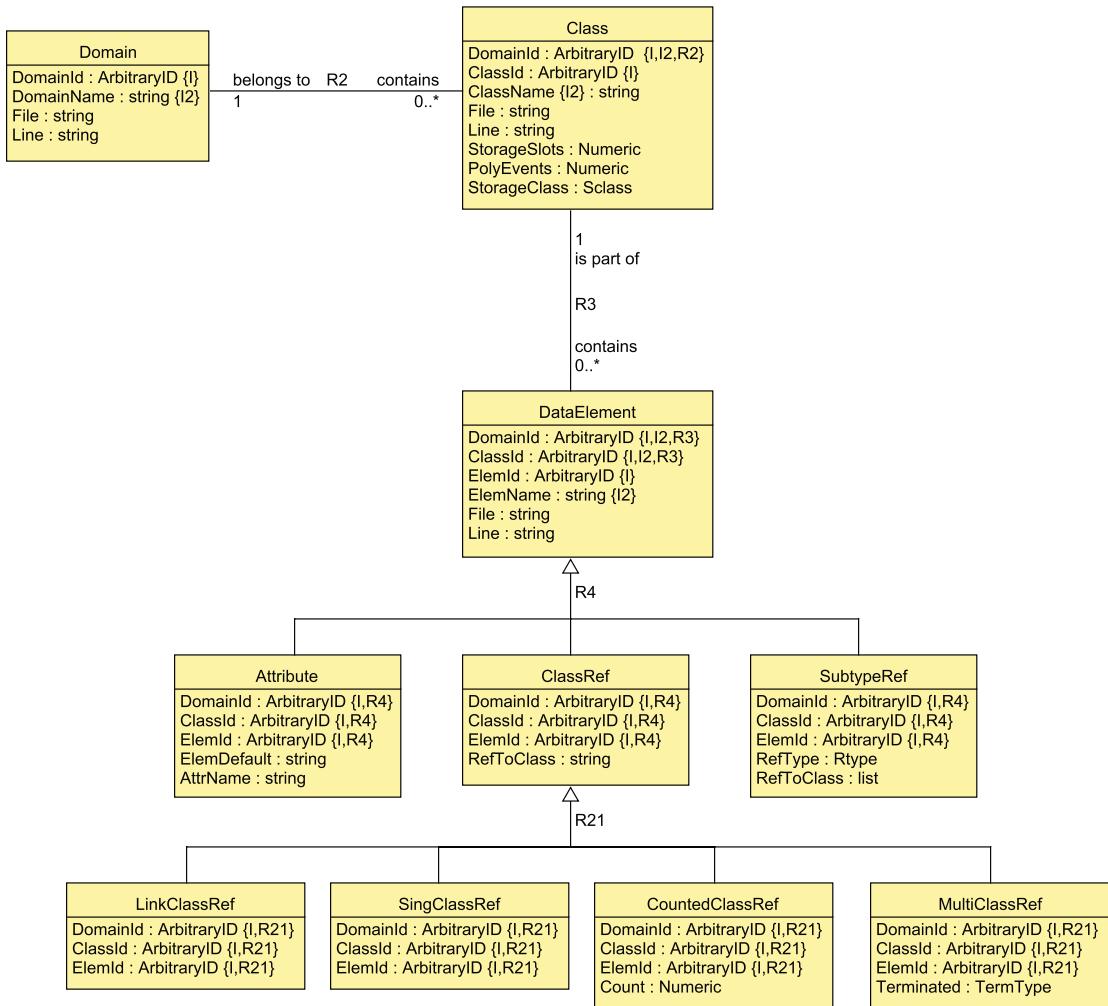


Figure 10-1. Fragment of pycca platform model

Notice that there are classes named Domain and Class. These classes model the implementation counterparts of a Domain or Class in the executable model.

The platform model states that Classes contain Data Elements and that a class may contain no Data Elements at all (R3). Data Elements are part of exactly one class, and so there is no sharing of Data Elements among Classes. All Data Elements are one of three types (R4): Attributes, ClassRefs, or SubtypeRefs. Further, class references are one of four types (R21).

For our target, we have decided that all class instances will be held in memory and stored in arrays of C structures. The members of a class structure are either attributes, references (of some form) to classes, or references to subclasses in a generalization. The classes in the platform model are used to model the implementation structure members that are generated by pycca. For pycca, the Class class in the platform model has a direct correspondence to the C structure that is generated for the implementation.

For example, consider the Duty_Station class definition from our Air Traffic Control model:

```

class Duty_Station
    attribute (Station_Number Number)
    attribute (Name_t Location)
    attribute (Aircraft_Maximum Capacity)
    reference R3 -> On_Duty_Controller
end

```

The R3 reference is a singular reference to an instance of the `On_Duty_Controller` class. An instance of `SingClassRef` (and all its related superclass instances) would be created to correspond to the reference statement in the `Duty_Station` class definition. When `pycca` generates code, it uses this information in two ways. When the `Duty_Station` structure is declared, it contains a member, `struct On_Duty_Controller *R3`. This is shown in the following `pycca`-generated structure definition:

```

/*
 * Duty_Station structure definition
 */
struct Duty_Station {
    struct mechinstance common_ ; // must be first !
    Station_Number Number ;
    Name_T Location ;
    Aircraft_Maximum Capacity ;
    struct On_Duty_Controller *R3 ;
} ;

```

When creating the initializers for the initial instance population, the R3 member is set to the address of an `On_Duty_Controller` array member or `NULL`, depending on the population requested. This is shown for the initial instance population of our example:

```

/*
 * Initial Instance Storage for, "Duty_Station"
 */
static struct Duty_Station Duty_Station_storage[3] = {
    {.common_ = {1, 0, &Duty_Station_class}, "S1", "Front", 20, .R3 = NULL},
    {.common_ = {2, 0, &Duty_Station_class}, "S2", "Center", 30, .R3 = NULL},
    {.common_ = {3, 0, &Duty_Station_class}, "S3", "Front", 45, .R3 = NULL}
} ;

```

Notice that many of the classes in the platform model have `File` and `Line` attributes. These attributes record the name of the input file and line number within the file where the entity was defined. This information is helpful for producing error messages as well as for inserting `#line` directives in the generated code file.

The `pycca` platform model has 35 classes. They cover the types of information that are specified in the DSL language statements, such as classes, attributes, state transitions, and initial instances. The `pycca` platform model forms the fundamental basis for how the rest of the program is organized and how the processing required for code generation works.

Domain Specific Language Processing

Generating a parser for a computer language is a well-understood problem. `Pycca` follows the usual pattern of defining a grammar and using a parser generator to create a `pycca` DSL interpreter. As the input is scanned and recognized, code is executed when grammar elements are reduced. The code executed when DSL statements are recognized creates instances of the platform-model classes. Data from the language

statements correspond to attributes in the platform-model class instances. You can think of the pycca DSL as a textual and more convenient representation of a platform-model population. The same effect could be created by populating the platform model directly, but using a DSL provides an opportunity to organize the platform-model population in a more human-friendly way.

As with other computer languages, it is possible to write pycca DSL statements with correct syntax that are meaningless. So, pycca performs a set of semantic checks after the input files have been parsed and the platform model populated. Some of the semantics are enforced by the constraints on the platform model, such as those implied by platform-model relationships. Others require executing code.

For example, if a state in a state model is marked as a final state, no outbound transitions are allowed from that state. Because an instance that transitions to a final state is deleted after its state activity is executed, it is not possible for it to respond to other events. Pycca also disallows isolated states (states that have neither outbound nor inbound transitions). There are 20 such semantic checks.

Template-Driven Code Generation

The code generator for pycca is designed using template expansion. This is a common idiom for generating everything from accounting reports to web pages. Pycca uses it much like any template system. Text is passed from the template to the output. Commands are embedded in the template, and when the template expander recognizes a command, it is executed, and the result returned from the embedded command is placed in the output. The template pycca uses contains embedded commands which query the platform model to find the information needed at that point in the code generation and then format the information as valid C language statements. Conceptually, there is little difference between what pycca does and what a banking program might do to consult a database and print an account statement. The goal of pycca is to produce output for a C compiler rather than a bank customer.

The use of a template permits us to order the generated output to suit the C compiler. There is a separate template for the generated header and code files. The templates are designed and ordered so that the embedded expansion commands place their output at the appropriate location in the output. The C language requires a lot of type annotation and insists that symbol names be declared before (or sometimes at the same time) as they are defined. Declarations and definitions are key concepts in C, and usually declarations must precede definitions. For example, forward declarations of state activity functions are placed before the definitions of the data structures that use the function names. These are then placed before the definitions of the state activity's function code.

Pycca Implementation

Pycca is implemented in the Tcl language. Tcl is a dynamic language often used as a scripting language. The choice of a scripting language for implementing pycca may seem unusual, but Tcl has many desirable characteristics:

- Tcl is a mature, stable language and has been under active development for more than 20 years.
- Tcl is platform independent and runs on Linux (and other UNIX derivatives), macOS, and Windows.
- Tcl programs can be distributed as single-file executable without external dependencies or complicated installation requirements.
- Tcl is extensible and supports a large standard library of procedures.

In practice, the choice of language used to implement a program such as pycca should be made on matters of availability and convenience. It should be a language that is familiar to the implementer and that conveniently supports the implementation of the design ideas from the previous section. Here we discuss how Tcl is used to implement the pycca design.

The platform model for pycca is a normalized relational schema, as are all of our example models. Pycca uses the TclRAL (Relational Algebra Library) package to implement the operations on the platform model. The TclRAL package implements relational algebra, in which the operators are Tcl commands and the values and variables are directly integrated into the Tcl language. You can think of the TclRAL implementation of the platform model as an in-memory database that uses ordinary Tcl language commands to query and manipulate the platform-model data. It was designed expressly to support the integrity constraints of our modeling approach. TclRAL is completely integrated into the Tcl value system. There is no “impedance mismatch” in TclRAL in the sense that one does not use a different language to query and manipulate data and then have to transfer query results back into the implementation language for further processing. Contrast this approach with SQL, which requires you to deal with the inevitable boundary between the implementation language and the query language.

Following in the tradition of the venerable lex and yacc programs, Tcl has fickle and taccle to perform the same functions of generating a lexical analyzer and a parser. In this case, the generated analyzer and parser are delivered as Tcl code rather than C code.

The standard Tcl library contains procedures to perform template expansion, and these are used to generate the header and code files. Templates contain ordinary text passed directly to the output and embedded Tcl commands. The embedded commands are executed, and their output is written to the generated file. The embedded commands are implemented as TclRAL queries on the platform model with appropriate formatting of the results into C language statements.

The pycca program consists of approximately 5,000 lines of Tcl code, grammar specification, and lexical analyzer specification, not counting comments or blank lines. We are not fond of using lines of code as a software metric and present only a gross, relative indication of the size of the pycca program. So pycca is not a large program, but much of that can be attributed to using Tcl and the fact that Tcl as a language is expressive in fewer lines of code. If it had been written in C, pycca would probably be several times larger.

Pycca Performance

Not only is it necessary to achieve a translation of a model into running code, but it is also essential that the quality of the resulting implementation meet the needs of the project. It does little good to produce insightful models and a faithful translation to code if the performance requirements are not met.

Our target platform is microcontroller-based systems. The major challenge for these types of systems is the limited computing resources available. Both memory and processor speeds are usually quite small. In this section, we show some performance numbers for the translation of the ALS system.

Target Hardware Platform

Many commercially available platforms would satisfy our needs. We have chosen the Giant Gecko from Silicon Labs. This microcontroller is available in a starter kit called EFM32GG-STK3700.

The microcontroller on the starter kit is the EFM32GG990F1024. This computer is an ARM Cortex-M3 based SOC with 1 MiByte of flash and 128 KiByte of RAM. The SOC is capable of ultra-low power sleep modes and consumes approximately $219 \mu A / MHz$ when executing code from flash memory. Although capable of running at a 48 MHz clock frequency, the microcontroller was clocked at 7 MHz in these measurements. These specifications place the Giant Gecko at the more capable end of the class of microcontrollers we target.

Target Software Platform

The code for this example was built using the Silicon Labs Simplicity Studio development environment. We have included code from the vendor-supplied hardware access library as well as startup code and other small code pieces required to build a complete application.

The application was built using the GNU compiler suite:

```
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 5.4.1 20160919 (release)
[ARM/embedded-5- branch revision 240496]
```

The application was compiled with preprocessor symbols `NDEBUG`, `MECH_NINCL_STDIO`, and `RELEASE` defined to remove all the assertions and uses of C standard I/O functions (for example, `printf`). Optimization was configured for minimum size using the `-Os` setting. Unused data and functions in object files were discarded in the final executable. This combination of build settings yields the smallest executable. When built for debugging and instrumentation, executables are often twice as large as those built for release.

ALS Code Size

The following measurements are for the ALS application example shown in Chapters 6–8. This example consists of two domains and the bridge code between them. The Lubrication domain contained external entity references to a UI and Alarms domain, and these references have been stubbed out. Also the SIO domain contained external entity references for access to device hardware, and these too have been stubbed out.

Table 10-1 shows the overall memory usage, in bytes, of the integrated lube/sio application.

Table 10-1. ALS Application Code Size (Bytes)

File	Code and Constants	Initialized Data	Uninitialized Data	Total
<code>lube_sio.axf</code>	12,064	1,784	756	14,606

The total memory usage for the application easily fits our target hardware and would fit many other target hardware platforms of this class as well.

We can break down the memory usage by examining the sizes of the domains and the bridge code, as shown in Table 10-2.

Table 10-2. ALS Domain Code Size (Bytes)

File	Code and Constants	Initialized Data	Uninitialized Data	Total
<code>lube.o</code>	2,663	476	0	3,139
<code>sio.o</code>	2,070	776	0	2,846
<code>lube_sio_bridge.o</code>	471	0	0	471
Total	5,204	1,252	0	6,456

Of the total memory usage, less than half is devoted to the domain and bridge code. We should note that the initial instance population for this application was quite small, and this is reflected in the size of the initialized data. Increasing the size of the initial instance population would only increase the usage of initialized data. The code and constants memory usage would remain the same, regardless of the size of the initial instance population, because that memory usage already includes all the code and pycca-generated information for the domains. Note that the initialized data also results in the same amount of space being

allocated to RAM, which does not show up in these totals. At reset time, the initialized data is copied from flash memory to RAM by compiler-supplied startup code. Because flash memory technology does not allow direct updating in the same way as RAM, the RAM copy allows the values of class instances to be updated during the running of the program. This need to allocate twice the space for initialized data (in flash and an equal amount in RAM) is a consequence of the usual split between flash memory and RAM characteristic of microcontroller SOC designs.

It is worthwhile examining the contribution of the ST/MX domain to overall memory usage. This is important because the Model Execution domain must be present to run pycca-generated applications and represents a fixed cost that must be amortized across the entire application. See Table 10-3.

Table 10-3. Model Execution Domain Code Size (Bytes)

File	Code and Constants	Initialized Data	Uninitialized Data	Total
mechs.o	2,032	12	720	2,764
pycca_portal.o	990	0	0	990
platform.o	471	1	0	472
Total	3,521	13	720	4,226

The `mechs.o` file contains all the target-independent code of the MX domain. This includes the event queues and event signal/dispatch code, and so forth. The `pycca_portal.o` file consists of the portal functions used in bridging. Finally, the `platform.o` file consists of the platform-specific code required by the MX domain. This includes control of the timing resource used for delayed events, code to deal with low-power mode sleep and wake-up, and code to interface to the synchronization queue. The uninitialized data is allocated to event queues, event control blocks, and other internal resources of the ST/MX implementation. These resources can be expanded or contracted as needed, and this total represents the default sizing of 10 event control blocks with 16 bytes of parameter data space and 10 sync queue slots.

The memory usage by the ST/MX domain compares favorably with that of many RTOS implementations that target microcontrollers. However, that comparison is not direct. An RTOS will include facilities for multitasking, inter-task synchronization, and mutual exclusion not present in ST/MX. ST/MX is single threaded, strictly event driven with run-to-completion execution, and does not need tasking services nor the synchronization operations and mutual exclusion mechanisms required to support them. Conversely, ST/MX can signal and dispatch events to state machines, a feature not available in RTOSs.

The approximately 25 percent of the remaining memory usage comes from startup code, standard C libraries, compiler libraries, hardware access libraries, external entity stubs, and other “glue” code necessary to obtain a running application. While this is a substantial part of the total memory usage, it represents a fixed cost for the application.

Execution Speed

We present only one execution speed measurement, shown in Table 10-4. Signaling and dispatching an event is a common operation in the ST/MX domain. Here we have measured the number of CPU cycles to signal and dispatch a self-directed event. This includes the time required to allocate the ECB, fill it in, add it to the event queue, return to the main loop to decide whether there is another event to dispatch, remove the ECB from the event queue, compute the transition, and enter the state activity. Conveniently, the ARM Cortex-M3 includes a cycle counter for these purposes.

Table 10-4. Timing to Signal and Dispatch an Event

Cycles	Time @ 7 MHz	Operations / s @ 7MHz
4,696	67 μ s	1,491

It takes 4,696 cycles to complete the signal/dispatch operation. At the 7 MHz clock frequency, this means it takes approximately 67 μ s or alternatively, we may perform approximately 1,491 such operations per second. These numbers appear distorted when compared to the capabilities of modern desktop and server class computers running at 2–3 GHz frequencies and having large instruction and data caches. We must be careful not to extrapolate between such vastly different computing technologies.

Performance Discussion

Performance comparisons in this realm are difficult to make. We have no benchmarks to test MX domain implementations. Direct comparisons are seldom possible, since providing the same functionality for an application in both a modeled and non-modeled implementation is so expensive.

The measurements presented here demonstrate that the performance of pycca-translated domains do match the computational facilities provided by many microcontroller-based systems. We have remained well within our memory targets and with a reasonable cycle count for the implementation of event signaling. Experience over many systems has shown that carefully tailoring the translation scheme and model execution domain to the target platform allows translated models to meet a project's performance goals.

Supplying Implementation-Specific Code

When we model application logic, the scale of the problem is one of the implementation aspects that we do not consider. Whether a class has only a few instances or millions of instances does not change the fundamentals of the application logic. If a particular activity is required to find a class instance based on the value of an attribute, whether there are only a few instances or a great many instances does not affect the fact that we must find the required instance.

But when we consider the implementation of such searching, scale matters a great deal. When the number of instances of a class becomes large enough and the frequency at which we must search the instances to find a particular one increases, the simple sequential search provided by pycca macros can become a performance problem.

Consider our Air Traffic Controller class from Chapter 4:

```
class Air_Traffic_Controller
    attribute (Employee_ID ID)
    attribute (Name_T Name)
    attribute (Experience_Level Rating)
    # ...
    # other parts of the Air Traffic Controller Class
end
```

We could use a pycca macro to find an instance of Air Traffic Controller that matches a given ID:

```
ClassRefVar(Air_Traffic_Controller, atc) ;
PYCCA_selectOneInstWhere(atc, Air_Traffic_Controller, strcmp(atc->ID, "ATC-137") == 0) ;
if (atc >= EndStorage(Air_Traffic_Controller)) {
```

```

    // not found
} else {
    // found
}

```

This code performs a simple, sequential iteration across all the instances of Air Traffic Controller and compares the value of the ID attribute looking for a match. This approach is simple, already provided, and for a small number of instances, works well. As we scale up the number of instances of Air Traffic Controller, we will need something better.

We could, for example, use a binary search to reduce the number of comparisons. A binary search requires a particular ordering of the searched items. Pycca, however, organizes the initial instance population in memory in the order of definition. To convert our sequential search to a binary search, we need to order the initial instances of Air Traffic Controller by ascending order of the ID attribute. Let's suppose our initial instance population is as follows:

```

table
Air_Traffic_Controller (Employee_ID ID) (Name_T Name) (Experience_Level Rating) R1
@atc51 {"ATC-51"} {"Ianto"} {"C"}      -> On_Duty_Controller.atc51
@atc53 {"ATC-53"} {"Toshiko"} {"A"}     -> On_Duty_Controller.atc53
@atc67 {"ATC-67"} {"Gwen"} {"B"}       -> On_Duty_Controller.atc67
@atc77 {"ATC-77"} {"John"} {"B"}       -> Off_Duty_Controller.atc77
@atc87 {"ATC-87"} {"Fred"} {"B"}       -> Off_Duty_Controller.atc87
# ...
# many other controllers in ascending order of the ID attribute
# ...
end

```

By defining a `class operation`, we can add our own code to apply a binary search to find a matching Air Traffic Controller. The standard library `bsearch()` function requires the following:

- A pointer to the search key
- A pointer to the beginning of an array of items to be searched
- The number of items in the array
- The size of each array item
- A function returning an integer that compares two items

We would define the `class operation` as part of the `Air_Traffic_Controller` class:

```

class Air_Traffic_Controller
    # ... other parts of the Air Traffic Controller Class
    class operation findByEmployeeID(char const *eid) :
        (struct Air_Traffic_Controller *) {
            struct Air_Traffic_Controller key = {
                .ID = eid
            } ;
            return (struct Air_Traffic_Controller *)bsearch(&key,
                BeginStorage(Air_Traffic_Controller),           // ❶
                ATCTRL_AIR_TRAFFIC_CONTROLLER_INST_COUNT,    // ❷
                sizeof(struct Air_Traffic_Controller),
                atc_compare_ids) ;                          // ❸
    }
}

```

```

end
implementation prolog { // ④
    #include <stdlib.h>
    #include <string.h>
    static int
    atc_compare_ids(void const *m1, void const *m2)
    {
        struct Air_Traffic_Controller const *atc1 = m1 ;
        struct Air_Traffic_Controller const *atc2 = m2 ;
        return strcmp(atc1->ID, atc2->ID) ;
    }
}

```

① The BeginStorage macro resolves to the address of the storage array for the given class.

② Pycca emits a macro definition for the number of instances of a class.

③ We must supply a comparison function for bsearch.

④ Placing the comparison function in the implementation prolog ensures that its definition appears before it is used in the class operation.

Now we can locate a reference to Air Traffic Controller, ATC-137, by using the following code:

```

ClassRefVar(Air_Traffic_Controller, atc) ;
atc = ClassOp(Air_Traffic_Controller, findByEmployeeID)("ATC-137") ;
if (atc == NULL) {
    // not found
} else {
    // found
}

```

There are, of course, other ways to implement this search. For example, we can use bsearch only for static instance populations. For a dynamic population of Air Traffic Controller instances, we might choose to keep a hash table. Then activity code that creates or deletes an Air Traffic Controller instance would also add or remove the instance reference from the hash table, which would be keyed by the ID attribute value. You would probably code the instance creations and hash table addition operations together into a class operation, and similarly with the instance deletion and hash table removal operations.

Our point here is that the implementation can be tailored to match the scale of the problem. Most important, the specific implementation mechanisms *do not affect* the model logic. If the model logic requires a particular Air Traffic Controller to be found, then the translation must choose the appropriate implementation for the search, and that choice is based, at least in part, on the number of instances that need to be searched. Searching is a well-researched problem in computer science whose results we can draw upon here.

Considering Other Platforms

In this book, we have remained focused on one particular target platform. This focus on a single platform has helped demonstrate concepts in translation without the burden of showing how the same effect is achieved using another implementation mechanism. In this section, we broaden our discussion to consider a different translation target. Space does not allow an extended discussion of the many possibilities of implementation technology that we might wish to use to meet the requirements of a system. To bound our discussion, we change only the platform requirement for how data is managed. We keep the programming language as C and the execution single threaded. Many of the concepts for our microcontroller target carry forward, and so we focus on how data management might be changed.

How data is held and managed is one of the key factors in determining the characteristics of an MX domain. In the microcontroller target platform, we decided that all domain data would be held in primary memory, directly addressable by the processor. Here, we change that requirement and insist that our alternative MX domain hold data in secondary storage. Many classes of application either have too much data to be held in primary memory or have other requirements to persist domain data into nonvolatile secondary storage.

In this section, we outline what a MX domain might look like that holds domain data in secondary storage. We do not present a complete MX domain in this discussion. Rather, we present a series of examples of how model-level data concepts might be implemented using a persistent data storage mechanism. Again, we must be precise about what the target platform supports:

- The implementation language is C.
- Domain data is managed using Berkeley DB.
- Execution is single threaded.
- The target hardware platform is a desktop or server class of processor.
- We assume a POSIX operating system environment with GiBytes of primary memory and secondary disk storage at least 10 times the size of primary memory.

The main differences between the microcontroller target that we have been discussing and this new platform is the use of Berkeley DB to manage the domain data and the assumption of a much more capable computer running a fully featured operating system. We have purposely kept the implementation language and the single-threaded nature of the execution the same as our microcontroller target to avoid introducing other elements.

Berkeley DB is a general-purpose embedded database engine. The central concept in Berkeley DB is that of a persistent key/value data store in which keys and values are arbitrary byte arrays of data. The library is mature, well supported, and provides features well beyond our uses in this example. Complete information on Berkeley DB can be found at the <http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html>.

Mapping Domain Data to Berkeley DB

Our first task is to map model execution data concepts to Berkeley DB implementation mechanisms. Figure 10-2 shows how model data management concepts are mapped onto Berkeley DB facilities and how Berkeley DB uses the file system for persistent storage.

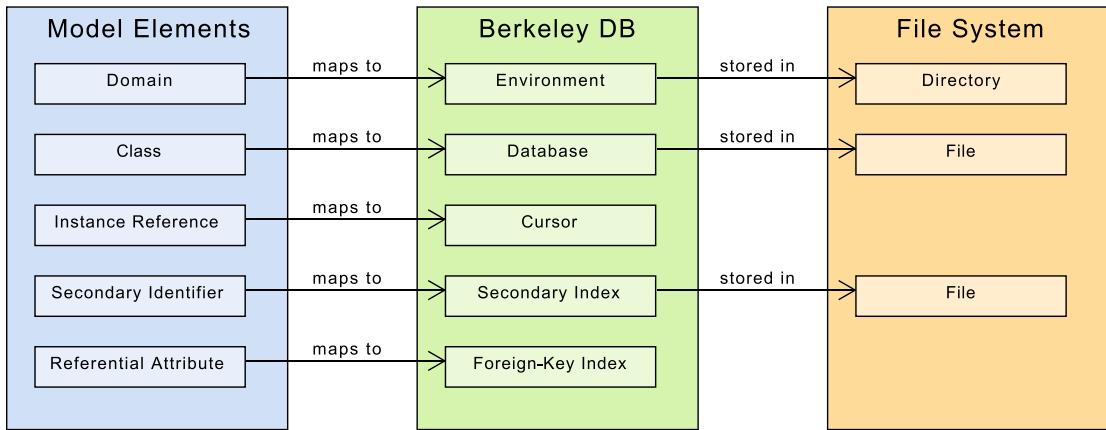


Figure 10-2. Mapping domain data to Berkeley DB

For this example, we use parts of the Lubrication domain from Chapter 6. We discuss each of these concepts and also show small code sequences to demonstrate how Berkeley DB functions might appear in the MX domain and how the implementation of the mapping of model data management onto Berkeley DB is realized in C code. We don't expect you to be a Berkeley DB expert and recognize every library call. Rather, you can get a general feel for how the data management would be coded, and documentation of the database library calls is readily available for those who wish to delve deeper. You can also get a good sense of how different data management is in ST/MX, where everything is held in memory, compared to using a key/value pair storage mechanism.

This example deals with just two classes from the Lubrication domain: Injector and Machinery. To refresh your memory, Figure 10-3 is a fragment of the class diagram from the Lubrication domain.

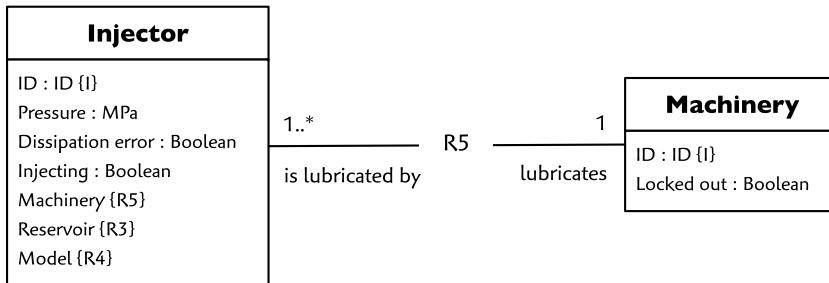


Figure 10-3. Lubrication domain class diagram fragment

Following the mapping in Figure 10-2, we start with enclosing a domain's data in a Berkeley DB *environment*. This construct suits our needs to manage data for a single domain. An environment provides a grouping for databases and transaction capability, and necessary files are stored in a single directory of the file system:

```

int dbres = 0 ;

dbres = db_env_create(&lube_env, 0) ; // ①
if (dbres != 0) {
    handle_error(dbres, "Error creating environment handle") ; // ②
}
    
```

```

dbres = lube_env->open(lube_env, "./lube_domain", // ❸
    DB_CREATE | DB_INIT_MPOOL, 0) ;
if (dbres != 0) {
    handle_error(dbres, "Environment open failed") ;
}

```

❶ Most entities in the library are created first, before any other operations are performed.

❷ For brevity, we assume some error-handling function.

❸ To use the environment, we must open it.

This code creates the environment if needed (it may already exist), and all the files will be placed in the `lube_domain` directory.

A domain class is stored as a *database*. In Berkeley DB, a database is roughly the same as a table, which matches a view of class data consisting of a set of instances that form rows in a table. After an environment is open, we can create and open the databases that correspond to the classes. We show only code for the Injector class, but all classes for a domain would have their own database for instance storage:

```

DB *injdb ;

dbres = db_create(&injdb, lube_env, 0) ; // ❶
if (dbres != 0) {
    handle_error(dbres, "Failed to create injector database") ;
}

dbres = injdb->open(injdb, NULL, "injector.db", NULL, DB_BTREE, // ❷
    DB_CREATE, 0) ;
if (dbres != 0) {
    handle_error(dbres, "Injector database open failed") ;
}

```

❶ The database is created in the context of the environment for the Lubrication domain.

❷ The database is stored in a file named `injector.db`.

Berkeley DB provides several choices for the details of how data will be stored. Here we have chosen a Btree for the underlying storage organization. This is common usage in Berkeley DB. Other choices might be better, depending on the details of the application demands for storage and access to the storage.

We represent an instance of a class as a Berkeley DB *cursor*. A cursor specifies a location in a database, can be used to access instance attribute values, and can iterate across instances. We discuss cursors later in this example. For now, it is sufficient to know that they can be considered (roughly speaking, again) as a reference to one or more instances.

As with the ST/MX domain for our microcontroller platform, we convert each class description into a C structure. The C structure for a class provides a convenient way to transfer values back and forth to Berkeley DB and still have direct access to the attributes. Berkeley DB treats key and data values as byte arrays, and variables of a C structure type can be used as a staging area in the transfer to and from the database. The C structure variables also provide convenient access to individual members when database values are held temporarily in memory. For our two classes in this example, the C structures would appear as follows:

```

typedef uint32_t uniqueID ;
typedef char InjModelName[16] ;
typedef char Name[32] ;

struct Injector {
    uniqueID ID ;           // {I}
    unsigned Pressure ;
    bool Dissipation_error ;
    bool Injecting ;
    Name Default_schedule ; // {R1}
    uniqueID Machinery ;   // {R5}
    uniqueID Reservoir ;   // {R3}
    InjModelName Model ;   // {R4}
} ;

struct Machinery {
    uniqueID ID ;
    bool Locked_out ;
} ;

```

Previously, we omitted identifying attributes from the C structure. Because ST/MX used its own identifier for an instance (that is, the pointer address of the instance in memory), we discarded identifying attributes if they were not otherwise used. Using Berkeley DB, we need a key to uniquely identify an instance, so we retain the identifiers in the model and use them as the key to the database storage. This is shown in Figure 10-4.

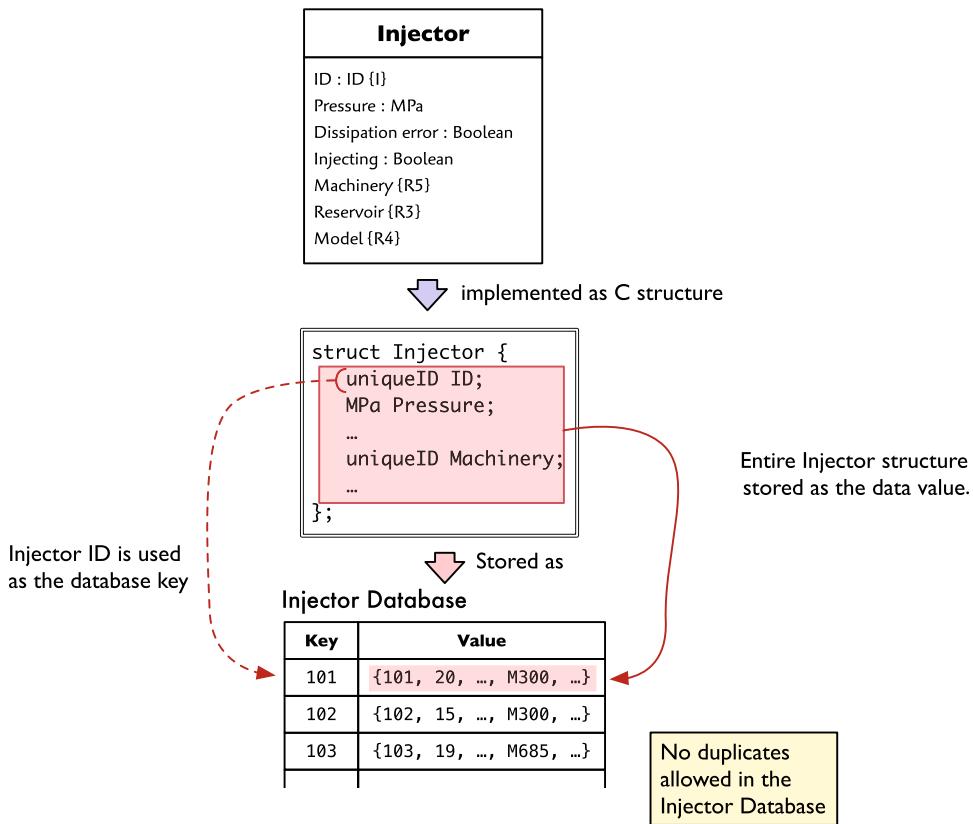


Figure 10-4. Mapping class storage to Berkeley DB

It is also possible for a class to have multiple identifiers. Consider using both a system-supplied identifier and a Customer's e-mail address as identifiers. We map each additional identifier to a Berkeley DB *secondary index*. A secondary index is associated with a primary database. This is shown in Figure 10-5.

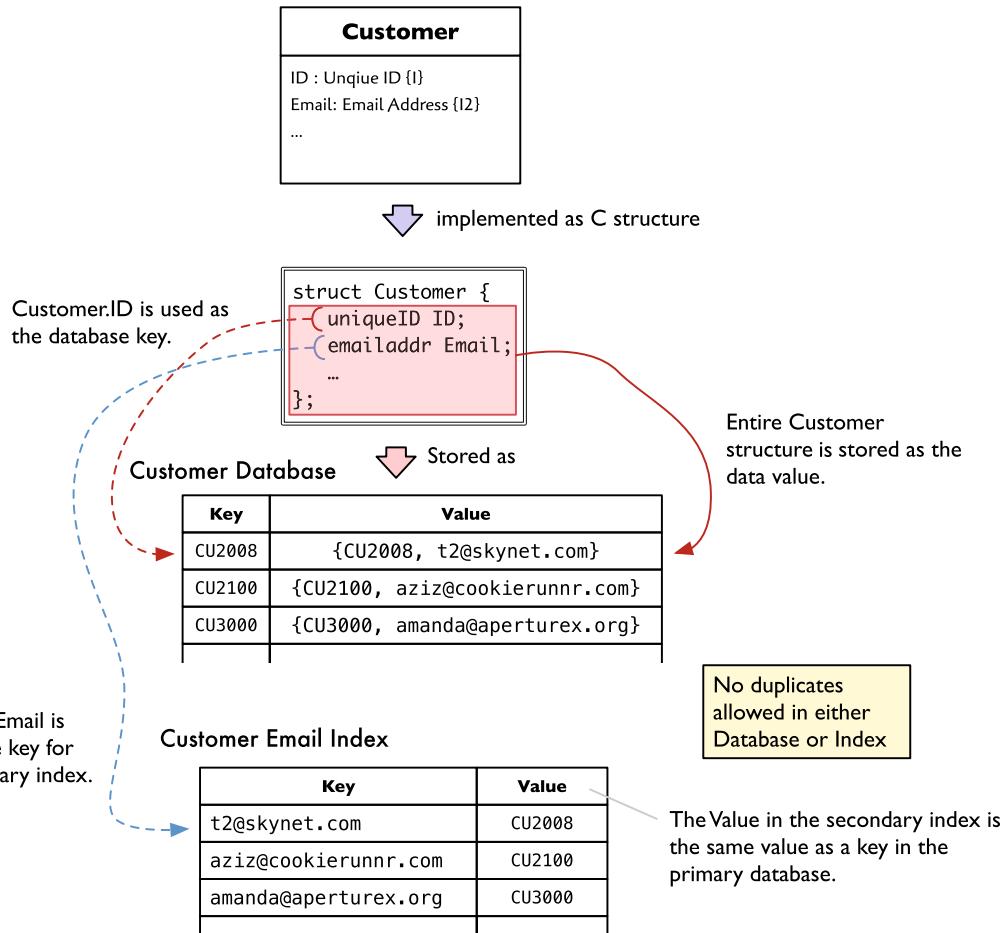
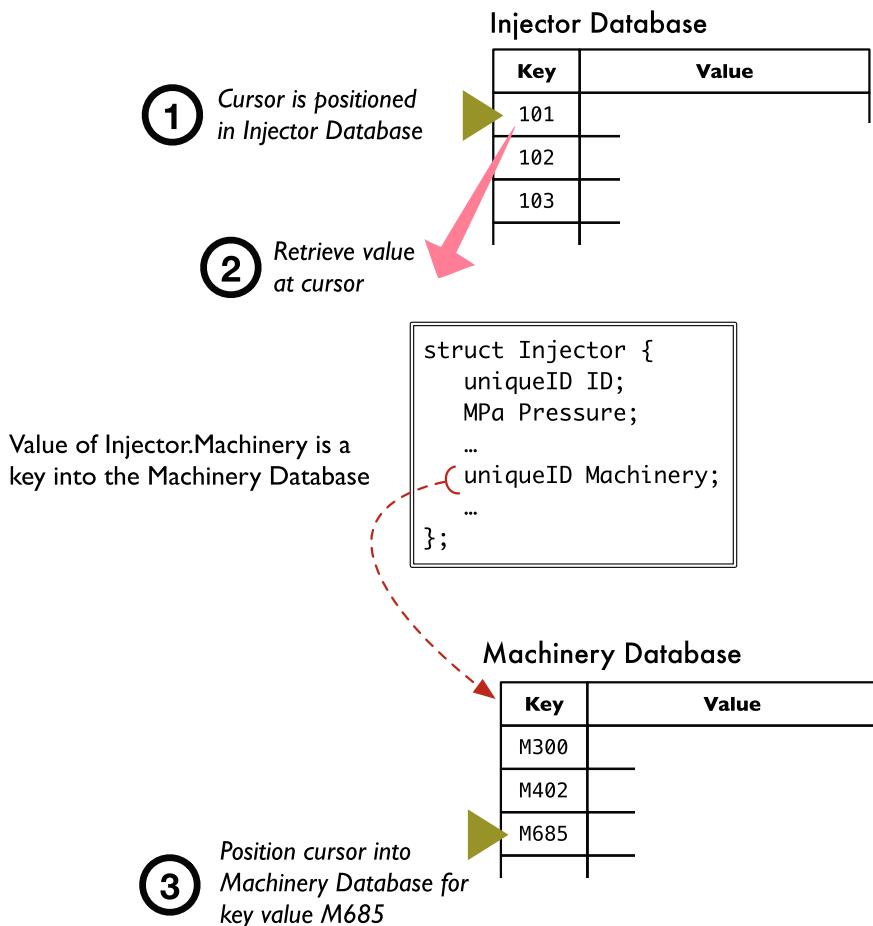


Figure 10-5. Using a secondary index for an alternate identifier

The secondary index is stored like any other database. The key portion of the index record is the value of the alternate identifier. The value portion of the index record is the value of the primary identifier. Berkeley DB arranges it so that each time a record is inserted into the Customer Database, a corresponding record is inserted into the Customer Email Index. Looking up a Customer by e-mail address involves obtaining a record from the Customer Email Index that matches an e-mail address and then using the value portion of that record as the key to look up a record in the Customer Database.

How information is stored to support relationship navigation is another important aspect of domain data management. Returning to the Lubrication domain class diagram fragment, consider navigating association R5 from an instance of Injector to an instance of Machinery. The multiplicity of the association establishes that we expect to obtain exactly one instance of Machinery. Figure 10-6 shows how an attribute from an Injector instance is used as a key into the Machinery database.

**Figure 10-6.** Navigating to a machinery instance

The code for navigating R5 in this direction might appear as follows:

```
struct Injector injinst ;
DBT key ;
DBT value ;

memset(&key, 0, sizeof(key)) ;
memset(&value, 0, sizeof(value)) ;
value.data = &injinst ; // ①
value.ulen = sizeof(injinst) ;
value.flags = DB_DBT_USERMEM ;

// Assume "injcursor" is positioned in the Injector database.
dbres = injcursor->get(injcursor, &key, &value, DB_CURRENT) ;
if (dbres != 0) {
    handle_error(dbres, "Failed to dereference injector cursor") ;
}
```

```

DBC *machcursor = NULL ; // ②
dbres = machdb->cursor(machdb, NULL, &machcursor, 0) ;
if (dbres != 0) {

    handle_error(dbres, "Failed to create machinery cursor") ;
}

memset(&key, 0, sizeof(key)) ; // ③
key.data = &injinst.Machinery ;
key.size = sizeof(injinst.Machinery) ;
memset(&value, 0, sizeof(value)) ;

// Position the Machinery cursor to the record matching the
// value of the Machinery attribute of the Injector instance.
dbres = machcursor->get(machcursor, &key, &value, DB_SET) ;
if (dbres != 0) {
    handle_error(dbres, "Failed to set machinery cursor") ;
}

```

① The attribute values of the Injector instance are retrieved into a local variable of type `struct Injector`.

② Create a cursor into the Machinery database.

③ The key for positioning the cursor is the value of the `Injector.Machinery` attribute.

We assume that we have a cursor into the Injector Database that locates the starting instance for the navigation across R5. Using the cursor, we fetch the value of the database record. In our case, the value is all the data of an Injector structure. Contained within that structure is the Machinery member. The value of the Machinery member is used as a key into the Machinery Database. So in this example, if, when we retrieve the Injector structure value, we find the Machinery member contains the value of M685, we are able to position a cursor into the Machinery Database corresponding to the record whose key is M685. After we have a cursor located at the related instance of Machinery, we can use it to read or update the value portion of the record as needed.

To navigate R5 starting at an instance of Machinery, we must determine how we are going to handle multiple instances of Injector related to a given instance of Machinery. The R5 association is “1..*” on the Injector side, so there can be many records in the Injector database with the same value of the Machinery attribute. The brute-force approach would be to scan the entire Injector Database, reading each record and looking for those records in which the `Injector.Machinery` value matched the value of `Machinery.ID` of our starting instance.

Fortunately, we can do better. Berkeley DB supports two concepts that can be used to navigate a relationship so that the result will yield more than one instance. The idea is to create a secondary index for the referential attributes that formalize a relationship and then to *join* across that secondary index. The secondary index is configured to allow duplicate keys, and the join operation will create a cursor that can access the multiple matching instances. This is shown in Figure 10-7.

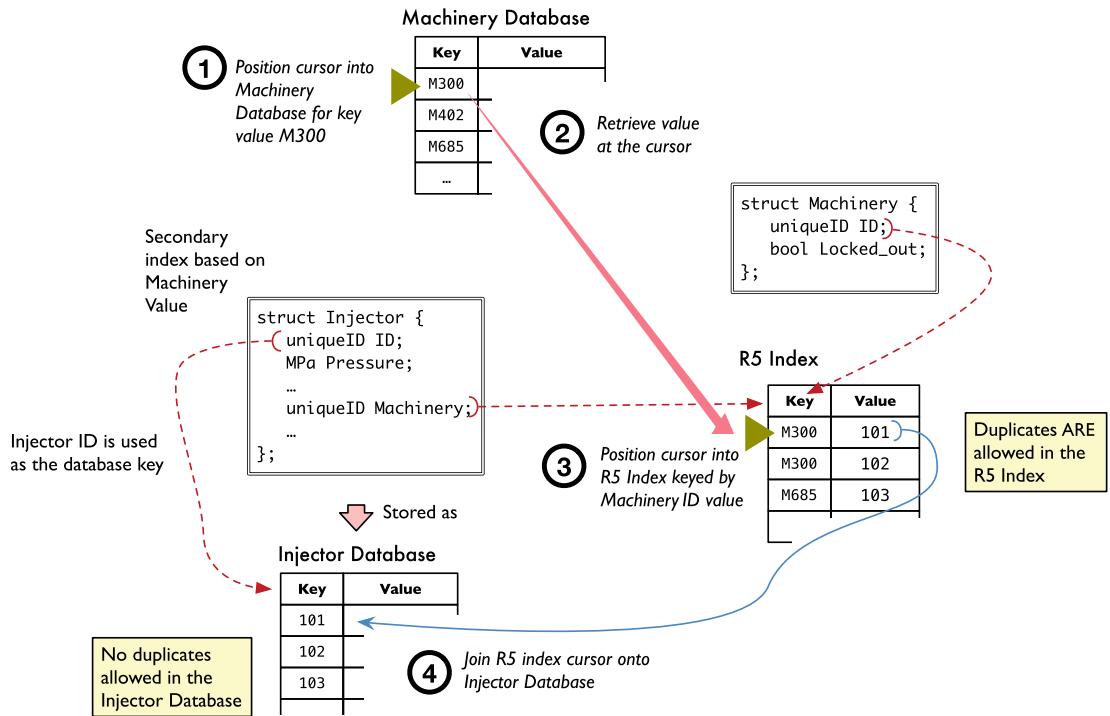


Figure 10-7. Navigating R5 from Machinery to Injector

Here the R5 Index uses `Injector.Machinery` as the key. So we must allow duplicate keys in the index. The value portion of an R5 Index record holds the value of an Injector ID. In the example, the R5 Index shows that the Machinery instance, M300, is lubricated by Injector 101 and Injector 102.

First we must create the R5 Index:

```
dbres = db_create(&R5db, lube_env, 0) ;
if (dbres != 0) {
    handle_error(dbres, "Failed to create R5 index") ;
}
dbres = R5db->set_flags(R5db, DB_DUP | DB_DUPSORT) ; // ❶
if (dbres != 0) {
    handle_error(dbres, "Failed flag setting on R5 index") ;
}
dbres = R5db->open(R5db, NULL, "R5.db", NULL, DB_BTREE, DB_CREATE, 0) ;
if (dbres != 0) {
    handle_error(dbres, "Failed to open R5 index") ;
}
```

❶ We must allow for duplicates. The `1..*` multiplicity of R5 on the Injector side means, in general, many instances of Injector will share the same value of Machinery.

Now we can associate the R5 Index to the Injector Database as a secondary index:

```
dbres = injdb->associate(injdb, NULL, R5db, getMachineryID, 0) ;           // ①
if (dbres != 0) {
    handle_error(dbres, "Failed to associate R5 index to injector") ;
}
```

① The `getMachineryID` function constructs the key for the secondary index.

As R5 Index records are added, we must supply a key for the record. The value portion of the record is already known: it is the key portion of the associated primary database. The key for the R5 Index is just the value of the `Machinery` attribute of the associated Injector instance:

```
static int
getMachineryID( DB *R5db,
    DBT const *pkey,
    DBT const *pdata,
    DBT *skey)
{
    memset(skey, 0, sizeof(*skey)) ;
    skey->data = &((struct Injector *)pdata->data)->Machinery ;           // ①
    skey->size = sizeof(IDvalue) ;
    return 0 ;
}
```

① Here we set the key value for the R5 Index to be the same as the value of the `Machinery` attribute of `Injector`.

To navigate R5 from `Machinery` to `Injector`, we start with a cursor into the `Machinery` Database. Retrieving the value at the cursor, we can use `Machinery.ID` as a key to position a cursor into the R5 Index. In our example for `Machinery`, M300, there are two records in the R5 Index the cursor would access. The R5 Index cursor is then *joined* to the Injector Database. The join operation establishes a cursor into the Injector Database that accesses all the values of `Injector.ID` for the records in the R5 Index where the key is the same value as that referenced by the R5 Index cursor. In our example, the join yields a cursor that can be used to access `Injector` records that have keys of 101 and 102.

```
// Set up data areas to get the value of the Machinery instance attributes.
DBT key ;
memset(&key, 0, sizeof(key)) ;

struct Machinery machinst ;
DBT value ;
memset(&value, 0, sizeof(value)) ;
value.data = &machinst ;
value.size = sizeof(machinst) ;

dbres = machcursor->get(machcursor, &key, &value, DB_CURRENT) ; // ①
if (dbres != 0) {
    handle_error(dbres, "Failed to set injector cursor") ;
}
```

```

DBC *R5cursor = NULL ;
dbres = R5db->cursor(R5db, NULL, &R5cursor, 0) ; // ②
if (dbres != 0) {
    handle_error(dbres, "Failed to create R5 cursor") ;
}

memset(&key, 0, sizeof(key)) ;

key.data = &machinst.ID ;
key.size = sizeof(machinst.ID) ;
memset(&value, 0, sizeof(value)) ;

dbres = R5cursor->get(R5cursor, &key, &value, DB_SET) ; // ③
if (dbres != 0) {
    handle_error(dbres, "Failed to set machinery cursor") ;
}

DBC *joinursors[2] = {
    R5cursor,
    NULL
} ;

DBC *navcursor = NULL ;
dbres = injdb->join(injdb, joinursors, &navcursor, 0) ; // ④
if (dbres != 0) {
    handle_error(dbres, "Failed to join across R5") ;
}

memset(&key, 0, sizeof(key)) ;

struct Injector injinst ; // ⑤
memset(&value, 0, sizeof(value)) ;
value.data = &injinst ;
value.ulen = sizeof(injinst) ;
value.flags = DB_DBT_USERMEM ;

while ((dbres = navcursor->get(navcursor, &key, &value, 0)) == 0) { // ⑥
    printf("Injector ID = %u\nMachinery = %u\n", injinst.ID, injinst.Machinery) ;
}

```

① Assume that `machcursor` has been set to reference a Machinery instance. This function gets the current Machinery instance values. In other words, we dereference the cursor.

② Create a cursor into the R5 secondary index.

③ Set the cursor to the beginning of the entries in the R5 index that match the Machinery instance ID.

④ Join across the R5 cursor instances. This creates a new cursor to access the related Injector instances.

❸ In the iteration at step 6, the Injector attributes are placed in a local variable for convenient access to the attributes.

❹ Iterate over the join cursor to access the related instances of Injector. The get function returns nonzero when all the joined records have been fetched.

Ensuring referential integrity is the final concept we consider. In the ST/MX for our microcontroller target, no provisions were made to check the referential integrity between the instances at runtime. The MX domain assumes that the model gets that right, and the translation provides no additional assurances. This is the customary trade-off made for these types of targets. The code and data required to enforce referential integrity are large enough, and the amount of dynamic activity in the applications deployed on such targets is small enough, that the trade-off is to verify referential integrity by scrupulous model review, simulation, and testing rather than at runtime.

However, we can do better in this particular MX domain. Berkeley DB supports the concept of a *foreign-key index*. In this arrangement, referential attributes can be used as keys in a secondary index, which is then used to restrict adding records unless the key for the record is present in an associated database. This enables us to enforce a limited form of referential integrity checking as a domain executes.

Referring back to our Injector/Machinery example, we would like to make sure that any record added to the Injector Database has a value for *Injector.Machinery* that matches a value of *Machinery.ID* from the Machinery Database. We have already constructed the R5 Index, and so we can enlist it to play another role as a foreign-key index. This is shown in Figure 10-8.

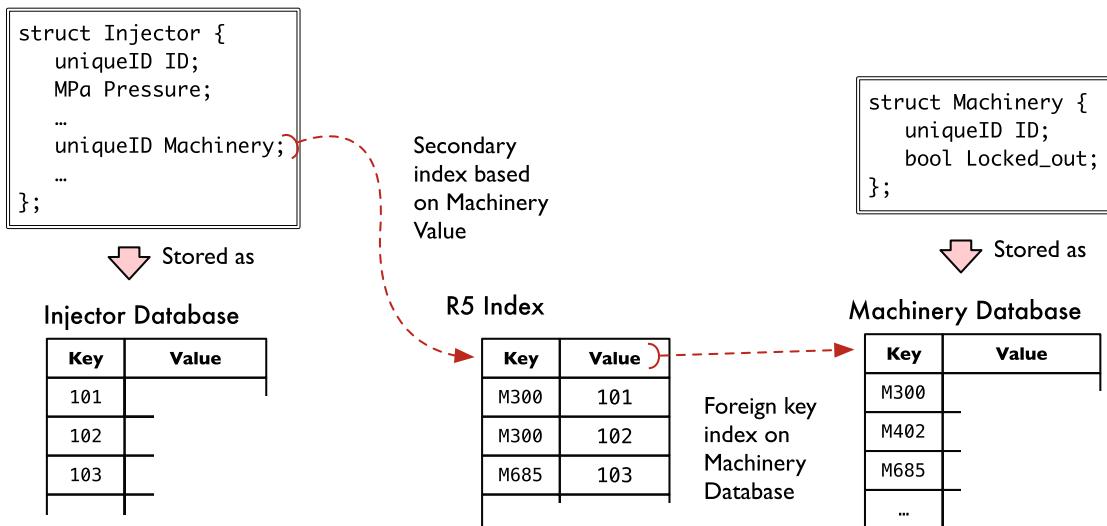


Figure 10-8. Foreign-key index for referential integrity

Each time a record is added to the Injector Database, Berkeley DB will add a record to the R5 Index, just as it would any secondary index. Because the R5 Index is also associated with the Machinery Database as a foreign-key index, the record is added only if its key value matches one of the existing key values in the Machinery Database. If there is no match when adding the record to the R5 Index, the insert fails and the record is not added to the Injector Database either. This behavior ensures that all Injector records refer to Machinery records that exist. Complementary actions are taken with records that are deleted.

We can now associate the R5 Index as a foreign-key index to the Machinery Database:

```

dbres = machdb->associate_foreign(machdb, R5db, NULL, DB_FOREIGN_ABORT) ;
if (dbres != 0) {
    handle_error(dbres, "Failed to associate R5 index as foreign key") ;
}

```

In this section, we have presented only the barest sketch of how Berkeley DB could be employed as a data-management component in an MX domain. Clearly, much more would have to be done to fully develop an MX domain based on these ideas. The code examples shown were specific to the Injector/Machinery example. In an actual MX domain, the code operations would be generalized to work on all classes of a domain. To get a sense of the data required to generalize the data management, we consider a platform-model fragment that deals with the ideas presented here.

Platform-Model Differences

The manner in which we manage class data by using Berkeley DB varies considerably from keeping all the data in primary memory. In the case of ST/MX, we discard identifiers and use pointers to store the necessary information for relationship navigation. The platform model for ST/MX reflects these choices by having classes directly related to references to class instances. This was shown in the platform-model fragment in Chapter 9.

In the Berkeley DB example, we decided to use the identifiers and referential attributes to create key/value databases, secondary indices, and foreign-key indices. This usage mapped conveniently onto facilities provided by Berkeley DB. We don't consider this to be a lucky coincidence. The ideas of identifiers and referential attributes are fundamental to the relational model of data, for which much research and mathematical fundamentals exist and which has proven valuable in many contexts. Berkeley DB is just another example. This example is more interesting because it is applied in the context of a key/value storage mechanism. The platform models for these two cases differ considerably. Figure 10-9 shows a fragment of a platform model that could be used in conjunction with a MX domain managing data with Berkeley DB.

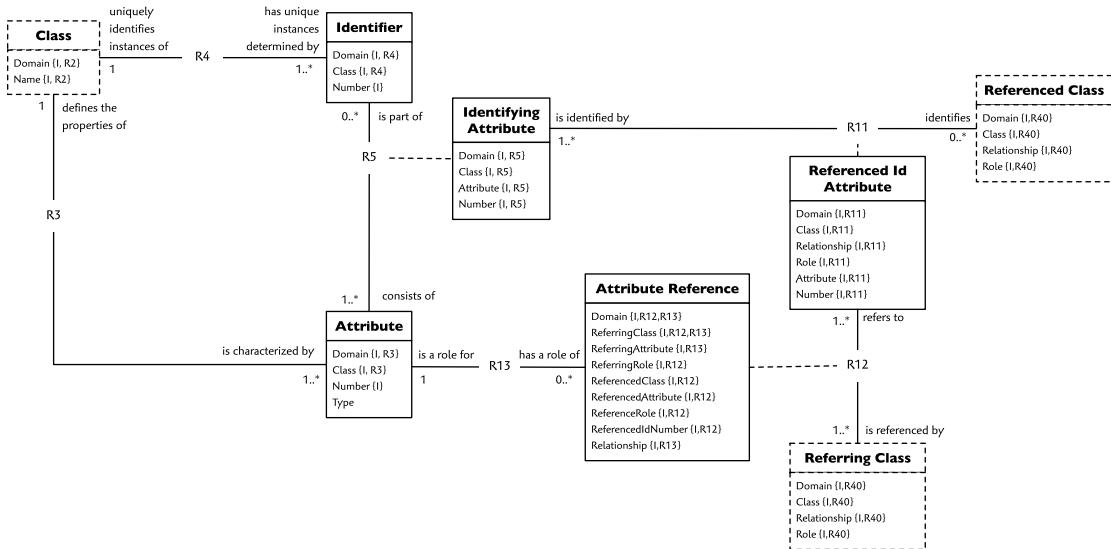


Figure 10-9. Platform model for class identifiers and references

We reiterate again, that, despite the class names, the classes in this model correspond to platform-specific entities. So, the Class class in this model is a platform-specific counterpart to a model-level Class.

The R4 association requires that each Class have at least one Identifier. Identifiers consist of one or more Attributes (R5). An Attribute may be part of more than one Identifier (or even no Identifier). This is not an unusual circumstance, although it does not appear in this example. Taking the Injector/Machinery example, Tables 10-5, 10-6 and 10-7 show a population of these platform-model classes that establishes the classes, identifiers and attributes, respectively. Note that we are showing only the population for the fragment of the example. For the entire domain, there would be many other table rows for the other classes.

Table 10-5. Class Population

Domain	Name
Lubrication	Injector
Lubrication	Machinery

Table 10-6. Identifier Population

Domain	Name	Number
Lubrication	Injector	1
Lubrication	Machinery	1

Table 10-7. Attribute Population

Domain	Class	Name	Type
Lubrication	Injector	ID	ID
Lubrication	Injector	Pressure	MPa
Lubrication	Injector	Dissipation_error	bool
Lubrication	Injector	Injecting	bool
Lubrication	Injector	Default_schedule	ID
Lubrication	Injector	Machinery	ID
Lubrication	Injector	Reservoir	ID
Lubrication	Injector	Model	Name
Lubrication	Machinery	ID	ID
Lubrication	Machinery	Locked_out	Boolean

The population of the Attribute table gives us enough information to define a C structure for each class. So for each Class, we can define a structure in which the members are named the same as the Name attribute value of the Attribute class, and the corresponding data type for the member is given by the Type attribute value.

Each Attribute used in an Identifier results in an instance of R5 and consequently an instance of Identifying Attribute, as shown in Table 10-8.

Table 10-8. Identifying Attribute Population

Domain	Name	Attribute	Number
Lubrication	Injector	ID	1
Lubrication	Machinery	ID	1

We can use the Identifying Attribute data to determine the set of attributes used as the key portion of a Berkeley DB database that would store the class instances. In this example, there is only a single attribute, but the technique can be extended to account for multiple attributes in an identifier. This information would also be used to create any secondary indices required for additional identifiers. In the example, each class has only a single identifier, and it is used as the key for the database storing the class instances. Additional identifiers would show up as Number attribute values other than 1.

In the example model fragment, the Injector class is associated with the Machinery class. In this association, the Injector class serves the role of Referring Class, and the Machinery Class serves the role of Referenced Class. We can distinguish those roles because it is the Injector class that contains the referential attribute referring to an identifier in the Machinery class. Our platform model would be populated as shown in Table 10-9 and Table 10-10.

Table 10-9. Referring Class Population

Domain	Class	Relationship	Role
Lubrication	Injector	R5	Referring

Table 10-10. Referenced Class Population

Domain	Class	Relationship	Role
Lubrication	Machinery	R5	Referenced

An Identifying Attribute may be referenced when it is an identifier for a class serving the role of a Referenced Class in a relationship. Each time that happens, an instance of Referenced ID Attribute is created, as shown by R11. This class represents an identifier being referenced (as opposed to just serving as an identifier for its class). If you have an identifier that is referenced, there is also a referential attribute performing the reference, as shown by R12. The corresponding instance of Attribute Reference represents the referential attribute having the same value as its referenced identifying attribute.

For our example, the Attribute Reference and Referenced ID Attribute populations are shown in Table 10-11 and Table 10-12.

Table 10-11. Attribute Reference Population

Domain	Referring Class	Referring Attribute	Referring Role	Referenced Class
Lubrication	Injector	Machinery	Referring	Machinery
Referenced Attribute	Referenced Role	Referenced ID Number	Relationship	
ID	Referenced	1	R5	

Table 10-12. Referenced ID Attribute Population

Domain	Class	Relationship	Role	Attribute	Number
Lubrication	Machinery	R5	Referenced	ID	1

These tables give us the information we would need to create Berkeley DB secondary indices and foreign-key indices. Using the Attribute Reference information, we must create a secondary index for each Relationship. The primary database would be the one corresponding to the value for the Referring Class attribute (Injector, in this case). The key to the secondary index would be the value of the Referring Attribute attribute (Machinery, in this case). With our example values, we would create a secondary index on the Injector database by using the Machinery attribute of Injector as the key for the secondary index. This was the code sequence we showed previously. There is also sufficient information here to generate the code for the callback function supplying the key for the secondary database. In the preceding example, the callback was named `getMachineryID`. Careful examination of this function shows that the name of the class, the name of the referring attribute, and the type of the referring attribute can be parameterized for code generation.

Using the Referenced ID Attribute information, we would associate the secondary index created for each Relationship as a foreign-key index of the database corresponding to the value of the Class attribute (Machinery, in this case). Again, with our example values, the secondary index created for the R5 relationship would be associated as a foreign-key index to the Machinery class, because it is that class that is referenced by the keys of the secondary index.

Alternate MX Design Discussion

We have briefly and incompletely shown how a MX domain might use a key/value store to manage domain data and provide the basis for supporting other platforms. Expanding on this example, we can enumerate the general points of our approach to building translation technology:

1. Choose the appropriate implementation technology for the class of applications to be deployed. The choice of implementation technology is rarely made in a vacuum. Most project teams have produced systems similar to the one they are undertaking. They have a good idea, even if it is not written down, of the scale and appropriate computing technology that their application will require. One is likely to fail deploying an online web store on a microcontroller. One is just as likely to fail deploying a pacemaker on a laptop.

Implementation choices often are made at the beginning of a project, even before basic requirements are well understood. We suspect that such early, non-requirements-driven choices confuse the activity associated with making decisions for real progress in completing the project.

2. Map out how model-level actions will be implemented in the MX domain. Some model-level operations may be directly supported in the implementation language. Others will require designing mechanisms in the implementation language or incorporating prebuilt components. The model execution rules are the same in all cases. What differs is the manner in which they are implemented and the resulting computational capabilities provided for the domains translated onto the MX domain.

3. Develop a platform model that captures the essential characteristics of the platform and supplies the MX domain with the required data. It is important that the platform model be accessible in a way that supports ad hoc queries, and so some relational-based implementation is easiest.
4. Design and write a DSL to populate the platform model. There are many ways to create language-based solutions. The venerable LALR(1) parser generators are typified by yacc. There are many other parser generators, such as [antlr](#). There are also parser construction techniques based on parsing expression grammars (PEG). The implementation language of the DSL should be chosen for convenience and does not have to be the same as the target language of the MX domain.
5. Write a code generator to produce the code and data that supports the model execution rules and drives the MX domain actions. As we implied in Chapter 9, code generation can be accomplished by template expansion, and most modern languages have libraries to support generating output based on templates. The template expansion queries the populated platform model to find the information needed. The code generator can be a separate part of the translation or, as with pycca, be invoked immediately after the platform model is populated.
6. Write the runtime code of the MX domain itself. This code will use the data produced by the code generator to manage the model data and execution sequencing.

We don't pretend this is a trivial process, but it is one that can be accomplished by an individual or small team managed as its own project. Furthermore, a robust MX domain and the ability to translate models onto it represents a valuable resource with potential for reuse. Writing code that writes code is thus a highly leveraged, if somewhat abstract, undertaking. Rosea, also available through the book site, is another example of this style of translation program in which the target language is Tcl.

Remember that the xUML model execution rules are the same, regardless of the means and mechanisms used to implement them. But you are always free to choose a different way to implement these rules, as long as you can guarantee that those rules work correctly. The goal is the production of a MX domain and its platform-model-based code generator that satisfies the scale and performance requirements of the class of applications you expect to deploy and has the implementation characteristics required in the deployed system.

Our approach emphasizes first establishing the required, often nonfunctional, characteristics of the implementation. Then a platform that supports the model execution rules can be constructed. Finally, we can then translate the logic of the models onto the platform with assurance that the system will have both the means to execute the logic of the models and acceptable performance characteristics when deployed to the field. Ideally, we would prefer that the entire workflow be integrated, automated, modular, and facile to use. We do not see that currently and cannot bank on the future. We must develop software systems in the present, given what is available, and cope with the engineering trade-offs as we encounter them. The demands for increasingly complex software systems will continue to accelerate, and the variety of implementation technologies, both hardware and software, will continue to grow at staggering rates. Our approach represents one attempt to draw upon this astounding advance of technology while maintaining a strict partitioning between logic and technology and with a keen focus on producing quality, working software.

Summary

We discussed the design and implementation of the pycca program itself. Pycca is designed as a language processor that reads DSL statements to populate a platform model and generates code using a template system that queries the populated platform model. It is implemented in the Tcl language.

We presented some size and execution speed measurements of the ALS system on a representative micro-controller platform to show that the resulting memory usage and execution speed are appropriate for our targeted platform.

Finally, we provided a quick overview of a target platform using Berkeley DB. Model execution domain mechanisms for how class instances are held and accessed and how relationships are navigated were mapped onto Berkeley DB facilities. Support for a limited form of referential integrity checking using Berkeley DB facilities was shown. We also presented a fragment of a platform model, demonstrating the information needed to support code generation for the Berkeley DB approach. This demonstrated how different platform implementation approaches would be reflected in distinct platform models.

CHAPTER 11



The Translation Landscape

In this book, we have shown, by a series of examples, how an executable model can be nondestructively translated into a running program. Our translation technique is *one way* to obtain code from models. We do not claim it to be the *only way* to translate models. Nor do we claim it to be the *best way* to translate. The techniques we have presented, like all software engineering processes, have benefits as well as drawbacks. But we have met our goal of producing running code that satisfies the constraints of our target platform by translation of an executable model. We consider that important because, to excerpt from the Agile Manifesto¹:

We have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

In our experience, tools that purport to be comprehensive aren't and do not substitute for individual engineering skill. And regardless of how insightfully an executable model captures the logic of a domain, modeling is a necessary, but not sufficient, step in the development of software. We do value tools, development processes, and well-documented models but, in the spirit of the Agile Manifesto values, we strive foremost for high-quality, working software produced by skilled individuals.

In this chapter, we take a broader view of model translation and examine how this might be accomplished in the general case. We start by presenting a reference workflow for translation. We use the reference workflow as an opportunity to discuss some of the difficulties encountered along the translation path. We show how pycca compares to the reference workflow. Finally, we discuss how our approach to translation might be applied to other target platforms.

A Reference Workflow for xUML Translation

Figure 11-1 gives a broad view of the tasks and work products necessary to translate xUML models into code. This diagram is intended to illustrate all of the key elements that must be present, in some form or other, in any xUML translation system. The pycca approach that we have described is a step in the direction of this idealized workflow. Later in this chapter we will show how the pycca approach fits into this broader context.

¹One of the authors, Stephen Mellor, was an original signatory to the Agile Manifesto.

Reference xUML Translation Workflow

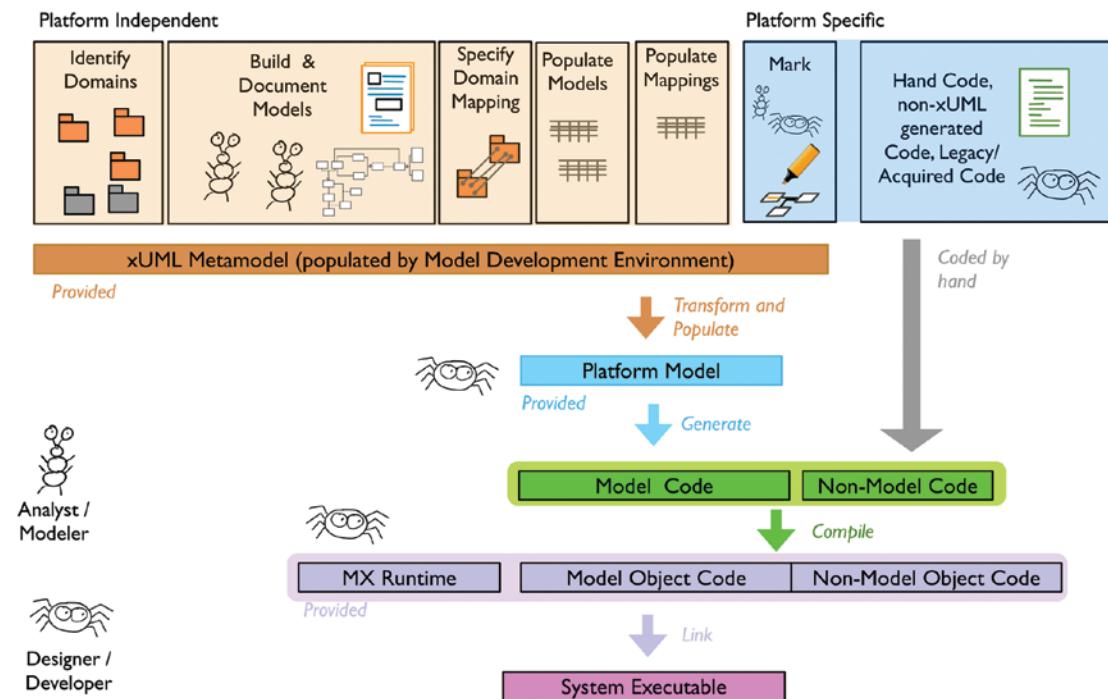


Figure 11-1. Reference translation workflow

The top of the diagram is divided into platform-independent and platform-specific areas. Moving left to right within the platform-independent section, we have a sequence of modeling tasks completed in the indicated order for a given iteration of the system. We do not want to imply that there is only one iteration. We advocate an agile approach to system development, but also carefully distinguish between the *technical process* and the *project management process*. Each iteration through the platform-independent section delivers an internally consistent, fully executable, modeled version of the system. The frequency and content of each iteration, how project team members are allocated to tasks, and how the results of each iteration are used to solicit feedback and additional requirements is part of the project management process. These are important aspects of any real-world project, but something each project must plan for itself. The discussion here is strictly on the technical aspects of the workflow.

Jumping across to the far right of the platform-specific section is the code that must be written by hand, acquired from a third party, previously existing or generated from non-xUML models. To the extent that requirements are known, the non-xUML code can be developed in parallel with the xUML modeling track.

The remaining Mark task must be performed when a consistent fragment of the domain model has been completed. Here, those modeled components that merit special consideration for performance reasons during code generation are annotated. For example, you might indicate which classes have large vs. small populations so that appropriate code can be generated, assuming the target platform model provides facilities to discriminate between these cases.

That covers the top of the diagram. Now let's move downward. As the models are created and edited, they are stored in a *model repository*. This repository is an implementation of the xUML metamodel. As such, the domain models and all populated instances themselves constitute the instance population of the metamodel.

Model marks may also be stored in the repository. This does not compromise the platform independence of the models, as the marks are stored as annotations cross-referenced against the models. So the model repository will permit multiple sets of marks, possibly for different platforms, to be associated with the same domain models.

The platform model defines the way xUML model elements are packaged and reorganized to perform efficiently on the target platform. It is populated with a program that transforms the xUML model elements into corresponding elements of the platform model. The platform model, its populator/transformer program, and the compiled MX runtime constitute the core models-to-code solution for a given class of platform.

So now the platform model has been populated from the marked xUML models extracted from the model repository. From here, code can be generated in the target programming language. This code is then compiled, along with the non-XUML code, to yield a set of object files. These object files are linked together along with the MX runtime to yield a complete system executable.

The MX runtime, you may recall, is the chunk of code that knows how to dispatch events, make state transitions, navigate relationships, and otherwise execute xUML models.

To summarize, the components that must be supplied to enable code generation are as follows:

- A platform model
- A program that populates the platform model from the xUML models
- A program that generates code in a target programming language from the population of the platform model
- A model execution runtime module

Key Challenges

The reference workflow presents an idealistic picture of the principal elements you need to build models and generate code from them. It only partially represents the reality of how model-based systems are built today. Unfortunately, many organizations get swept up in the ideals of model-based software engineering (MBSE) and commit to an expensive and potentially constraining model development workflow. Without prior experience, project teams proceed to acquire model-drawing tools and start making pictures because model diagrams are a key artifact of the workflow. However, modeling is about solving problems in logic, whereas diagrams are simply a means of capturing the problem solution in a form that directly contributes to producing a software solution. Later in the project, theory and reality often collide in an expensive cloud of disappointment. The result is filling the gap between the diagrams and the code by using one of the approaches discussed in Chapter 1 (namely, gradually or abruptly), neither of which achieve much benefit from modeling.

Inexperienced project teams should seek help in training the team to undertake such a qualitatively different way of building software. Yet, it is hard to convince project teams that have successfully built software using conventional techniques that modeling and translation requires skills and thought processes they may not possess and that manipulating diagrams in a modeling tool will not provide those skills. In this section, we flag some of these ugly realities for you. Considering each stage of the translation workflow, we point out the common difficulties and challenges that we have seen in our many years of model-based development.

We do not mean to imply that modeling and translating are an immature or unworkable development approach just because difficulties exist in the state of the practice. Many project teams produce high-quality systems by using modeling and translation. We could easily enumerate an even longer list of problems with conventional software development techniques. We have purposely not done so because Internet sites document the horrors discovered in real-world programs more comprehensively than we could here and because highlighting problems in other approaches does nothing to solve our immediate concerns. Software development approaches are subject to passing technical fads, just like any other complex human undertaking. We won't indulge in Pollyannaism aimed to convince you that everything is smooth and easy. We don't believe there is any silver bullet that will slay the software werewolf (See the papers "No Silver Bullet—Essence and Accident in Software Engineering" and "No Silver Bullet Refired" by Fredrick P. Brooks,

Jr.) Modeling and translation represent real, practical progress in grappling with the beast. But forewarned is forearmed, and we hope our concerns and frustrations with current practices in modeling and translation will save some project teams from their own disappointments.

Identify Domains

The first step in developing the software system is to identify all of the required domains. For translation purposes, we are interested in which domains are modeled and which ones are provided as code. Those provided as code could be hand coded, acquired from a third party, available as legacy code or existing libraries, or generated from models in a non-xUML language. Ultimately, a diagram called a *domain chart* is produced that inventories all domains and their dependencies.

The key challenge in this step is factoring the domains correctly. This is not a problem fixable with tools. You need experience and skill and the right approach to thinking about the problem.

When it comes to dividing up a large system, it is easy to lean on the crutch of familiar platform technology such as library, task, and hardware boundaries. These boundaries are just so tangible! But experienced developers know that technology changes all the time, and tangibility and frailty go hand in hand. “Don’t worry, the hardware design is frozen...” Right. More to the point, a platform-specific partitioning yields platform-specific models, which defeats much of the purpose of modeling.

Even if platform boundaries are successfully ignored, it is also common to split up a system into functional rather than subject-matter categories. This mistake makes it difficult to develop solid class models, which are the foundation of each domain. And let’s not even get started on pointy-headed non-engineering boundaries such as managerial or political.

Here are a few guidelines that may help. For any prospective domain chart, consider these two questions:

1. Would the domain chart change in *any* way if the platform technology changed?
It should not. For example, what if you have two CPUs instead of one? What if you are using tasks and threads? What if you are running the system on a distributed platform?
2. Is there any class that must simultaneously exist in one or more domains?
There should not be any. A subject matter is defined by a vocabulary of classes and relationships. Such a class-based, rather than function-based partitioning then demands that each class live in only one domain. In the ALS, the Injector pressure “exists” in both SIO and Lubrication; however, it means something entirely different in each domain. In SIO, there is nothing special about pressure; it is just data coming from an input point that needs to be scaled and converted. But in the Lubrication domain, pressure really is pressure, and there, the technology to gather and convert is of no consequence. Similarly, the Injector class lives only in the Lubrication domain. For example, if we had divided the Lubrication domain into Injecting Normally and Diagnostic Injection functions, each would need to share the Injector class.

Consider the ALS domain chart from Chapter 6. It is entirely platform independent. You could put all those domains in a single task, as we have done with pycca, or spread them across multiple processors or threads. The hardware onto which the software is deployed has no impact on the domain chart itself. Note that the domains are defined by what they *know* rather than what they *do*. The Lubrication domain knows about the lubrication equipment and how lubrication works. It knows nothing about systematic handling of alarms, it knows nothing about signal processing, and it knows nothing about user-interface technology. All the functionality in the Lubrication domain follows from what it knows about the way equipment must be lubricated. The SIO domain knows about signals and actuators, but assumes no specific meaning for any of the data in the signals or controls over the external world.

Ultimately, the best way to test a domain chart is to build a certain percentage of the class models inside each of the modeled domains as a validation exercise. During this process, it is common to rethink the partitioning and end up refactoring the domain chart. Had you not identified Signal I/O as a domain on the first pass, for example, you may have found that the state models for lubrication got excessively complex, constantly polling for new data. Each physical device with sensor-driven attributes would need to replicate the same polling or event-response patterns. When you see the same cookie-cutter patterns replicating across multiple state machines in a domain, it is usually a sign that a service domain is missing or some sort of domain refactoring needs to be done.

It is often the case that the coded or non-xUML domains are not complete as domains and require some kind of wrapper that may or may not be modeled. So additional modeling may be required in these domains, which must be interfaced with the supplied code.

Build and Document the Models

For each domain modeled in xUML, one or more analysts work together with subject-matter experts to model whatever subject matter is relevant to that domain. Figure 11-2 illustrates this concept.

**Special communication and documentation skills
are required to analyze a subject matter.**

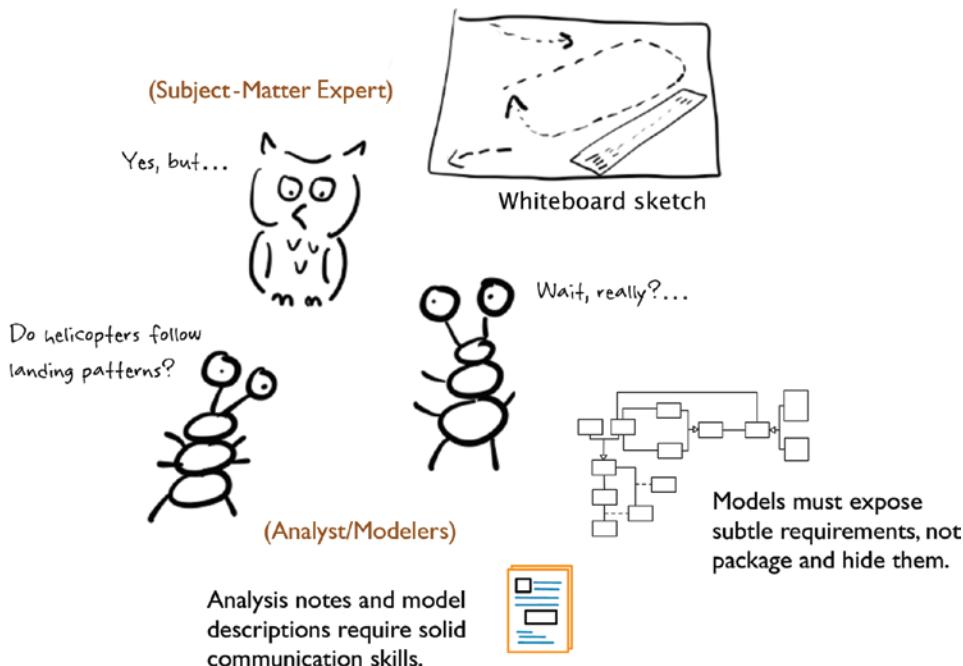


Figure 11-2. The right talent is essential to successful modeling.

The sheer number of challenges in this phase require an entire book—*Executable UML: How to Build Class Models*, by Leon Starr (Prentice-Hall, 2001)—but here are a few of the most significant:

- Get the *right talent* working together.
- Enter and edit the models *productively*.
- *Usefully* document the analysis and models.

Use the Right Modeling Talent

Three distinct talents are necessary to build models: analysis, modeling, and subject-matter expertise. Rarely do all three reside in the same individual to an adequate degree, so it is generally necessary to get teams working together. The first mistake most projects make is to just grab a bunch of programmers, because those are the people who happen to be around, and assign them to modeling duty. This is a mistake because most programmers lack the proper skills, knowledge, or inclination to build models.

Analysis is the ability to ask questions about a subject matter, and to write, draw, and otherwise describe the subject matter. This requires good communication skills. The analyst is always trying to break down problems and find the interesting cases that almost never happen but must be handled correctly when they do. The analyst must have solid communication and presentation skills to elicit expert feedback. Most important, the analyst must discard preconceptions about a subject matter and routinely expose his or her ignorance of a topic in order to elicit the fine details from subject-matter experts. These are not skills typically cultivated by programmers. Programmers like to show up to the party with patterns and libraries in hand, ready to write code.

Modeling is the ability to take the analysis and formalize it in an objective, testable way. This requires skill at putting the platform-independent building blocks of classes, relationships, states, and so forth together. This sounds a bit like programming, but there is one key difference. Whereas a programmer is constantly coming up with clever ways to package things so that they work efficiently, the modeler is typically unpacking ideas so as to expose the critical and subtle differences. The modeler is concerned about efficiency, in the sense that an idea should be expressed as simply as possible and as clearly as possible, while simultaneously handling all the subtle cases uncovered in the analysis. Whereas the programmer often takes pride in having as few elements as possible in a program, packaging and hiding along the way, the analyst takes pride in unpacking and exposing complexity. That extra class, attribute, or relationship that reveals an overlooked case is a source of pride for the analyst.

Finally, subject-matter experts need not know anything about modeling. They just need to know their subject matter well and have the inclination and time available to explain it to the analyst/modelers. And, of course, it doesn't hurt if they can read the models to verify that they are being understood.

On real projects, these three skills come in overlapping combinations with various individuals. The necessity is to get enough of all three skill sets together to get the models done correctly. The best results occur when you pair up a good analyst with a good modeler and give them access to at least one subject-matter expert.

Enter and Edit the Models Productively

As models are developed, they are ideally entered into a graphical editor. The editor stores the non-graphical model information in a repository. To make the model information available in a useful manner to downstream tools, such as those for code generation, the repository design should be based on the xUML metamodel.

Okay, so how hard can it be to draw boxes and arrows? Based on the current crop of model-editing tools, it turns out, surprisingly hard and painful. To see what we mean, compare the task of entering and editing graphical models to that of writing code in your favorite editor. Today's programmer expects a lot from a code editor. Features such as autocomplete, refactoring, expand/collapse, search/replace, documentation lookup, syntax checking, keyword highlighting, and visual diff allow a programmer to move nimbly through a large, complex code base. The productivity a programmer experiences is not even in the same league as that of their modeler counterpart using a graphical editor.

Compare textual vs. graphical layout, for example. While a programmer effortlessly indents, closes braces, expands and collapses a function block, a modeler limps along pushing rectangles and arrows, trying to click connection points and shifting text labels a few pixels this way and that. You get carpal tunnel just watching a large model being rearranged.

To be fair, some graphical editors take some of the pixel-shifting drudgery away and perform varying degrees of automatic layout. Unfortunately, the geometric algorithms are still rather simplistic, yielding clunky layouts. This may sound like nitpicking, but geometric layout to an experienced modeler is every bit as important as text layout is to a programmer. Imagine the outrage of a programmer working with a bizarre indenting and line-wrapping scheme, with open and closed parentheses being scattered to the wind. Experienced programmers are quite careful about how they organize their code. Experienced modelers are no different.

More pertinent to translation is the storage of non-graphical model data. Many tools store this information in either a file using some sort of interchange format or in a database or both. Our concern is the way that data is organized. Ideally, the organization of the database, or definition of the file format, should be derived directly from a model of the executable language (a metamodel). The language we are using is xUML. Most model editors use the UML standard, which brings with it a lot of stuff we don't need. There is a standard for model interchange called XMI. But, again, it is intended to support the full UML standard. So it is filled with lots of content that we don't need. Our preference would be to have model-level data in a commonly used implementation form on which we could directly operate, such as a population of a relational database. What we really need is a tool that uses an xUML metamodel as the basis for model storage.

In the meantime, what is the modeler to do? It really depends on whether you are taking a pycca-like approach or using a comprehensive draw tool/translation environment. In the pycca case, you are going to encode your translation decisions by hand, anyway. Here you want a model editor that will let you specify xUML elements nicely, such as verb phrases on associations, identifiers, and referential attribute tags. You want something that isn't trying to be too smart and preventing you from drawing a model the way you want it to look. Umlet, OmniGraffle, Visio, and DrawExpress are some choices. None of these tools stores any model semantics, but if you are proceeding to text, by hand, it doesn't matter. On the other hand, if you are using a tool such as BridgePoint, you don't have any choice. You have to live with the model editor provided, warts and all.

Usefully Document the Models

Models are not self-documenting (and neither is code, despite some programmer assertions to the contrary). They do expose application logic in a formalism that excludes implementation aspects. But what is the reasoning behind that logic? Why is this class or attribute necessary? Why is relationship R3 unconditional on the many side? The why's need to be answered clearly.

For the model to be useful, it must be adequately documented. In fact, we would go even further and say that good model descriptions are more important than the model graphic. Only in the descriptions can the basis of the abstraction for the model be explained. The model graphics present the model in an information-dense form, but do so by using mnemonics such as class names, attribute names, and relationship phrases. Names in a model use common natural language words that require additional explanation to give them the meanings needed to build a precise vocabulary for the subject matter of the model. The precise meanings of the mnemonics are contained only in the descriptions, which need both text and informal diagrams. Without a precise meaning for the model terms, model readers will simply apply their own notions to the model based on their own understanding of the natural language words used in the model diagram. Because natural language relies so much on context for its precise meaning, confusion is the usual result.

Useful documentation goes way beyond short text descriptions. It includes the numerous analysis notes, copies of whiteboard discussions, and informal (non-model) sketches, mathematical formulas, and other supporting technical notes that are developed in the process of building the models. Without real-world context, models quickly lose their value. Without the inclusion, integration, and maintenance of adequate documentation, there isn't much point in bothering to model in the first place.

Unfortunately, graphical model editors tend to attach slots to various model elements that are filled in like forms. This sounds reasonable at first. But in practice, it results in lower-quality model documentation. Modelers tend to fill in the slots in a robotic fashion without giving thought to the big picture. This tendency is frequently reinforced by project development rules that insist every slot be filled in like a Bingo card, resulting in little regard for the content. The descriptions tend to restate what is already obvious on the diagram. It is our experience that informal non-model diagrams are essential elements of good model descriptions. But most tools facilitate or accept only text. Finally, it is all too easy to look at a diagram and not see the underlying documentation or to delete model elements, inadvertently trashing pages of descriptions. True, good model documentation is hard work, but it is the foundation of a long-lived knowledge base; it is valuable intellectual property, essential for training project team members and the basis for future maintenance.

It is helpful to consider two distinct approaches in practice that have arisen to deal with code documentation. In one approach, documentation is placed in stylized comments directly in the source code file, and a software tool is used to extract and format the documentation. Literate programs, a concept introduced by Donald Knuth, takes the opposite view. In a literate program, the source code is placed into the document, and a software tool is used to extract and reorganize the code from the document. Both techniques can be used to produce good documentation. We prefer the literate program approach because it allows more flexibility for the order of presentation and for including supporting materials. The worked-out examples available as supplementary material for the book are literate programs.

We develop our models using tools primarily intended to produce high-quality documents and import the model graphics as simply another diagram in that description. We do not want to imply that we think the model graphic is not useful. It is extraordinarily useful, but only when combined with the background and abstractions detailed in the model descriptions.

Specify Domain Mapping

A dependency between two domains on the domain chart is referred to as a *bridge* in xUML. The domain on the tail of an arrow plays the role of client, to the domain on the arrow side playing the role of a service. The direction of the arrow simply means that one domain exists for the express purpose of satisfying the needs of another. In the ALS domain chart in Chapter 6, for example, the Lubrication domain needs some way of interacting with the physical world, which is satisfied by the SIO domain.

The domain chart says nothing about what data and interactions are exchanged on a bridge. Furthermore, the models of one domain are designed to be unaware of any particular model structure in any other domain. This notion of domain-level encapsulation is a key organizational principle in xUML. That said, certain model elements in one domain will have corresponding elements across each connected bridge. These must be mapped together somehow, without muddying the interior of any one domain with knowledge of another domain's model content.

The bridging technique we demonstrated is based on explicitly invoking actions on an external entity. Referring back to the Lubrication domain of Chapter 6, when an Injector began to apply lubricant, it made an explicit call to the Inject operation of the SIO external entity. The activity of the Start injection state makes clear its delegation of the physical injection process at a precise point in the model execution.

For some situations, sprinkling the model action language with external entity invocations detracts from the intent of the model and inhibits its potential for reuse. Consider the Reservoir state model from Chapter 6. Each of the state activities makes explicit calls to an ALARM external entity to reflect the status of the Reservoir. The intent of the actions is to mark the state of the Reservoir and signal other instances in the domain at key boundary conditions (that is, when things are Normal or when the Reservoir is Empty). The ALS system is required to post alarms at these critical junctures in the Reservoir life cycle, but the ALARM entity invocations are intrusive and have nothing to do with the essentials of the Reservoir class. Further, what is considered worthy of an alarm is subject to considerable requirements churn. Project often start with the notion of *alarm everything* only to find out that overwhelming amounts of information cannot be digested into practical actions.

We would prefer to specify how the ALARM entity is notified apart from the action language of state activities. We want to say that after the LOW state activity of the Reservoir class executes, the *Set lube level low* operation is to be invoked on the ALARM entity. The action language for the LOW activity would no longer include the explicit invocation of *Set lube level low*, and the translation mechanism would arrange for the ALARM invocation based on our specification.

This type of bridging arrangement is called *implicit bridging*. The idea has much in common with the concepts of aspect-oriented programming (AOP). AOP has an entire vocabulary to describe the approach, which we do not discuss here.

Implicit bridging is intended to reduce the coupling between domains by removing at least some of the explicit external entity interactions. By providing a level of indirection in specifying where the external entity operations are invoked, the potential for reuse of the Lubrication domain is enhanced, because the dependency on an ALARM domain is no longer explicitly encoded in the domain actions. The invocation of ALARM operations could also be applied to other domains that had not been built with any alarm concepts in mind.

We see the process of defining explicit or implicit bridges between domains as the same. It is still necessary to mark, map, and populate the bridges. The difference is that implicit bridging uses a means apart from the action language of a state activity to specify when the bridge operation is invoked. The implicit means of specifying the bridge operations is also done relative to generic (metamodel) entities. In this example, we specified the ALARM interactions relative to state transitions. Other model-level elements are also candidates, such as creating or deleting class instances or signaling events.

Sadly, we see no substantial tool support for mapping and populating bridges. Explicit bridge operations are present, but we are not aware of any implicit bridging support. It is a significant complication to the model translation process that has not been overcome. We demonstrated in Chapter 8 that, despite the lack of a complete theory, it is nonetheless possible to tackle bridging methodically using nothing more than a spreadsheet application.

Populate the Models

It is common to document scenarios with real instances as you develop your models. The diagram of air traffic controllers introduced in Chapter 2, for example, or the lubrication configurations shown in Chapter 6 are quite typical.

Once an iteration of the models is complete, it is helpful to construct initial populations for it. In fact, you usually create multiple populations for various scenarios. In the ALS example, you could create one or two populations for testing and then another two or three for anticipated real-world configurations of the ALS. By careful selection of the attribute data values that influence the execution paths through the activities, test populations can be constructed to force a larger trace of execution than might happen with a population delivered for deployment.

Tools tend to focus on the development of the models, with the instance populations as an afterthought. Few tools provide quality facilities for specifying initial instance populations. Again, our advice is to put them in spreadsheets. The examples we have presented had small initial instance populations. When the number of initial instances is small, almost any strategy to handle them will work. However, as the number grows, it becomes more difficult to manage the populations.

We believe that initial instance populations should be specified entirely by the values of the attributes of the model classes. In our examples, all the model-level descriptions of initial instance populations were accomplished by treating the class as a table and specifying instances as rows of values. By contrast, we dislike the approach of having to compose action language to explicitly create class instances and assign the attribute values of the initial instance population. This approach is tedious and error prone, and does not scale well. Worse yet, because many attributes are given the same value, the motivation to use action language variables to hold repeated values is hard to resist. The values of these variables change during runtime and, unlike explicit attribute values, are difficult to verify by simple inspection.

Another problem with the action language approach is that it excludes non-modelers from providing the initialization data. Consider an electric utility control system that requires a large amount of data to describe the transmission lines, substations, transformers, and other equipment required to distribute electric power. The control system will use the connectivity implied by the transmission specification data to route and otherwise manage the distribution of electric power. The utility staff that updates and manages the topology of the distribution grid is also the primary resource for obtaining correct instance population values. A database management system is critical at this scale, so translation tooling should either take initial instance population data directly from the database or at least provide an interface to which query results from a database can be included as a domain's initial instance population.

For instance populations of an intermediate size, fashioning a domain-specific language (DSL) can help solve the instance management problem. In the SIO domain example, the initial instance population consisted of 50 instances of the various classes that specified the properties of 11 I/O Points. Directly specifying the attribute values was not an onerous task because the population was small.

But imagine a case with 100 I/O Points (not an excessive number even for a microcontroller-based system). Specifying values for all the attributes of the approximately 500 class instances now becomes a significant task. To reduce the amount of detailed knowledge required to directly populate the class instances, we could construct a DSL such as the following:

```

population SIO test_population_1 {
    Continuous_Input_Point inj1_pressure {
        group pressure_group
        thresholds {
            max_pressure
            inh4_pressure
        }
        scale ihn4_scaling
    }

    Conversion_Group pressure_group {
        period 500 ms
        converter main_converter
    }

    Point_Threshold max_pressure {
        direction rising
        limit 20 MPa
        successive_over 2
        successive_under 3
    }

    Point_Scale ihn4_scaling {
        multiply 100
        divide 27
        intercept 100
    }

    # ... and other similarly styled declarations
}

```

Such languages should be declarative in nature and minimize any knowledge of the class model required to specify the attribute values. In this example, the fact that there are several generalization relationships in the model is hidden by focusing on the attributes of the leaf subclasses. In contrast to an

action language, there is no order of execution. Variables are not used, and the output of the language processing is direct input to the translation mechanism. Such population languages are usually better suited to service domains in which the potential for reuse is higher than a domain whose subject matter is tied closely to a particular application. We recommend the rule of three: when a domain is reused for the third time, stop and invest some effort in making that reuse more productive.

Although constructing DSL's is not a large undertaking, project teams must determine whether the investment has good return. Many tools are available to construct *small language* processors such as this. The DSL program can define a syntax and then use well-established techniques of lexical and parser generators to form the core of the processing. Alternatively, some dynamic scripting languages such as Tcl or Python are well suited to use for DSLs (sometimes called an internal DSL), and then the parser of the scripting language itself can be used for parsing the DSL. Language-based solutions to application configuration problems have a long tradition in software, and existing techniques can be employed for specifying an initial instance population.

Populate the Domain Mappings

The population of the mappings between domains that form the basis of how a bridge is realized may be known at translation time (at the time the code is generated from the model), at runtime, or a hybrid of both.

If the model elements involved in the mapping do not vary during the running of the system, the population of those mappings is specified before translation. When the initial instance population is determined, the mapping tables can be filled in. This was the case of the bridge mapping between the Lubrication and SIO domains in Chapter 8. For example, the mapping for the *Inject* and *Stop injecting* external entity operations was onto the *Write Point* domain operation of SIO for I/O Points that were defined as part of the initial instance population and did not vary as the system ran.

If the model elements of the domain mapping are created or deleted during the running of the system, it is not possible to populate the domain mappings before translation. In this case, the bridge code will populate the mapping at runtime. The bridge will include operations that map the creating and deleting of model elements in the client domain to the corresponding elements in the service domain and record the mapping for later use. Although the mapping table heading still describes the information that needs to be collected and maintained, it is not possible to specify the values of the tables before translation. The bridge code implementation is usually more complicated, as it must know when the creation and deletion of model elements occur, and uses more-sophisticated data structures to handle the dynamic nature of the half tables themselves.

Hybrid mappings can also occur. In this case, some of the mapping population is known at translation time, and the remainder occurs at runtime. In this case, the bridge code is patterned after the dynamic case but may start with a non-empty half-table population to be augmented as the system runs.

Again, we find no substantial support for this activity in available tools, so we recommend spreadsheets as an easy and available way to enter and display the required mapping data. The example mapping tables from Chapter 8 can serve as a guide. A specialized database application based on a spreadsheet metaphor could also work well for larger mappings.

When populating the domain mappings, three data sets are being managed:

- Model element populations from the client domain
- Model element populations from the service domain
- Mapping between the two populations of model elements

With that many things in play, the challenge is that you might find out that things don't quite match up. Filling in data values to the mapping tables is the ultimate check on whether you have the populations and domain mappings right. Don't be surprised if you need to revise things. For example, you may find an instance in the client domain that has no corresponding instance in the service domain and have to adjust your initial instance population for the service domain. You might also find that the way an instance is identified in a service domain can't be determined from the elements of the client domain mapping, and

you might have to adjust the domain mappings themselves. To avoid too many surprises, it is advisable for the modelers of a client and service domain to communicate routinely. This ongoing communication is essential to ensure that the assumptions and dependencies given to the service domain by the client domain are accounted for. As the models progress, the conversation can be extended to ensure that proper domain mappings exist to realize all of the dependencies.

Marking

A complete iteration of the platform-independent models is marked with platform-specific features prior to translation. For illustration purposes, it is helpful to imagine marking as involving a transparent sheet laid over the top of the models and a box full of markers of different colors. The specific colors and number of markers available depends on the features provided in the platform model. For example, one platform model may prefer that you distinguish instance populations based on whether they max out at 1, 10, or 1 million for any given class so that the appropriate storage and access mechanism may be selected. Using the *max population* marker provided by that platform model, you mark classes accordingly. Note, however, that you aren't marking the classes directly, thus the transparent sheet laid on top. That way, the models remain platform independent. It is only the marked-up sheet that has the marks. Because marks themselves can be abstracted as annotations, they can be stored in the metamodel. In fact, the same set of models can be marked differently for each potential target platform. Just swap the marking sheets.

Marking has two key challenges. There must be adequate types of marks available to tune the models for translation. There must also be a way to specify them without permanently embedding them into the models themselves.

In reality, we don't use markers or transparent sheets, though these might still make a good user-interface design metaphor in some model-editing environment. Instead, there may simply be a text file that provides a keyword for each mark type and a list of affected model elements. In the pycca approach, the marking features are mixed into the DSL used to specify the design. While the markings on a particular model are interpreted to create a platform-specific implementation, they are just a particular kind of annotation that can be stored in the metamodel.

Care is taken to ensure that marks reference model elements but are never permanently mixed into the models themselves. This is in stark contrast to the elaboration style of producing code from a model, in which implementation artifacts are indiscriminately blended into the models. This style unnecessarily destroys platform independence in the process of delivering a system.

The xUML Metamodel

Now we move downward underneath the platform-independent section to consider the xUML metamodel. A *metamodel* is just an ordinary model whose subject matter happens to be the modeling language itself. Whereas our air traffic control domain model captured classes such as Duty Station, Air Traffic Controller, and Control Zone, a model of xUML would have classes such as Class, Association, Attribute, State, and so forth.

A metamodel serves as a formal definition of the language it is modeling. Thus, an xUML metamodel would serve as the ultimate definition of the xUML language. *Executable UML: A Foundation for Model-Driven Architecture* does a fine job of informally describing xUML, and it should serve as the key input to an xUML metamodel. As with any subject matter, any ambiguity, inconsistency, or incompleteness in the informal description of the modeling language should be resolved by the completed metamodel.

Just for fun, let's assume that such an xUML model exists. Here's how you could use it.

Imagine that you build a database schema based on such a metamodel. Assume that the database perfectly imposes all constraints defined in the metamodel. Now let's say that you have built an application model, the Air Traffic Control application, let's say. You should be able to populate the metamodel database with your application model. For example, you would create instances of Class: Air Traffic Controller, Control Zone, Duty Station, and so forth. You would proceed to create instances of State, Transition, Attribute, and so forth until your entire ATC domain was instantiated in the metamodel.

If you found that you were unable to populate the metamodel database without triggering errors, you would know that your models were incorrect. (We're assuming a perfect metamodel database implementation here.) For example, xUML requires that each transition exiting a state have a different event specification. So if you have already inserted a transition out of state X on event A, and you attempt to define another transition out of X with the same event A, you should get an error and the edit operation should fail.

This means that if a model has been successfully entered into the xUML metamodel database, it is linguistically correct. The model may have runtime problems or may be incomplete, but at least it doesn't break any of the modeling language rules. So, with respect to a set of application xUML models, a metamodel would serve the same role as a programming language grammar, and the metamodel schema populator would play the same role as a parser.

Because the metamodel should also capture instance and instance value information, you should be able to enter not just your model, but your model's population as well. In the case of the ATC model, you could enter a population (or in fact, multiple populations) for the same domain model.

You can also use the metamodel to design a DSL for storing a model and its population in text files. Each metamodel element might correspond to a DSL statement, for example.

So the model serves as a formal definition of the modeling language, as a reference for the design of any model repository or model file format. It also means that any downstream translation or model-processing tools should be built to process any structures that can be inserted in a metamodel database.

Now for the bad news. There is no *complete* xUML metamodel currently in existence (as far as we know). On the bright side, there are many partial metamodels, and we anticipate that a complete one should be available in the near future. The lack of a complete, accepted xUML metamodel hasn't stopped us from having translation tools. But it would certainly be nice to sort this out.

In the past, a number of tool-specific metamodels have been built. One open source xUML tool, BridgePoint, for example, is built around an xUML metamodel. Unfortunately, it has been modified away from the xUML as described in *Executable UML* to support a variety of tool-specific features. It goes under the name *xtUML*. Another tool-based metamodel called *iUML* also varies considerably from the xUML definition.

Our effort is titled *miUML*. The intention of this metamodel is to be open source and as tool independent as possible, with a focus on both *Executable UML* and the Shlaer-Mellor methodology from which it originates. It also strives to be based on firm foundations of relational theory with an orthogonal type system for attributes. It has been implemented partially with a relational database schema and editing functions in Postgres. Additionally, the models are thoroughly documented for your reading pleasure. You can download it from www.executableuml.org. As of this writing, this model is incomplete with respect to polymorphism and data types. But it has strong constraints for identifiers and referential attributes.

The OMG's UML standard publishes a UML metamodel. It's not much use in the context of xUML for the following reasons. xUML uses a subset of the UML notation. xUML is built on relational foundations and has special rules for the use of identifiers and referential attributes. xUML has built-in executable semantics that the greater UML lacks. UML does define a framework called *Foundational UML* (fUML) for defining executable UMLs. At this point, no attempt has been made to develop an fUML definition of xUML. It could probably be done, but it is not clear that it would be worth the effort.

In addition to having an agreed-upon metamodel, it is also important to have agreed-upon implementation representations of the metamodel. To be tool independent and support a more modular approach, there needs to be specific and easily accessible implementation infrastructure for populating and querying the metamodel. One such approach, as we have mentioned, is to use a relational database management system. These are typically queried using SQL and so allow for the ad hoc queries that are necessary to make effective use of the metamodel structure. Unfortunately, SQL exists as many vendor-specific dialects, and we are faced with having to support several representations, depending on the underlying database system. Other interchange representations may also be used, such as XML or JSON. The important point is that the representation must completely cover all aspects of the metamodel and provide a convenient starting point where an implementation of modeling tools can access the metamodel population.

The xUML Language

While we are on the topic of challenges, we need to discuss a few revolving around the modeling language itself.

There are several reasons for our choice of xUML. The primary reasons are that it was designed to support the translation of models on the widest variety of platforms: everything from cloud distributed to tightly embedded. This means that models built in xUML can be widely reused. The data and execution rules of the language are based on relational data theory, finite state automata, and data-flow execution rather than object-oriented foundations. There is no presumption that the target programming language be object-oriented, though there is certainly nothing prohibiting or hampering such an implementation. The language is designed to be as lean as possible. Rather than having lots of complex rules and model elements, there are only a small number of building blocks that can be assembled strategically to tackle considerable real-world complexity. This again is largely a consequence of its mathematical foundations. The benefit of this property is twofold. From an analysis perspective, language simplicity means the modeling artifact fades into the background, putting the emphasis on the subject matter being modeled. It is much more difficult to spot a subtle flaw in application logic if the modeling notation itself is intruding with its own complexity. From an execution and translation perspective, it is easier to run the models and generate code, because there are so few elements and rules to implement. Furthermore, it is easier to devise model execution platforms that guarantee the model execution rules work on diverse and challenging platforms.

But if you are coming from an object-oriented programming perspective, as most of the greater UML is practiced, you will find some aspects of xUML to be a bit alien. There are no hidden identifiers and object links, for example. The data is simply organized in a way to enforce the connectivity of the instances.

And instead of relying on a separate language to express constraints, such as the Object Management Group's (OMG) Object Constraint Language (OCL), constraints are built directly into the class and relationship data. By declaring an identifier of a Die on a semiconductor Wafer to be Grid Location {I} + Wafer {I, R}, we have effectively declared that you cannot have two Die at the same grid location on the same Wafer. Identifiers and referential attributes can be combined in various ways to form a set of declarative model constraints without requiring extra "check the constraint" code to be generated. These built-in mechanisms cover most constraint circumstances with the exceptions handled by light annotation. In fact, we see OCL as an artifact of the lack of inherent constraints in object-oriented programming languages, filling that gap by constructing syntax trappings on predicate logic. Sadly, so many examples of the uses of OCL are based on poor models and only highlight the need for better modeling to capture the problem logic rather than explicit constraints.

Unfortunately, there is no official, widely accepted standard defining xUML. The best we have is an informal standard consisting of various white papers describing the Shlaer-Mellor method, the predecessor to xUML, and the *Executable UML* book. This serves as a fine guide for the analyst/modeler, but leaves a bit open for interpretation when it comes to building model execution platforms and translation tools. This of course, is where the previously mentioned metamodel fills the gap.

Fortunately, the modeling language is simple enough that there is general agreement, within the Shlaer-Mellor, xUML community on most of its class and state modeling features. There is however, some variance with regard to how activities are modeled.

Action Language

The manner in which algorithmic computations are specified is one of the more challenging areas in the translation workflow. The syntax and semantics of an action language involve many trade-offs. Because writing action language appears, superficially, to be like writing program code and because, as programmers, we have definite opinions about how best to write program code, action language syntax is subject to the extremes of a programmer's personal taste.

But we do not consider writing action language to be *coding* in the usual sense. Coding is directed at making a computing machine operate in a specific manner to achieve a desired result. By contrast, action language is directed at specifying the algorithmic processing of a domain model void of implementation technology considerations. A large fraction of what model activities do is directly related to model-level concepts, such as signaling events, navigating relationships, and updating attributes. So we do not consider it to be a *more abstract* version of program code but rather a detailed specification of operations supplied by the formalism of the model execution rules. Clearly, action languages must be transformable into program code. But we consider that transformation to be a discontinuous operation directed by mapping functions and not a process of gradually elaborating the action language statements to some lower form of abstraction.

In other words, the notion of starting out with fuzzy actions written in natural language and then inserting more and more code-like fragments to tighten it down as a means to get closer to implementation is the antithesis of our approach. As sure as plaque will rot your teeth, elaboration erodes away the platform independence of a domain model until it is neither a good model nor a good implementation. Instead, we aim to map model-level operations to whatever constructs and idioms are appropriate to the target programming language, be it object-oriented, functional, scripting, or otherwise.

Several action languages consistent with xUML have been defined, but because of the many ways that algorithms may be stated, we see little convergence in the syntax. For example, BridgePoint (xtUML) Object Action Language (OAL) and iUML's Action Specification Language (ASL) have semantics that match closely to our approach and have working implementations. Other action languages have been proposed, such as Shlaer-Mellor Action Language (Small) and Starr's Concise Relational Action Language (Scrall). See www.executableuml.org for links to xUML-compatible action languages, but have not seen any production-ready translators.

On the wider front, Alf (Action Language for Foundational UML) has been established as a general-purpose UML standard. We don't find Alf appropriate to our approach because it covers conventional UML semantics and carries the burden of object-oriented programming language constructs upon which UML was based. We see, for example, the Alf constructs for namespaces, public/private declarations, collection data types, inheritance, and so forth as implementation concepts. In our platform-independent context, these constructs offer more confusion than clarity for specifying the model-level processing of application logic.

Desirable Characteristics of an Action Language

To be truly platform independent, a model should not specify a particular sequence of computation unless that sequence must be enforced on every potential target platform. Here is an example of an arbitrary computation sequence:

```
y = scale(x)
z = filter(i1, i2, ... in)
result = y + z
```

Depending on the implementation, actions 1 and 2 could be reversed or executed concurrently. Action 3, on the other hand, must wait for both actions 1 and 2 to be complete. If the action sequence as written is intended to indicate a required sequence of computation, the model is unnecessarily limiting implementation choices. This breaks the principle that the model must specify only what is required on all potential platforms. By breaking that principle, the model loses a bit of credibility. "What else in the model might I ignore?" the implementor now begins to think!

Consider Figure 11-3.

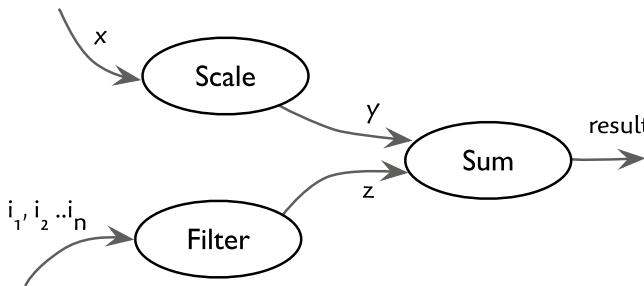


Figure 11-3. No arbitrary sequencing in data-flow representation

Each action is represented as a circle, and we interpret each action to be runnable when all of its inputs are available. Action 3 must therefore wait until both actions 1 and 2 have produced output. This data-flow view of computation eliminates the statement of arbitrary sequencing. This is why the data flow is our fundamental view of algorithmic processing in xUML.

Unfortunately, text representations are faster and easier to edit than graphical representations of data flows. In Chapter 2, we showed a data-flow diagram of the Logging In activity from the Air Traffic Control model. The difficulties of dealing with data-flow graphics and specification of the data-flow processes have meant that all translation schemes of which we are aware use a text-based language to specify activities.

The use of a text-based action language does not preclude having data flow semantics in the language. Early attempts at action language, such as Small, used a UNIX pipe style of syntax to indicate data flows. The simple actions in Small would appear as follows:

```
x | scale > ~y
Input(all).i | filter > ~z
(~y, ~z) | sum
```

The text can then be processed in such a way as to yield a data-flow representation as input to the code-generation process.

And, in fact, the original step-by-step text formulation with steps 1–3 is even okay if it is understood that a data-flow analysis will determine the implementation sequencing. The important thing is that adequate information is present to construct an intermediate data-flow representation before proceeding with translation.

Unfortunately, constructing a MX domain that actually takes advantage of this fine-grained concurrency is nontrivial. Code that can deal with concurrency and map it onto the available processors to achieve parallel execution without mucking everything up is difficult problem. As programming language support for this type of fine-grained parallelism becomes available (consider Go), we hope better use of the inherent concurrency of the model execution can be achieved as actual parallel execution.

Object-oriented and other common programming idioms impose more implementation biases in action language syntax. Most action languages are patterned after programming languages and never fully escape their roots. Because statically typed, usually object-oriented, languages are the most common translation targets, most action languages have telltale signs of these programming languages baked into their design. For example, the use of an object's address in memory as an implementation-generated identifier is common. An instance reference serves as a thinly disguised pointer, which, sadly, encourages modelers to think in those terms.

There is an unnecessary distinction between sets of instance references and a single instance reference, as if sets somehow cannot contain a single member. We suspect the differences are more related to the ease at which most programming languages can hold a single pointer value in a simple variable of a language-supplied type compared to a collection of pointers that requires a more costly implementation construct. The translation knows, by the nature of the instance selection, when the outcome can be more than one instance. The translation mechanism should then be responsible for choosing the optimal mechanism

to hold the result and not press that decision back onto the modeler. On some occasions, the result of an instance selection must be limited to being less than the full selected set—for example, selecting an arbitrary instance from a set of otherwise identical instances or limiting the selection to a given number based on sorting criteria. But those operations limit the cardinality of the selected set, the result of which is still a set. The limiting operation does not dictate a particular way to store the result.

The remnants of imperative programming language constructs are evidenced in the fact that action languages require excessive explicit iteration over class instances. In xUML, the instances of a class form a set and so *set at a time* operations would be a desirable replacement for explicit iteration. For example, to give a percentage price discount on items in a store, we would prefer to say something like

```
Item().Price *= percentDiscount // (). selects all instances
```

rather than what is more common:

```
items := select all instances of Item
foreach i in items {
    i.Price := i.Price * percentDiscount
}
```

or worse yet:

```
items := select all instances of Item
for count ranging 1..items.count() {           // assuming indices start at 1
    item[count].Price = item[count].Price * percentDiscount
}
```

Model-level actions can be applied to sets. Consider signaling a torpedo recall for a certain model of torpedo. Again, we would like to say something like

```
// Send a recall event to each Torpedo having a given design specification
Recall -> Torpedo( Spec:specToRecall )
```

compared to this:

```
torpsToRecall := select many from Torpedo where (Spec == specToRecall)
foreach torp in torpsToRecall {
    signal Recall to torp
}
```

Allowing an instance to be created without setting a value for each of its attributes is another example of implementation seeping into an action language. Allowing attributes not to have a value assigned at creation time follows from the assumption that an instance is a block of memory allocated out of a pool or heap. It is an untrustworthy arrangement because different execution paths through the activities could potentially leave one or more attributes uninitialized and containing whatever random bit pattern might already be stored in the instance memory. An action language should not make assumptions about instance data being stored in memory. An MX domain may choose any method of data management that meets the needs of the targeted class of applications, such as a relational database management system (RDMS), a key/value pair database, a flat file, or even an EEPROM.

The only way to keep an action language executable, yet free of any implementation assumptions or biases, is to build it on mathematical foundations. [Relational algebra](#) is a branch of mathematics, extended from set theory, functions, and predicate logic that can serve as a basis for action language operations.

We do not suggest that actions should be written as pure relational algebraic or predicate logic expressions. Rather, action language operations on class instances should be derivable from relational algebra and provide operations that clearly express model-level execution concepts. From that basis, the translation mechanism can then transform the operations into the programming constructs that integrate with the data management services provided by the MX domain. A lean and orthogonal basis for action language operations makes the transformations to a wide variety of other data management techniques easier to accomplish.

Application models need a way to express and process real-world data such as pressure, temperature, video images, geophysical coordinates, stock prices, and so forth. But most action languages do not acknowledge value types of anything other than basic, supported system types or a type matching that of a class structure. This eliminates values that might be composed of the join of two classes. For example, if we wished to have a report of the launch type of our torpedoes, most action languages force us into awkward constructs such as

```
torps := select all instances of Torpedo
foreach thisTorp in torps {
    itsSpec := select Torpedo Spec related to torp across R5
    UI.REPORT(torpedo : thisTorp.Torpedo ID, type: itsSpec.Launch Type)
}
```

where we must compute the join by navigating associations, rather than this:

```
UI.REPORT( Torpedo().(ID, /R5/Torpedo Spec.Launch Type) )
// Join is implied
```

or this:

```
UI.REPORT( Torpedo join R5 Torpedo Spec.(ID, Launch Type) )
```

In fact, relational theory permits a value to have a type of arbitrary complexity. For any given data structure, the modeler must decide whether exposing that structure contributes to the domain analysis or distracts from it.

Consider, for example, a domain that tracks geographical boundaries. Certainly, there will be a need for some kind of two-dimensional *Point* type. You will need to store Points as attributes of classes. A Point is a representation of a two-dimensional vector. The algebra of 2D vectors is well understood, and the domain activities will need the ability to add points, multiply points by a scalar value, and determine the distance between two points. An action language should support having a Point data type along with the algebraic operations on it. The algebra of the user-defined type should be fully integrated into the action language syntax.

Encapsulation of any internal components of a user data type is critical. Even if we need to obtain the value of the components of a Point, we should not know how the components are internally represented. We may choose to hold the 2D point in Cartesian coordinates. However, if there is significant circular symmetry in the application, using polar coordinates would be a better choice for the implementation of Point operations. The two representations are equivalent, and each can be converted to the other.

We need to be able to define, presumably outside the action language itself, how user types would be implemented. We do not want to be forced to define Point type attributes as two scalar numeric attributes and have to expose to the model the details of the operations on the two attributes. This would lend nothing to the analysis of a domain dealing with geographic boundaries as its subject matter. Neither do we think data-type operations should be defined as external entity operations. Sprinkling external entity invocations into the domain activities solely to accomplish abstract data-type operations compounds the lack of user-defined data type support with action language clutter.

Furthermore, we probably don't want to implement the algebraic operations on the Point type at all. We would rather call an existing library that is better coded and tested than what we would write in action language. The lack of good support for user-defined data types becomes more intractable when considering larger algebraic structures such as matrix algebra, which would be required of a domain dealing with three-dimensional graphics.

Translation Considerations

Referring back to the translation workflow, we work our way downward from the metamodel and produce code. The reference translation workflow shows the generation of model code as a two-step process. First, the populated metamodel is transformed into a population of the platform model. Second, the populated platform model is transformed into the model code. The generation of model code from a populated metamodel could be accomplished in a single step, and many translation schemes operate in that manner. We prefer a two-step transformation for its added flexibility. Exposing a distinct platform model also allows a close examination of how well the model fits the platform needs of an application:

- The platform model is most dependent on the class of applications and the demands those applications place on computing technology to run them. We would like to see the development of platform models that account for different mechanisms to manage domain data, execution concurrency, and other key aspects of how model execution rules are realized by implementation technology appropriate to a particular class of applications.
- The code generation is most dependent on the chosen implementation language and the interface details of the MX domain runtime functions.

Our conjecture is that the separation of model code generation into two steps would facilitate a modular approach as well as allowing easier support for both differing implementation mechanisms of the model execution rules and a larger variety of implementation languages. We have no direct evidence to support that conjecture. However, for conventional computer language compilers, automatic generation of a code generator from a machine description is an established technique. By analogy to language compilers, we think such ideas may be applicable to automatically generating the transformation from a platform-model population to implementation language code based on a description of the platform-model characteristics. We think separating the transformation of the metamodel population into a platform-model population as a separate phase from generating model code from the platform-model population would contribute the flexibility to try such an approach. This is clearly a subject for additional research.

Adding steps to the overall workflow unfortunately increases the difficulty of tracing between models and executing code. When something goes wrong in the execution of a modeled and translated program, it can be arduous to work back up through the layers of transformation. In general, the transformation from code to models is not reversible without additional information. Conventional language compilers accomplish the feat by recording debugging information about how the generated code is related to the source files and symbols of the program. Such backward traceability is desirable, but difficult to achieve both in terms of recording the information during the translation processing and in having a program to interpret it.

Even well-established techniques sometimes don't work. For example, C supports adding `#line` directives to source code to indicate that the code originated from a file other than the one being compiled. Pyccca supports adding these directives so that error messages and source debugging can reference the ultimate pyccca source file (call it `sio.pyccca`), rather than the generated C file. But sometimes source-level debuggers, particularly those used for microcontrollers, do not admit to the fact that C source code might be contained in a file that does not end in a `.c` suffix.

Execution tracing embedded into the generated code can help in some of these situations. It is particularly useful to trace state machine dispatch information. Tracing other aspects, such as method invocations, is sometimes available. Unfortunately, emitting trace information at runtime is intrusive on the execution speed of the program as well as its size. If the tracing is removed for deployment, as often it must be, then we are still faced with how to handle tracing execution faults that occur after deployment. Although this problem might be solved for a specific tool chain, we do not see a more general solution.

The Pycca Workflow

The reference workflow just described represents the ideal situation. By contrast, the pycca translation workflow, shown in Figure 11-4, is less ideal, but it is a working solution that yields real code for a certain class of platform and applications. The software is open source, and you can download and use it today. Realistically, we expect you'll spend a bit of time reading through the online documentation, configuring your environment, studying the online examples, and experimenting a bit. So maybe you'll use it tomorrow. But everything you need is, in fact, available now.

Pycca Translation Workflow

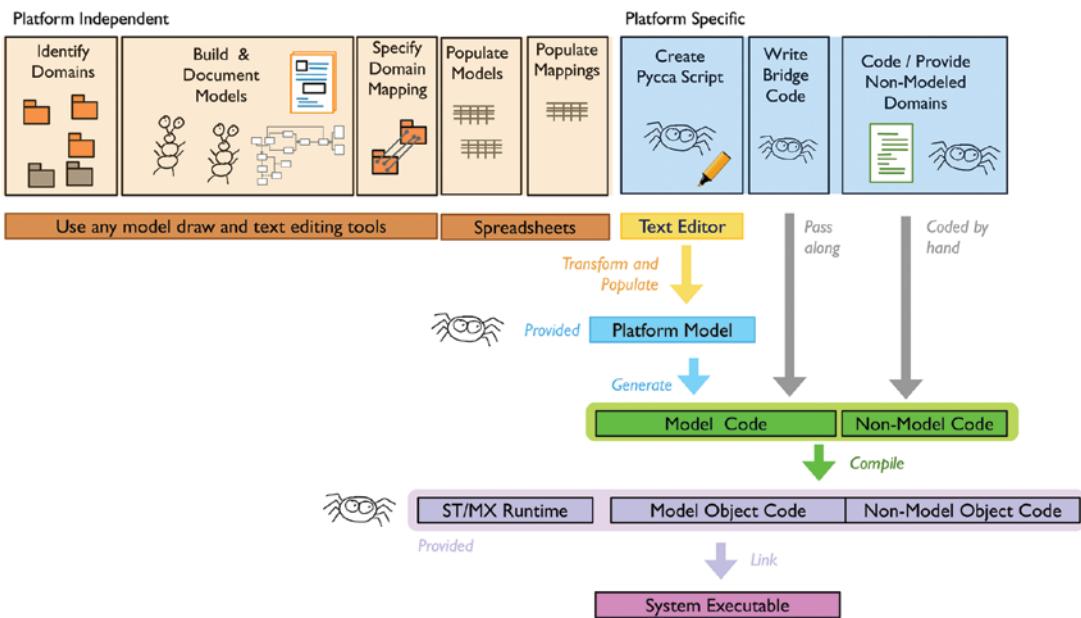


Figure 11-4. Translation workflow using pycca

A pycca translation starts with one or more completed domain models with fully specified domain bridges. This includes, at a minimum, all three facets of the model:

- A class model of the domain data including full descriptions of the classes, attributes, relationships, and data types.
- An initial instance population.
- The state models of all active classes expressed as both diagrams and tables.
- The action language for all the domain activities, including the state activities, any class methods, or domain operations. You may use pseudo-code, but it must cover all model-level actions and fully specify all algorithmic computations.
- If the domain has any external entities defined for it, bridge markings and mappings must also be supplied.

For the reference workflow, we assume that the transformations are carried out programmatically. The transforming programs would extract the necessary information from a metamodel population. As an example, consider the decisions that must be made regarding how association navigation is implemented. The multiplicity of the association is determined directly from its definition. By parsing the action language of the activities and walking the abstract syntax tree (AST), we can examine the operations performed on an association. This will tell us if, during runtime, any instances of the association are created or destroyed. Similarly, analyzing the AST of the domain activities can determine whether an association is navigated in both directions. This information can be recorded and used to make programmatic decisions as to how the class data is organized to support navigating the association.

Human Roles in Translation

For the pycca workflow, human intervention is required. The model artifacts may exist in any format or media because it will all be processed manually. For example, the state tables and instance population could be entered into a spreadsheet. The class models may be drawn in any tool you like.

The *human translator* must perform three tasks:

- Analyze the model to decide how model-level constructs are mapped onto the implementation constructs provided by pycca. The decisions involved in mapping model-level constructs were discussed in Chapter 3. Pycca provides a set of choices for how model-level constructs can be implemented. For example, association linkage is configured based on the multiplicity, dynamic aspects of the association, and whether the association is navigated in a particular direction. References to support relationship navigation need to be included only for those paths navigated by the action language. Frequently, both directions are not needed. Some attributes will be elided. For example, attributes used strictly for identification purposes and not otherwise referenced in the action language need not be included in the implementation class because the model execution domain for pycca uses a platform-specific identifier (the address of the class instance in memory). All of this information can be found by reading the action language and annotating the model graphic for each attribute that is read or updated and noting the direction that each relationship is navigated.
- Transcribe the model structure into the pycca DSL. In Chapter 4, we showed the translation of the Air Traffic Control model. Each of the three facets of the model have a correspondence in the pycca DSL. State models have the most direct representation in pycca and correspond closely to the information contained in the state transition matrix. Pycca class definitions will vary from those on the class model by virtue of the translation decisions made in the first step. Here you do get to think in terms of what C structure implements the model-level intent of a modeled class. Of the three facets, the state activities require the most transformation effort. It is necessary to formulate the semantics of the action language into C code by using the provided pycca macros to perform model-level actions. The transformation of action language to C usually falls into two categories: model-level operations and flow control or computation. If you are transforming a model-level action such as signal event, or select instance x related to instance y, use a provided macro. Expression evaluation and flow of control are directly represented in the C code.
- Write the bridge code. In Chapter 8, we demonstrated how bridging is accomplished. After the domain mapping is done, data structures and code must be written to implement the intent of the domain mapping. In simple cases, this can be done with an array to map identifiers from one domain to another, and a small piece of code to pass control into the target domain. Pycca provides a *portal* into domains that can be used to execute simple model-level actions from outside of a domain. Frequently, the domains involved in the bridge will provide domain operations to assist in the bridging.

Note that the marking task from the reference workflow is missing. It is, in fact, folded into the pycca script-writing task. There is no distinct marking going on. While writing the script, the human translator makes implementation choices based on the action language.

Our experience shows that translations benefit greatly when done by someone other than the person doing the domain analysis. We find this true even in more automated translation schemes. This allows project teams to specialize between those people more interested and skilled in application analysis and those more interested and skilled in implementation technology. As previously stated, great proficiency in both skills is rarely found in the same person.

Even if the same person serves multiple roles, making a sharp distinction between roles sets the boundaries for the thought processes involved with each. The discipline of “thinking inside the box” is, in this case, a highly valued skill. The process of translating a domain model is an ideal way to become familiar with the logic of the domain in a way that merely reading the analysis materials does not accomplish. Project teams that practice *pair programming* will find pairing a domain analyst with a domain translator yields a higher-quality output and a deeper understanding of model and implementation for both team members.

Running pycca on the resulting source performs the two transformations shown in the workflow, populating a platform model and generating code from the populated platform model. We discussed the pycca platform model in the preceding chapter and gave a brief overview of the processing performed by pycca. Processing by pycca yields C code files, which can then be integrated with bridge code and the ST/MX domain code to produce a running program.

Pycca vs. the Big Tool Approach

The pycca approach is oriented to the implementation side of the complete development workflow. The most important reason for that orientation is to be able to obtain the required characteristics of the implementation. As beneficial as we consider modeling and translating to be, modeled and translated programs that do not achieve the expected implementation characteristics do not see the light of day as real products. Project teams that cannot see a clear path to an implementation that meets all their nonfunctional requirements are not inclined to undertake a model/translate development approach.

A vast array of implementation technologies is available in the wider computing world. From operating systems to database management systems to programming languages to web frameworks, programmers have written many useful implementation components. As programmers, we delight in finding common and reusable processing that can be implemented as a single body of code and still satisfy a larger scope of need. We want to be in a position to make better use of available implementation components when translating models if we are to achieve the required implementation characteristics and obtain the true benefits of modeling.

It is unrealistic to expect tool vendors to supply translation mechanisms that can use this variety of implementation technology, especially when the technology choices for a project must be very specific to meet broader corporate or team needs. Commercial concerns alone dictate that tool vendors attempt to make their products appeal to the broadest possible set of customers. The drive is to the lowest common denominator, as that yields the largest potential market. Tooling usually trails the leading edge of implementation concepts and technology substantially due to the rapid pace of platform technology growth and the limited market each technology presents.

Project teams have many of their own constraints to address. Target programming language support is one example. It is common for tool vendors to support C, C++ (and to say C/C++ in one breath as if they were the same language) and perhaps Java or Ada. A dizzying number of programming languages are available, and more are always being developed. There may be compelling reasons for a project to use a particular implementation language. Those reasons can be based on legacy system integration, target platform restrictions, or knowledge and convenience of the development staff. For example, Python is a popular language and has been used for implementing some rather large systems, but we are not aware of any tool vendors targeting it.

Another example is the use of a database management system. There can be compelling reasons to use them, because a central data store can be a powerful integration point for other programs of a larger system or for ad hoc queries that may be needed to satisfy future requirements. But we are not aware of substantial tool vendor support in this area either.

We believe that the separation between modeling and translation, which we have described as the separation between logic and implementation technology, is a fundamental concept that should allow us to choose the implementation technology most appropriate to our immediate engineering needs and still obtain all the benefits that modeling gives.

The primary focus of our approach is to create platform models and MX domains specifically tailored to the needs of the implementation requirements of the class of applications that a project team intends to produce. The approach attempts to solve the overall workflow by starting at the implementation side and working backward to integrate the analysis and modeling tools. We gain the benefit of all the structure provided by the model execution rules, and we are sure that the executable models have a translation. However, integration to front-end analysis capture tools is lacking. The trade-off is to place a human in that role.

It is conceivable that integration with front-end tools could be achieved when using a translation scheme such as pycca. Many tool vendors supply configurable code generators that could potentially be modified to produce pycca source rather than programming language source.

A Role for Humans in Code Generation

We don't consider using a human in a particular role of the software development process to be an unusual circumstance. After all, project teams embark on developing large systems of many tens, if not hundreds, of thousands of lines of code with little more than a text editor and compiler and a few auxiliary tools, perhaps in an integrated development environment tool. We are mystified in the divergent attitudes: it is acceptable to undertake large software projects using conventional ad hoc design techniques with little if any tooling, but somehow the use of models and translation necessitates extensive and complex software tools. We suspect that project teams do not generally have a clear understanding of precisely how the translation is achieved and so have little choice but to assume that the gaps are filled in by software tooling.

We have demonstrated in this book that large, complex tools are not required to achieve the benefits of a model/translate development approach. We are not antagonistic toward tool vendors. They supply a vital role and service. The target platforms supplied by tool vendors are entirely appropriate for many projects. However, many organizations can ill afford the capital costs of complex tools, nor the learning costs associated with tools, nor the administrative costs required to keep complex tools operational. We are also disappointed by the monolithic nature of the available tools and would prefer a more modular solution. We are disconcerted by modeling tools that segment the application class by performance characteristics. For example, some modeling tools purport to be useful for producing "real-time" models. But it's not the models themselves that are real-time. It's not clear how such tools add much support for nonfunctional requirements such as fixed time limits in which a system must respond or deterministic response timing. It is also disheartening that current usage of the term *real-time* has devolved in many contexts to mean "fast enough so that a human is not annoyed." The problems of real-time response must be solved by the implementation strategies of the MX domain, and we see no model-level impact.

Summary

The reference workflow describes the essential tasks and work products necessary, in some form or other, to build and translate xUML models. Any particular approach to translation, including pycca, can be contextualized with respect to this workflow.

The reference workflow describes the steps of building and storing xUML models, populating them, marking them, integrating the modeled and non-modeled domains, translating the models to a platform-specific model, generating code compatible with a model execution runtime component as well as passing through hand code, and then compiling and linking everything into a complete system.

The challenges and tool support relevant to each of these tasks was discussed.

The xUML modeling language can be formally described with a metamodel. This is a model of the language defined itself in xUML. The metamodel also serves as a design for a database schema to store xUML models. At present, most existing metamodels tend to be tool-specific representations. Open source efforts are underway to define a tool-independent metamodel of xUML. (Though, as the pycca approach demonstrates, you can get by without one).

The pycca workflow was presented in the context of the reference workflow. Notable differences are the use of simple draw tools and spreadsheets to capture the models instead of a monolithic tool. Design specifics, including marking, is performed in the process of creating a pycca script. This script is then used to populate a distinct platform model prior to code generation. Although human intervention is required to a greater degree than with a monolithic tool, there are more opportunities to customize the code. Nonetheless, the source models are always carried forward and never altered in the process of translation, as is the case in the reference workflow.

APPENDIX A



xUML Summary

Here is a quick reference to the key features of the Executable UML we use in this book. At the time of this writing, the most complete description is available in the book, *Executable UML: A Foundation for Model-Driven Architecture* by Stephen J. Mellor and Marc J. Balcer (AddisonWesley Professional, 2002). Resources can also be found in the sites mentioned on the Contact Us page in the back of this book.

xUML

xUML is a subset of UML notation with platform-independent execution semantics.

xUML supports object/data-oriented analysis without assuming any sort of object-oriented implementation.

All model elements are executable in a platform-independent context.

This means that you don't need anything extra (for example, code inserted somewhere) to run a model.

If you have a tool that executes the model rules, you can test a model on the computer. Otherwise, you must apply the rules by hand. Either way, the rules of execution are the same. They are never subject to interpretation!

Analysis benefit:

When you review a model, you should be looking at application details, not distracted by modeling rules or fancy notations.

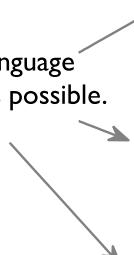
Test benefit:

Easier to memorize the execution rules so you can even run the models by hand.

Implementation benefit:

A lot fewer rules and elements to translate into code. Easier to create a simple and efficient model execution platform.

Have as few model language elements and rules as possible.

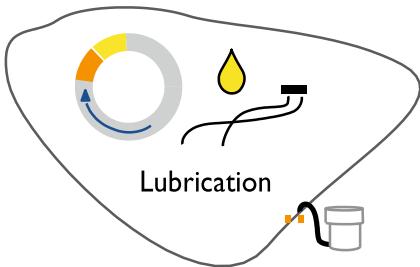


Domain

A domain is a subject matter representing the rules and vocabulary of a real or abstract world.

Examples:

The Lubrication domain knows all about the basic equipment and processes required to perform automatic lubrication on machinery.



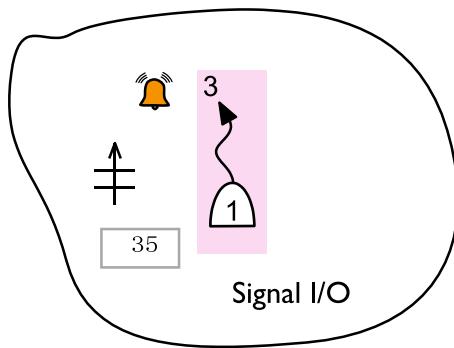
It knows nothing about user interfaces, sensor or actuator technology, alarm management, or logging.

Each domain has cohesive, internally consistent vocabulary defining various rules, policies, and behaviors regarding a particular subject matter.

Domains are defined by the kinds of things they know about, not by any particular function. “Linear Algebra” is a domain. “Matrix multiplication” is not a domain; it is a function.

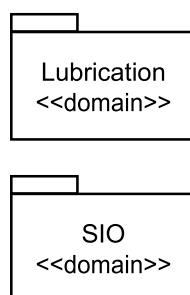
The UML representation is a package stereotyped as a domain. The “domain” stereotype is specific to xUML.

The Signal I/O domain knows all about processing signals to and from the physical world.



It does not know about specific devices or the actual meanings of the signals. But it can be configured to interact with a wide variety of actuators and sensors.

Note that this is not a platform-specific boundary such as a task, thread, library, or process. (It can be implemented many ways and in many distributions.)



If we see these on an xUML domain chart, we assume that all of the packages are domains, so we can omit the <<domain>> text.

Bridge

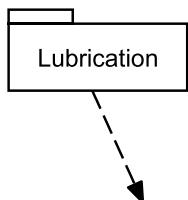
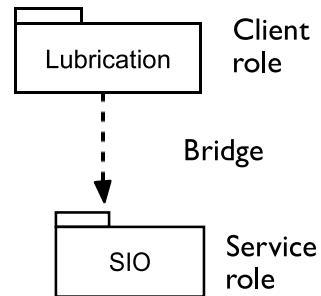
A bridge is a relationship between two domains: one domain requires a capability that it cannot perform itself from another domain.

The domain that requires the capability plays the role of a client. The domain that provides the needed capability plays the role of a service.

A client domain uses a bridge to impose requirements on a service domain.

A service domain gathers requirements from one or more client domains to define its own subject matter.

With the exception of the top-level domain on a project, all domains service at least one client. The top-level domain is often referred to as “The Application.”



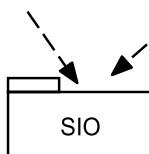
The Lubrication domain needs a way to interact with the physical world. So it requires that capability from another domain.

(Need a way to get current
? pressure and other sensor values. Need
a way to command solenoids.)

The SIO domain provides sensor data and sends data/commands to devices.



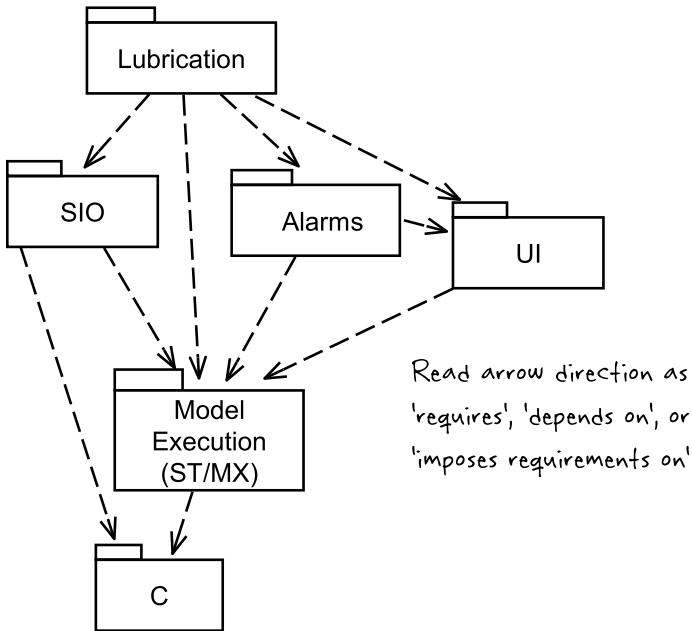
Clients that need access to the physical world via sensors and actuators



Services should be reusable. Because SIO has an Input Point class and no application-specific classes, it can service any client that requires sensor data.

Domain Chart

The set of domains and their connecting bridges necessary to build an entire software system is shown on a domain chart.



A domain may be modeled in xUML, partially modeled in xUML, modeled in another language, hand coded, created in existing legacy software, or acquired from a third party.

Remember, any domain can be split across many processor/thread/task boundaries, or multiple domains can be put in the same implementation unit.

A domain chart is platform independent, so platform- and other technology-specific boundaries are not shown on the domain chart

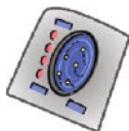
(though you can always draw a layer on top with that information for a particular implementation)

Class

A class is a definition of a set of instances such that all members of the set:

- Have the same characteristics
- Have the same behavior
- Are subject to the same rules and policies

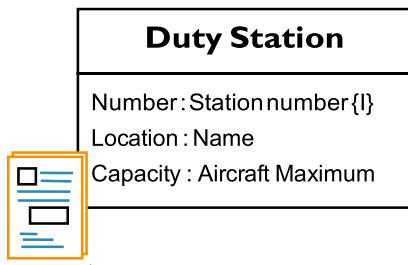
Example:



Each instance of Duty Station has a unique number, a location within the facility, and a maximum number of aircraft it can manage.

Representation:

The set definition can be drawn with a UML class symbol.



The reasoning behind the abstraction decision is captured in a class description.

Execution:

One of the easiest ways to visualize the population of values for the attribute x instance sets is with a table.

Duty Stations		
Number	Location	Capacity
S2	Center	30
S1	Front	20
S3	Front	45

The symbol and table are not interpreted as a programming language class or a database style table, though those are among many possible implementations.

It is a mathematical construction that crosses a set of attribute, data type pairs with a set of instances. Consequently, row and column order have no significance or influence on an implementation.

Attribute

An attribute is a characteristic of something that has been abstracted as a class. It follows from the class definition that each instance must have a meaningful value for each of its attributes at all times.

At all times: The meaning and use is explained in an associated attribute description.

The values that may be assigned to an attribute are constrained by a data type.

Example:

Duty Station
Number : Station number {}
Location : Name
Capacity : Aircraft Maximum

Data type:

Aircraft maximum

A standard number of maximum aircraft that can be managed. Legal values are 20, 30, 45, and 60.

Representation:

This is not the same as just saying *integer* as only certain values are allowed. But it is a numeric quantity.

The UML standard notation is used for showing an attribute on a class symbol.

For xUML, the tags I, I<num>, R<num> may also be placed after the data type between UML tag {} brackets.

See Identifier and Association headings for explanation of these tags.

Unconstrained data types such as integer, string, and real are rarely used in xUML models because they don't reflect real-world constraints. An attribute such as *altitude* may be subject to real number operations, but it is probably constrained by a max and min value.

Supported operations such as those for comparison, manipulation, unit conversion, and component access must be defined for every type.

Data Type

A data type is a constraint on the real-world values that may be assigned to an attribute and the operations that may be applied to those values.

Examples:

Pressure, mpa

Count

Count can be defined as an integer [0..maxint]

Even though it is based on integer, the only supported operations are increment, decrement, and reset.

Aircraft tail number

This is not defined as string, because many strings would be illegal. A regular expression could be defined to describe legal tail number names.

Also, most string operations would not be allowed on such a value.

Point

A 2D point type could be defined with operations that produce either a Cartesian or polar point, thus encapsulating the internal representation.

```
r, theta = Track.Origin.Polar
x, y = Track.Origin.Cartesian
```

Definition of types (type theory) is orthogonal to relational theory used for defining class and association structures and behavior.

Identifier

An identifier is a constraint on a set of one or more attributes on the same class such that no two instances of a class share the same value.

Real-world identifier

An identifier can be a real-world identifier such as Tail Number. Real-world agencies establish this constraint.

Formalizes a real-world constraint.

Invented identifier

Or it can be an arbitrary value invented for the purpose of uniqueness. In this case, the modeler establishes the constraint.

It is useful in conjunction with referential attributes to formalize relationships and other constraints.

Symbol

Identifiers are numbered starting at 1 for each class. Each attribute component of an identifier is tagged with {I} or {In}, where {I} means ID 1 and {In} indicates the participating ID number. An identifier attribute may participate in one or more of its class's identifiers.

Examples:

Airport
Name : ICAO Name {I}
Center location : GPS {I2}

Each Airport is assigned a unique ICAO assigned name.

No two airports may share the same geographic center.

Runway
ID : ID Number {I}
Airport{I2,R1}
Side : LeftRight {I2}
Name : Approx Heading {I2}

Each Runway has a unique invented ID (1, 2, 3, and so forth).

You cannot have two 27R runways at the same Airport.

The greater UML community encourages the implicit assumption of an architecture-supplied handle to manage links among instances. In xUML, we rely on at least one explicit identifier on each class. This ensures that we have a consistent system of relationship references, and it gives us a powerful tool for expressing restrictive constraints on relationships.

Association

An association is a relationship that holds systematically among instances.

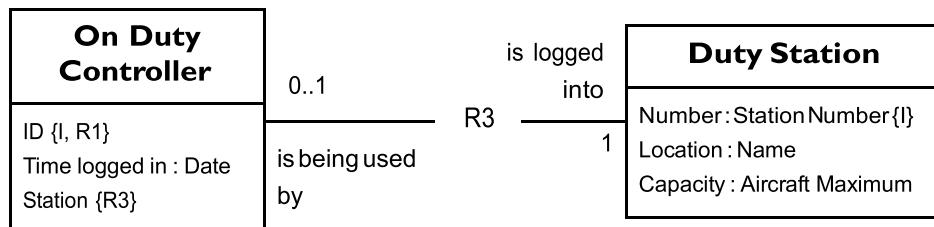
Example:

On Duty Air Traffic Controller is logged in to a Duty Station.

A Duty Station is being used by zero or one On Duty Air Traffic Controller.

The rule is systematically applied to all instances of On Duty Controller and Duty Station, without exception.

For example, an Air Traffic Controller may not go on duty if there is no Duty Station available.



Referential attribute

In xUML, associations are not merely drawn. They are formalized with one or more referential attributes.

Any value assigned to On Duty Controller.Station must match the value of Number for an instance of Duty Station.

On Duty Controllers

ID	Time logged in	Station
ATC53	Center	S2
ATC67	Front	S1

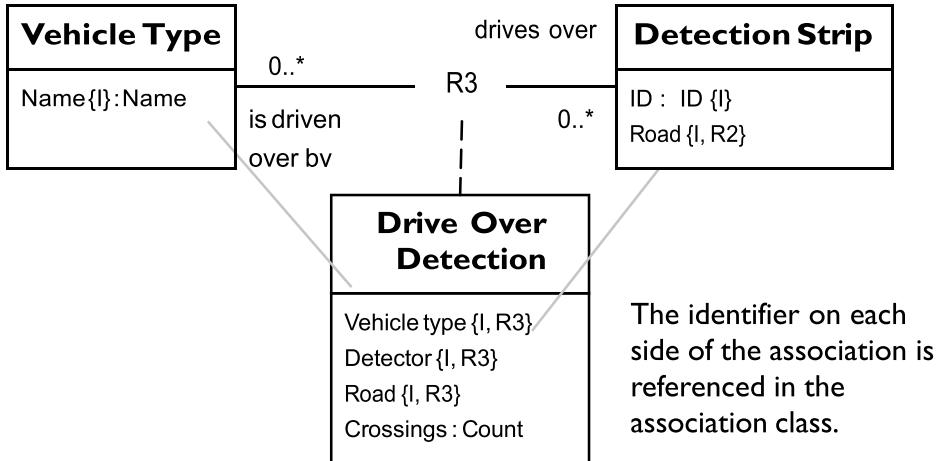
Duty Stations

Number	Location	Capacity
S2	Center	30
S1	Front	20
S3	Front	45

ATC53 is “linked” to S2 by virtue of its current Station value. Yet there is no actual link in the object-oriented programming sense.

Association Class

An association class formalizes an association between two other classes.



For a many-to-many association, both references must be combined to form an association class identifier.

Drive Over Detections

Vehicle	Detector	Road	Crossings
Heavy Truck	D802	HWY1	214
Motorcycle	D72	I280	50
Light Truck	D72	I280	102

The table represents a functional mapping between Vehicles and Detectors.

An instance of Vehicle Detection cannot exist without a “drive over” interaction between one Vehicle Type and one Detection Strip.

Always use an association class for many-to-many and any conditional-to-conditional or many-to-0..1 association to ensure that a referential attribute always has a non-null value.

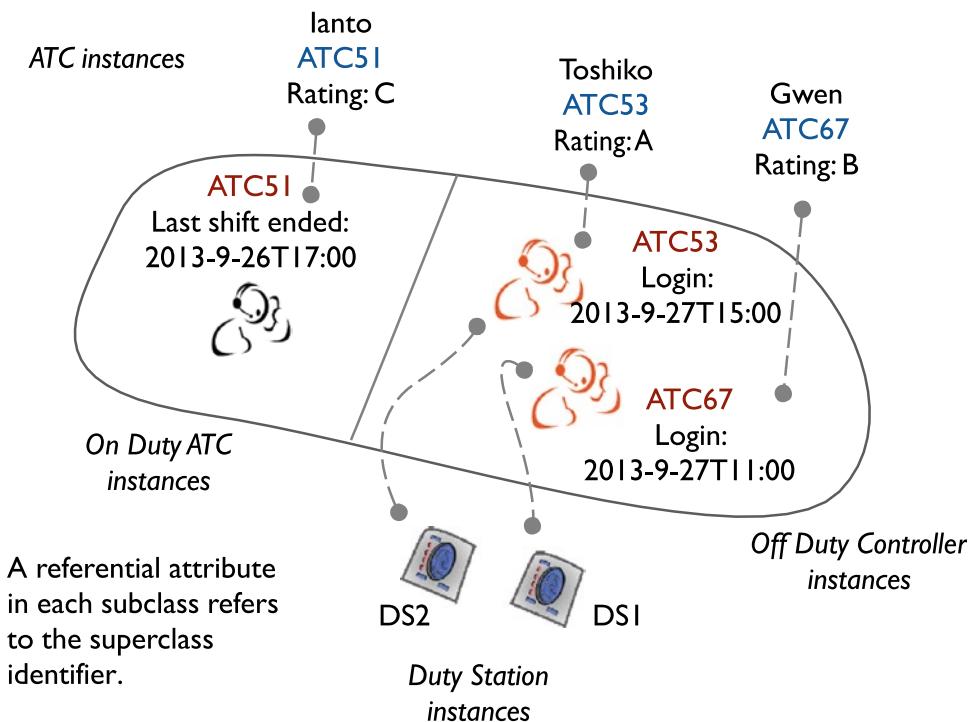
Generalization/Specialization

A set of instances may share properties and rules and at the same time have exceptions. To handle this situation, the set definition established by a class is partitioned into mutually exclusive subsets.

At all times:

An ATC cannot exist without being On or Off Duty.

An On or Off Duty Controller is certainly an ATC.



Each real-world object has simultaneous membership in the superclass and subclass sets. Consequently, one object = two connected instances in a generalization.

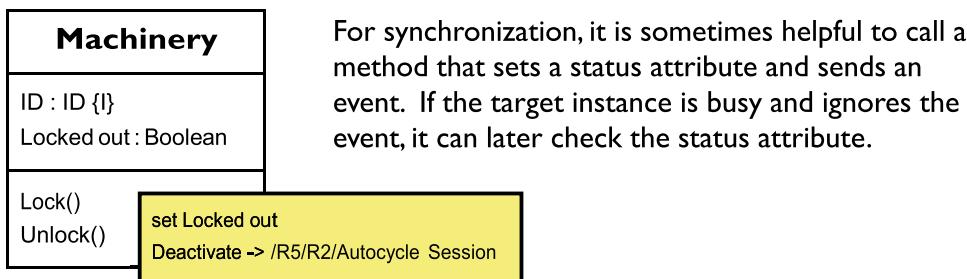
Generalization, in this case, is employed to establish an exclusive OR constraint.

Other Activity Types

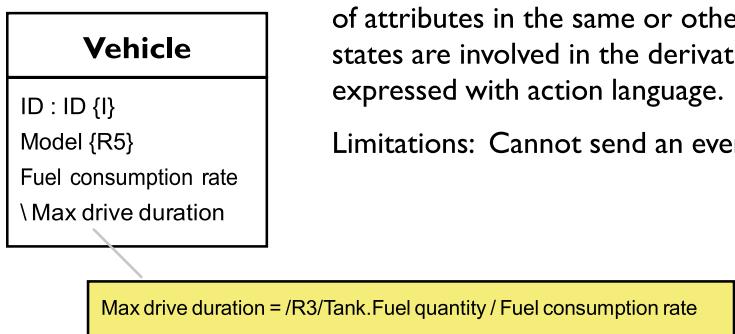
In xUML, activities are primarily state activities. But they are also written in class methods, external entity operations, and domain operations.

Class Method

There are several reasons to put an activity in a class method rather than in a state. For example, an activity might be triggered from multiple states. Or a class may not have a state model, but still be subject to state-less processing.



Derived Attribute Definition



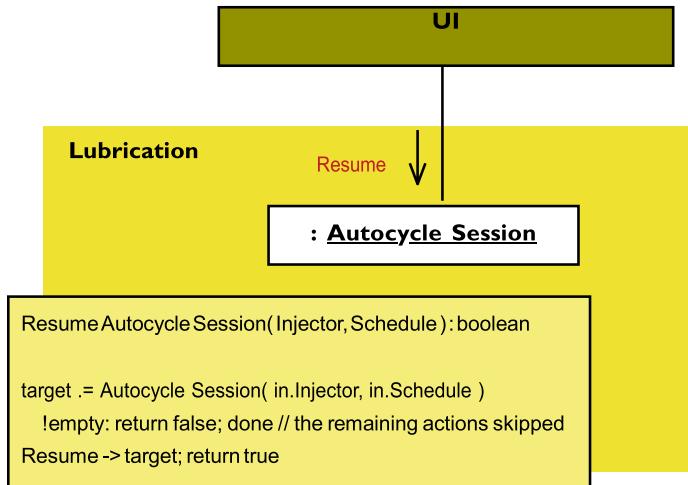
The model merely states that when a derived attribute is accessed, it provides a correct value according to a platform-independent formula. When and how frequently the value is updated, on the other hand, is platform specific and, hence, not in the model.

Domain Operation

A domain operation provides a runtime, model-level API to external domains. The activities should do nothing more than validate incoming data, locate relevant model elements, and then issue an appropriate event or invoke a class method.

Limitations: Cannot refer to self attributes because there are none.

Domain operation in Lubrication domain invoked via bridge from UI domain



State Model (Instance Lifecycle)

By definition, all instances of a class share the same behavior. This behavior follows a common life cycle that can be formalized with a state model. (There are many other uses of state models in software, but in xUML, we use them exclusively to capture instance life cycles).

State Machine Behavior

An instance remains in a state of its state machine for events to occur.

When one or more events occur, one of them will be consumed according to the event synchronization rules.

A consumed event results in one of three responses:

Follow a transition (as shown on the state model diagram)

The instance follows the transition and immediately begins executing the activity in the destination state (which may be the same state).

Ignore the event

The event is consumed and exists no longer.

Can't happen

The event is not anticipated and constitutes an unrecoverable error in the model. The model execution domain is responsible for handling the error outside the model.

Platform Independent Synchronization Rules

Events

At some time after an event is generated, it is made available to the destination instance or external entity.

Events are never lost: every event will be delivered to an instance or external entity to which it is directed.

After an event is consumed by an instance, it no longer exists.

Multiple events can be pending for a given instance.

(For example, events may arrive for an instance while it is executing an activity. Or, multiple events may simply arrive concurrently.)

If events are pending for an instance that were generated by different senders, it is indeterminate which event will be consumed first.

An event to self is always consumed by the instance that sent it before that instance consumes any other event.

If an instance generates multiple events to a target instance, the events will be received in the order generated.

Activities

An activity takes time, possibly none, to execute. (The actual duration depends on the platform.)

Once initiated, an instance's state activity must complete before another signal can be accepted by the same instance.

An activity always runs to completion in xUML. So, if you need the ability to interrupt behavior at the model level, break it down into multiple activities spread across one or more state models.

Only one state activity of a given instance can be in execution at any time, because an instance is in only one state at a time.

Multiple actions in an activity can execute concurrently. Activities in different instances can be executing concurrently.

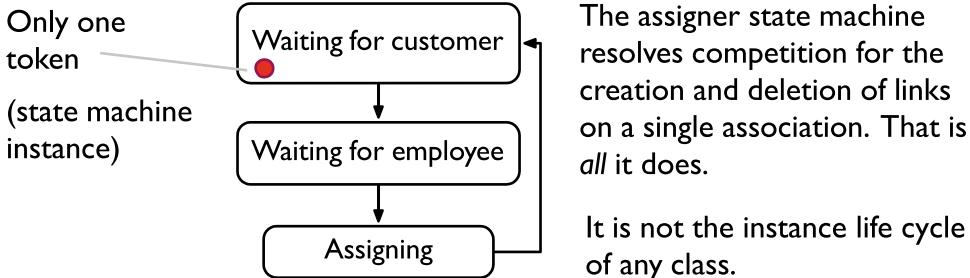
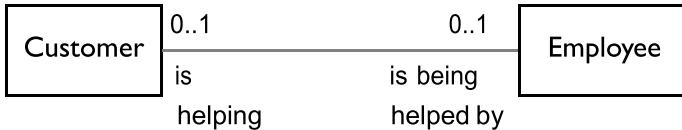
When an activity completes, it must leave the system consistent.

State Model (Assigner)

An assigner is a special state model that manages a competitive association. It follows all of the synchronization rules, but has no instances. Instead, a token moves through the states.

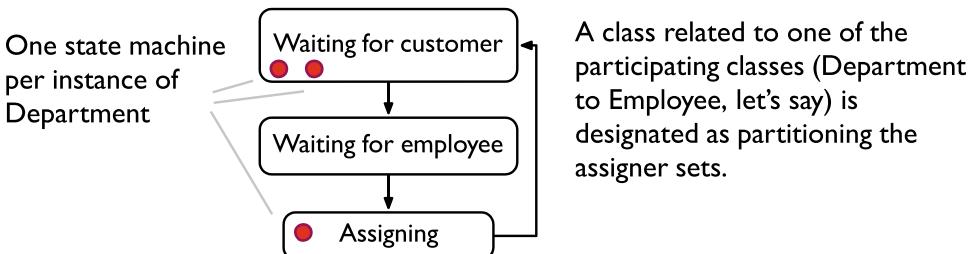
We need them when competition among instances makes it impossible for instances on either side of the association to avoid linking to the same instance.

Single Assigner



Multiple Assigner

Now let's expand the scope to consider multiple departments in a store. In this case, there is localized competition among customers and employees within each department. This requires a multiple assigner.



The important concepts are 1) a single point of control is necessary to resolve competition in an asynchronous world and 2) the xUML synchronization rules make it possible to specify a platform-independent solution. The solution is not left as an “implementation detail.”

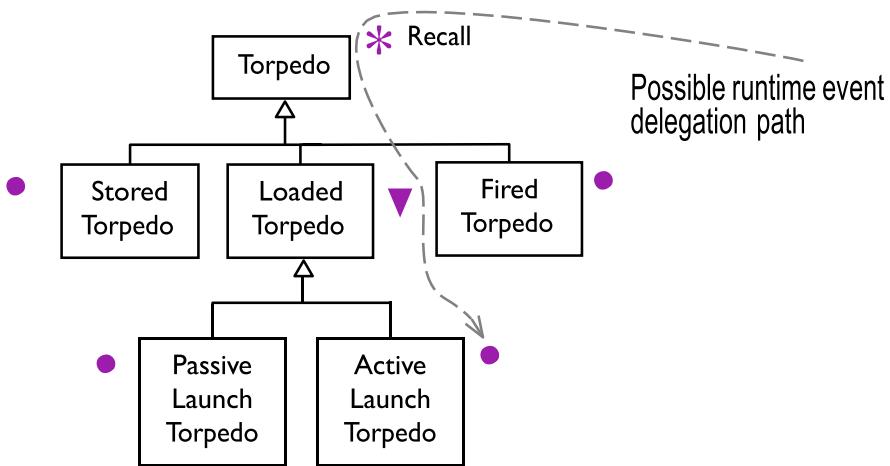
Polymorphic Events

A polymorphic event is declared on a superclass that does not define a response to the event and need not have a state model at all.

The event is delegated through each specialization branch originating from that superclass.

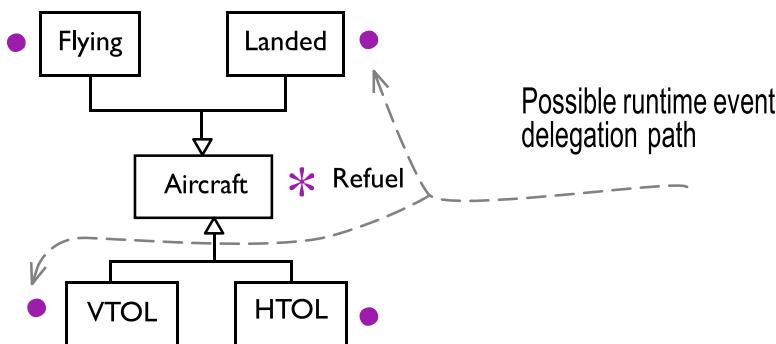
Each encountered subclass in each specialization hierarchy must either further delegate the event or respond to that event in its own state model.

In a single, or repeated specialization (same direction), exactly one state machine must respond to any runtime instance of the event.



- * Polymorphic event declared or delegated
 - ▼ Polymorphic event delegated
 - State model responds to polymorphic

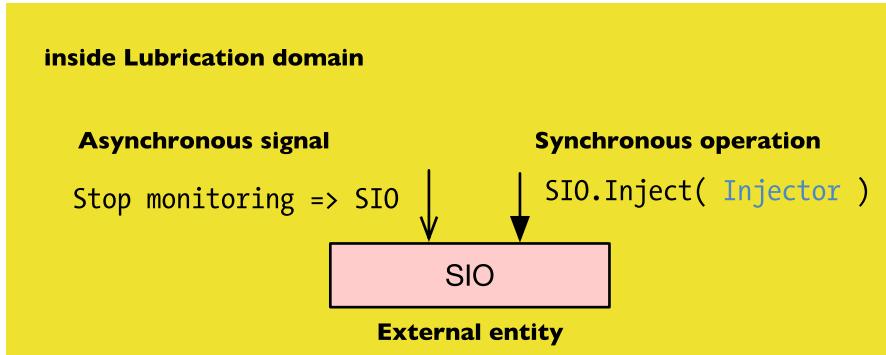
If there are multiple specializations (different directions), the event is received by one state machine instance in each.



External Entity

An external entity serves as a proxy for runtime communication with external domains. It can be named to match the domain or it can represent a large-scale or abstract entity handled by the domain. (Care should be taken to avoid naming a specific model or code element in the other domain subject to change.)

An external entity operation is triggered by a synchronous or asynchronous call from inside a domain. The activity typically invokes a domain operation provided by another domain.



An asynchronous signal to an external entity initiates behavior in another domain that may be ongoing.

Regardless, control is immediately returned to the sending domain with no value returned, much like sending an event.

After the Stop Monitoring signal is sent to the SIO external entity, ongoing periodic behavior is initiated in the SIO domain, but control returns to the Lubrication domain immediately.

A synchronous operation initiates behavior in another domain and waits until completion and the possible return of a value, much like a method call on a class.

When the Inject synchronous operation is invoked on the SIO external entity, control is not returned until the corresponding solenoid is triggered. A value could be returned, but none is specified in this particular operation.

APPENDIX B



Scalll Overview

Scalll (Starr's concise relational action language) is the psuedo-action language that we use in this book to specify model actions. You can find the most complete and current description of the language at www.executableuml.com. For now, we present those aspects of the language used in this book to provide more context than the running explanations in the text.

Principles

As with the class and state model facets of xUML, the action language must be executable and platform independent. Scalll is designed to be consistent with xUML execution semantics. At the same time, it strives to be easy to read and write.

Names

Scalll is designed to be readable without being needlessly verbose, delegating the hard work to the parser and advanced editing tools. The names of model elements, variables, and values may contain spaces, though care must be taken not to incorporate keywords delimited by spaces into names. To maximize the comfortable use of whitespace, effort is made to define as few keywords as possible. A variable name such as the `shift spec` is, therefore, perfectly legal. This approach differs from the design of a language that attempts to be readable by introducing wordy keyword phrases.

Variable Types

There are three kinds of variables: instance set, scalar, and relation. Only the first two are used in this book, so we only lightly describe the relation variable in this appendix.

Instance Set Variable

An *instance set variable* is typed as a class in the local domain and refers to zero, one, or many instances of that class.

Relation Variable

A *relation* is a set of attribute/data type pairs crossed with a set of tuples, each of which supplies a value for every attribute. There may be zero, one, or many tuples assigned, and zero, one, or many attribute, data type pairs. A relation variable holds a relation that can be visualized as a table (even though a relation really isn't a table). A tuple is a simple case of a relation, so there is no need for an additional tuple variable type. Here is an example of an assignment to such a variable:

```
acdata #= /R8/Aircraft.(ID, Altitude, Airspeed)
```

Here the acdata relation variable holds a relation with the attribute:data type pairs ID: Tail Number, Altitude:MSL Altitude, and Airspeed:K, and a tuple supplying a set of corresponding values for each instance of Aircraft. Again, it is easy to visualize this as a table with attributes in the heading and each row representing an instance of Aircraft. The #= assignment operator implicitly types acdata as a relation variable.

Scalar Variable

The term *scalar* in our context means “not a relation.” A scalar variable holds a single value belonging to a set defined by a data type. The definition of data types is orthogonal to the relational organization of instance data in classes. Examples of system-supplied scalar data types are integer, real, rational, and string. By convention, system types are lowercase names. An example user type might be GPS Coordinate. By convention, user types are all initials uppercase—for example, Compass Heading.

Data Types

A *data type* defines a set of scalar values *and* the operations that can be performed on members of that set. System-supplied data types are typically boolean, integer, real, and string. User types such as GPS Coordinate, Pressure, or License Number can be constructed based on restrictions, extensions, and structural combinations of these types.

Operations can be performed by using infix, prefix, and suffix operators and with names accessed using . notation. For example, the Count data type may be defined as an integer in the range [0..maxint] with the supported operations being incr, decr, and reset.

`++cars` passed or `cars .incr` might both invoke the increment operation defined for the Count type. `cars .passed.reset` could set the value to zero.

`current latitude = position.latitude` invokes the latitude accessor on the GPS position type.

System Variables

System-supplied variables are prefaced with an underscore. `_now`, for example, provides the current time to models in any xUML domain. So that the models can be run on as many MX domains as possible, there are few system variables.

```
Time logged in = _now.HMS
```

Here the `_now` system variable invokes the system-defined HMS operator, which returns time in hours, minutes, and seconds. The assignment implicitly types the scalar variable on the left side.

No Literals

Literal values may not be specified in actions. The idea is to eliminate the encoding of data into action language rather than where it usually belongs, as instance data in the class models. Mechanisms are provided for allowing access to special values such as pi, zero, and so forth. In all cases, though, these are named values defined outside the action language itself. For example, default initial values can be specified with the standard UML notation on the class diagram. Double underscores reference a named value defined somewhere outside the state models `_pi`, `_0`. Note that with a variable typed as Count described earlier, you can set it to zero by invoking the reset operation.

Boolean Values

The `set` and `unset` operations are used instead of explicit assignment to literal `true` and `false` values.

Examples:

```
Injecting.set ⇒ set to true
Injecting.unset ⇒ set to false
if not Injecting ... ⇒ to test state
```

Enumerated Values

Although literal values are forbidden, you can define enumerated data type values. A `Fluid State` data type might enumerate the values `low`, `very low`, `empty`, and `normal`. To refer to an enumerated value, put a `.` in front of it.

```
Level = .very low
```

If the left side is an attribute, it is already typed, presumably as `Fluid State` in this example. With a scalar variable, it may be necessary to explicitly type it with the `::` type declaration operator.

```
level var::Fluid State = .full
```

Spaces are allowed in enumerated values.

Attribute References

An instance refers to its own attribute values without qualification. So an ATC instance can refer to its `Last shift ended` attribute without the need for any kind of self-qualification. Input parameters are distinguished with the `in` preface to avoid name collisions. And a variable name may not match any attribute name of the local class.

Assignment Operators

Variables are implicitly typed when first introduced on the left side of an assignment operator. It is also possible to explicitly type them by using the `::` operator, though this is rarely necessary.

Operator	Meaning
.=	Assigns zero or one instance. If multiple instances are found, an error results. Normally used in conjunction with a set of identifier values to select an instance.
..=	Assigns zero, one, or multiple instances to an instance set variable.
.=.	Assigns an arbitrary single instance from a set of instances, or no instance if none is found.
=	Assigns a scalar value.
#=	Assigns a relation.

Instance Selection

To select an instance, specify a class name followed by optional selection criteria in parentheses.

Selection with No Criteria

No critiera is required to select a singleton instance of a class.

```
the shift spec .= Shift Specification
```

The preceding statement selects one instance of the Shift Specification class. Because no criteria is supplied, and because there is only one instance in a singleton class, that instance is selected. If none is found, an error occurs.

Or you can select all instances of a class:

```
all aircraft ..=Aircraft
```

Because no criteria is specified, the all aircraft instance set variable will contain references to all instances of Aircraft. If there are presently no instances of Aircraft, zero instances will be referenced by the variable.

Selection with Criteria

The usual comparison operators (>, >=, and so forth) are supported. The : symbol stands in for == because in the most common case, you want to match a value, such as selecting the instance with a matching ID. Conceptually, you are looking for a match, so a single symbol is used. Read it as *matches*.

For example:

```
my station .= Duty Station( Number: in.Station )
```

This selects a single instance of Duty Station whose Duty Station.Number matches an input parameter value.

Boolean operators such as and, not, and or may be mixed in. For convenience, the , separator is interpreted as an and operator. The combination of the : and . operators make it easy to select a single instance based on a set of matching values without the clutter of a series of and and == symbols.

```
landing runway .= Runway( Airport: in.airport code, Runway ID: in.runway )
```

This selects an instance of Runway that matches two values. Presumably, the two values taken together are supposed to be unique for the Runway class. If more than one instance is found, an error results.

```
high fast aircraft ...= Aircraft( Altitude > high alt and Airspeed > high speed )
```

This selects all aircraft above a certain altitude and faster than a certain speed.

Relationship Navigation

The / symbol delimits hops across class model relationships. The leftmost / indicates the beginning of a relationship path expression beginning at the local instance.

```
hoff zone .= /R2/Control Zone( Name: in.Zone )
```

The path expression on the right-hand side starts at the local instance and hops across R2 to all related instances in the Control Zone class. The selection criteria narrows the set to the single instances whose Name attribute matches the in.Zone value. Assuming that the Name attribute is the identifier of Control Zone, either zero or one instances should be assigned to the left side.

Relationship names (R1, R2, and so forth), verb phrases, and class names may appear between the hop delimiters to define an unambiguous selection path.

Signaling

To send a signal, use the following expression:

```
<event name>(params) [after <duration>] -> instance set
```

For example:

```
Low injection pressure -> /R3/Reservoir
```

Here an event is sent to the linked instance. If multiple instances are in the target instance set, a *separate* event will be delivered to each.

An instance can send an event to itself by using the me keyword. This keyword represents the local state machine.

```
Injection canceled -> me
```

An assigner state model does not refer to any local instance, but because the me keyword refers to the local state machine, it can be used there also for self-directed events.

A delay can be specified as shown:

```
Good injection -> me after /R4/Injector Design.Good injection duration
```

Here the Good injection event is sent after the duration specified in the related Injector Design. The relationship along R4 is presumably to one unconditional. An error occurs if the path expression does not select exactly one instance.

Link/Unlink

The association operators are `link &`, `unlink !&`, and `swap`.

The form is as follows:

```
[<instance set>] & <association>/<instance set>
```

Each instance in the optional leftmost instance set is linked to each instance referenced by the set on the right across the path. If the instance set on the left is not specified, it is assumed to be the local instance.

```
& /R3/my station
```

This links the local instance to each instance referenced by the variable across R3.

If the association is formalized by an association class, an instance of the association class will be created for each link. The required referential attributes will be set to values referring to each participating class instance as appropriate. Any other attributes will be set to their default initial values.

Another way to link is to set the values of a set of each referential attribute on the association to match the values of corresponding identifier attributes on the other side of an association. In general, it is safer to use the link action.

The unlink operator is `!&` and uses the same form as the link action:

```
!& /R5/takeoff runway
```

This unlinks the local instance from all instances referred to by the `takeoff runway` variable across R5.

Another way to unlink is to simply delete the instance holding the referential attributes referring to the other side of an association.

With an unconditional one association, it is important not to leave the association unconnected. To avoid this situation, use the `swap` operator instead of a pair of link and unlink actions:

```
swap /<association>/<old instance> with <new instance> [on <association>]
  !new missing: <action block>
  !old missing: <action block>

swap /R2/hoff zone with new controller [handed off]
  !new missing: UI.Unknown controller( Controller: in.Controller)
  !old missing: UI.Zone not handled by( Controller: ID)
  !handed off {
    // further actions assuming handoff succeeded
}
```

The error conditions underneath the swap operation are explained next.

Error Handling

Some actions may trigger unrecoverable errors. Each such error sets a named guard. The names are predefined for the action. If the error is, in fact, recoverable, the modeler can specify an internal response and set an optional user-defined guard, with the `[]` guard set brackets to regulate downstream processing.

In the swap action example, two guards, `new missing` and `old missing`, are predefined for the action. A user-defined guard is set to the right of the swap action, which will be disabled if any error occurs. A single action is specified for each of the two possible errors, making them both recoverable. If neither error occurs, the handed off guard is not disabled, and an action block may be specified associated with that guard.

```
my station .= Duty Station( Number: in.Station ) [station OK]
  !empty: UI.No such station( Number: in.Station )
!station OK {
  // actions that assume a station has been selected
}
```

Here the `.=` assignment action has two errors predefined: `empty` and `many`. If both are ignored, they are unrecoverable. By inserting the `!empty` clause, the modeler indicates that the empty case is okay and there is a response. Note that in the earlier singleton selection example, the error should be unrecoverable. The `station OK` guard is disabled if `empty` occurs and can be used to enable further processing.

If there are several actions to perform, they can be enclosed in brackets to form an action block. Otherwise, a `:` follows the error name.

Subclass Migration

The `migrate` operation deletes the local instance in one subclass, creates it in another, and relates the new subclass instance to its parent superclass:

```
migrate [<instance set>] to <subclass name>
```

All instances in the set are migrated to the destination subclass. If the instance set is omitted, the local instance only is migrated:

```
migrate to On Duty Controller
```

If the local instance is an `Off Duty Controller`, it will be migrated to the `On Duty Controller` subclass.

This is a convenience action that replaces the corresponding create and delete actions that would be otherwise necessary and ensures that a correct migration occurs without breaking the integrity of the model by, say, leaving an illegal orphaned subclass instance behind.

Interaction with External Domains

Other than special access provided to system-supplied underscore variables, there are two ways to interact with other domains at runtime.

Asynchronous: Signal to External Entity

Here a signal is sent to an external entity, which is then responsible for mediating that signal out to another domain in some form. Unlike a regular event, an asynchronous signal is not directed to any particular instance in the external domain. It is simply addressed to the external entity.

```
Start monitoring => SIO
<signal name><parameter list> => <ext entity name>
```

This is used when you want to signal an activity in an external domain with the expectation of an asynchronous response at an undetermined point in the future.

Here the warning signal is sent to the UI external entity acting as a proxy, presumably for the UI domain.

Synchronous: Invoke Operation on External Entity

Here there is an immediate and complete interaction with an external domain. A value may or may not be returned, but there is no further activity triggered externally.

```
UI.Warning( MSG Control Zones Active, ID )
<EENAME>.<operation name><parameter list>
```

Self Reference

When sending a signal to self, the `me` keyword refers to the current state machine instance. In an instance life-cycle state activity, this corresponds to the local instance's state machine. In an assigner state activity, there is no class instance, so `me` refers to the assigner state machine instance.

Events to Assigner State Machines

An assigner state machine is attached to a competitive association—that is, an association with constrained multiplicity (a 1 on one or both sides), where instances on both sides are concurrently attempting to link to one another.

In such a case, the association is bound to one or more state machines that resolve the competition. This is called a *single assigner association*. If all instances on one side can link to all instances on the other side, a single assigner suffices. But if an instance on one side of the association may link to only a subset of instances on the other side, a separate assigner state machine is required for each subset. This is called a *multiple assigner association*.

The format for sending an event to an assigner state machine depends on whether the target competitive association is a single or multiple assigner.

For a single assigner, do this:

```
<event name> -> <path to association>/assigner
Customer waiting -> /R2/assigner
```

For a multiple assigner:

```
<event name> -> <assigner association>(<partitioning instance>)
Group ready -> /R6/assigner(/R5/Signal Converter)
```

The partitioning instance is the one that breaks, via some association, one class participating in the competitive association into subsets.

In both cases, the event is addressed to a single assigner state machine and not to any instance life-cycle state machine.

Class Method

A class method is invoked by using dot notation on an instance set variable. If the set contains multiple instances, each instance's local method is invoked.

```
allAircraft ..= Aircraft // All aircraft selected  
allAircraft.verifyPosition() // is applied concurrently to each instance  
Aircraft.verifyPosition() // does the same thing without the instance set variable
```

Parameters input to an activity are available, as `in.<parameter name>`.

When specifying a parameter list for a class method invocation or any other action that takes a parameter list, the format is as follows:

```
( <parameter name>: value, ... )
```

There are some shortcuts to reduce verbosity. If the parameter name and the name of the variable supplying the value are the same, the parameter name can be omitted.

More Commands

Only the subset of Scrall necessary to follow the examples in this book has been covered in this appendix. A more complete (and evolving) specification is available online at www.executableuml.org.

APPENDIX C



Pycca Language Overview

This appendix summarizes the pycca domain-specific language and the most important C preprocessor macros used for translating state activities. Complete documentation can be found at the book's website, www.modelstocode.com.

Invocation

```
pycca ?OPTIONS? FILE ?FILE FILE ...?
```

The pycca program translates a specification of a domain into C code that is suitable for use with the ST/MX Model Execution domain. The language that pycca translates is a simple configuration language that allows the specification of domains, classes, state machines, and other model elements. Any associated processing is specified in ordinary C code.

Pycca generates two files from its input. One file is the generated C code, named by appending a `.c` suffix to the basename of the first input file. The other file is a generated header file that has an `.h` suffix. More than one input file may be given in the invocation. Subsequent files are processed as if all the files had been concatenated together. Typically, second and subsequent files hold domain population information so that a domain may be populated differently without modifying the file containing the translation of structural aspects.

Options

-help or -?

Print the help message.

-version

Print the version number and license for pycca, and then exit.

-noline

Do not output #line directives in the generated file that reference the pycca source file. Normally, the generated C code contains line directives too, so that compiler error messages reference the pycca source rather than the generated code. However, some debuggers are confused by these directives.

-output *filename*

Output the results to *filename*. This option allows finer control of the output file names. By default, pycca derives the output file names from the source name by substituting any suffix in first *FILE* argument with .c and .h. If *filename* is a directory, the output file names will be constructed as usual, except with *filename* as the leading part of the path. Otherwise, *filename* is used as the base for constructing the output file names by substituting .c and .h for any suffix that *filename* may have.

-instrument *pattern*

Insert instrumentation code for all classes that match *pattern*. To instrument everything, specify *pattern* as an asterisk (*). *Pattern* is a list of simple wildcard-type string-matching expressions. Any functions for a class whose name matches any of the given patterns will have instrumentation code inserted at the beginning of the function that is intended to help trace code execution. Note that *pattern* is interpreted as a list of whitespace-separated elements and that any wildcard specifications in the *pattern* will usually require quoting to prevent interpretation by the invoking command shell.

-noC99

By default, pycca produces code intended for a C compiler that meets the C99 standard. Supplying this option removes any C99 constructs in the generated code. This is for the benefit of old compilers. Note, however, that pycca does not examine any of the passed-through code, and it may well contain constructs that a particular “C” compiler does not accept.

-dataportal

Insert into the generated code file a set of data structures and encodings that allows certain model-level operations to be invoked from outside the domain. This data is used with the portal code to support bridging into the domain.

-portalcode

Create the files pycca_portal.h and pycca_portal.c in the current directory. Pycca exits successfully after creating the files. These two files are the C code used in conjunction with the data structures generated with the -dataportal option.

Lexical Conventions

Pycca uses several lexical conventions to distinguish language constructs for C code constructs.

Comments

Start with a sharp character (#) and continue to the end of the line.

Whitespace

Is generally ignored, and no constructs depend on any particular arrangement of whitespace.

C Variables

Any text appearing between matching parentheses, (...), is taken to be a list of comma-separated C variable or parameter declarations. The C declarations of parameter lists and attribute declarations are parsed. Because of the inherent ambiguity of `typedef` type aliases and the complexity of certain C declarations that involve constant expressions, it is possible for `pycca` to incorrectly parse the declaration. These issues can usually be solved with an appropriate `typedef` included in the implementation prolog section.

C Code

Any text appearing between matching braces, {...} is taken to be a C code sequence and is otherwise uninterpreted. The determination of the matching enclosing brace accounts for the syntax of the C language. Note that comments in C code must follow C language comment conventions, because any code is passed through to the C compiler.

Name

A sequence of a letter followed by an arbitrary number of letters, decimal digits, or underscore characters that is not also a keyword. This is the same convention used for C identifiers.

Number

A sequence of decimal digits.

Keywords

The tokens in Table A-1 are keywords and may not be used where an arbitrary name is required.

Table A-1. Pycca keywords

attribute	class	constant	constructor
default	destructor	domain	dynamic
end	epilog	event	external
final	implementation	initial	instance
interface	machine	operation	polymorphic
population	prolog	reference	set
slots	state	static	subtype

Other Tokens

The strings in Table A-2 are also lexical tokens.

Table A-2. Other Pycca Tokens

->>
->>c
->>n
->>l
-ddd>>, where <i>ddd</i> is a sequence of decimal digits
->

Domain Definition

The pycca language allows the definition of *domains*. Domains consist of a set of components such as classes and operations. In this section, we show the syntax for defining domains.

domain

Domain definitions start with the `domain` keyword followed by the name of the domain and stops at the matching `end` keyword.

```
domain lube
    # Put your domain definition here
end
```

The remaining statements in this section may appear inside the domain definition. For brevity, we assume that the statements are enclosed in the `domain` statement.

class

A `class` is a template for data and behavior. A class name must be a valid C identifier. A class definition starts with the `class` keyword and stops at the matching `end` keyword. See the class definition section for statements to define the properties of classes.

```
class Air_Traffic_Controller
    # statements defining the Air Traffic Controller class
end
```

domain operation

The procedural interface to a domain consists of a set of *domain operations*. Domain operations are converted into ordinary C functions. They are made external in scope, and their prototype is inserted into the generated header file. If a domain operation returns a value, the type is specified by following the interface with a colon (:) and a variable type. If no return type is specified, the function is typed as `void`. To help manage the global namespace, the name of the domain and an underscore are prepended to the C function that is generated for a domain operation.

```
domain operation
Injector_max_pressure(
    InstId_t injId)
{
    PYCCA_checkId(Injector, injId) ;
    ClassRefVar(Injector, inj) = PYCCA_refOfId(Injector, injId) ;
    if (IsInstInUse(inj)) {
        InstOp(Injector, Max_system_pressure)(inj) ;
    }
}
```

external operation

Complementary to domain operations are external operations. They are defined similarly to domain operations. External operations define the external function dependencies of the domain.

```
external operation
SIO_Inject(InstId_t injectorId)
{
    // any code here is not passed through
}
```

The domain expects these functions to be supplied from elsewhere. The preceding example places the following external declaration into the generated header file (assuming the operation is defined in the lube domain).

```
extern int eop_lube_SIO_Inject(InstId_t injectorId) ;
```

interface prolog, implementation prolog, interface epilog, implementation epilog

Additional code can be placed in the generated files either at the beginning (prolog) or at the end (epilog). The **interface prolog** statement places code near the beginning of the generated header file, and **implementation prolog** places code near the beginning of the generated code file. Replacing **prolog** with **epilog** causes the code to be included at the end of the respective generated files. There may be multiple **prolog** or **epilog** statements, and the contents are accumulated across all such statements to be placed in the generated files at the appropriate place.

```
interface prolog {
    #include "pycca_portal.h"
    #include <stdint.h>
    // Any additional interface includes, etc.
    typedef uint32_t Count ;
}

implementation prolog {
    struct Point {
        int x ;
        int y ;
    } ;
    static int multPoint(int a, Point *p) ;
}
```

```

interface epilog {
    #define POINT_DEFINED 1
}

implementation epilog {
    static int multPoint(
        int a,
        Point *p)
    {
        // implementation of multPoint
        p->x *= a ;
        p->y *= a ;
    }
}
end

```

instance

The instance statement is used to define initial class instances. Instances are given names specified as `class@name`. The values of all attributes that do not have a defined default value must be specified. For attributes that have a defined default value, they are given that default value if not mentioned in the instance definition. Otherwise, the default value is overridden with any value given in the instance definition.

```

instance
Injector_Design@ihn4
    (Model_Name Model)          {"IHN4"}
    (MPa Min_delivery_pressure) {19}
    (MPa Max_system_pressure)  {35}
    (MPa Max_dissipation_pressure) {32}
    (Seconds Delivery_window)  {90}
    (Seconds Good_injection_duration) {9}
end

```

table

When there are a number of instances of a particular class, the instance statement can be tedious to use and obscures the nature of the instances as a group. In this case, the table command allows many instances to be defined in a tabular arrangement. Default values for attributes can be specified by using a hyphen (-).

```

table
Lubrication_Schedule
    (Name_t Name)
    (Duration Wait_interval)
    (Duration Monitor_interval)
    (Count Max_low_lube_cycles)
    (bool Default_continuous_operation)
    (Count Default_max_cycles)

@gearbox {"Gearbox"}    {210}   {45}   {8}    {true}   {5000}
@generator {"Generator"} {120}   {25}   {10}   {true}   {10000}
@shaft     {"Shaft"}      {90}    {30}   {10}   {true}   {10000}
@test2     {"Test2"}       {20}    {15}   {1}    {false}  {200}
end

```

Class Definition

In this section, we describe the pycca statements that are used to define the properties of classes. Class definitions are contained within enclosing class and end statements. For brevity, from here on we omit the surrounding class and end statements.

attribute

Attributes are declared in the same way as structure members in C, but without any punctuation. Following the lexical conventions, the C declarations appear in parentheses. The default value of an instance can also be specified. The default value is used only if no value is specified when an initial instance of the class is defined. Default values are used only when specifying the set of initial instances. For dynamically created instances, all attribute values are set by running code. The default value must evaluate to a valid C compile-time constant expression (because it will be used as an initializer). Because default value specifications are passed through to the compiler, they must be enclosed in braces ({}).

```
attribute (Model_Name Model)
attribute (MPa Min_delivery_pressure)
attribute (MPa Max_system_pressure)
attribute (MPa Max_dissipation_pressure)
attribute (Seconds Delivery_window) default {30}
attribute (Seconds Good_injection_duration)
```

reference

A reference is a special kind of attribute that is turned into a pointer to a specific class structure. The reference is given a name, which generally will be the same name as the relationship it implements (perhaps augmented with an annotation if the relationship is reflexive). The type of the attribute need not be specified because it can be deduced as a pointer to a structure that matches the class name.

Singular References

A single valued reference is used to implement traversal of a relationship on the side that is *one* or *one-conditional*. In the case of a singular reference, a simple pointer member holds the address of the referenced instance, and NULL may be used to indicate conditionality.

```
reference R4 -> Injector_Design
```

Multiple References

A multiple reference implements a relationship traversal for a side that is *many* or *many-conditional*. Using the ->> symbol will cause pycca to insert an array of pointers. The ->> notation comes in several alternate forms that are used to control the details of how the array of pointers is allocated. If ->> or ->>n is used to define the multiple reference, the class structure has a pointer member defined for it that will point to a NULL terminated array of class references. For example, the class fragment:

```
reference R2 ->> c1
```

If `->>c` is used to define the reference, the array of class references is *counted*, and the class structure will have two members defined for it, a pointer to the array and a count value.

```
reference R3 ->>c c5
```

For dynamic relationships, two alternatives are provided. If the reference specification is of the form `-ddd>>`, where the *ddd* stands for a set of decimal digits, then an array of class references is allocated in nonconstant memory. When the count form is used, the class instances will have an array of pointers of the specified size defined as part of their class structure.

```
reference R4 -20>> c7
```

If the reference specification is of the form `->>l`, a doubly linked list is set up to manage the multiple relationship. The memory for the links is easily managed, and referenced instances may be easily added and removed from the list. A set of macros, defined later in this appendix, is provided to hide the details of the linking, unlinking, and traversal mechanism.

```
reference R2 ->>l Control_Zone
```

subtype

The subtype statement is used to declare the necessary data structures to hold generalization relationship information in class structures. Generalization relationships can be implemented as either a reference or union generalization.

Reference Generalization

A class defines storage for a generalization relationship implemented by reference using the subtype ... reference statement. This statement gives a list of classes that are to be considered as the subtype classes.

```
subtype R1 reference
    Off_Duty_Controller
    On_Duty_Controller
end
```

Union Generalizations

In simple cases, typically a generalization hierarchy that is only a single level deep, it is usually more convenient to hold the subclasses of the superclass directly in the storage of the superclass instances as a union data type.

```
subtype R1 union
    Off_Duty_Controller
    On_Duty_Controller
end
```

machine

The `machine` statement is used to define a state model for a class to capture its life-cycle behavior. The statements used within the `machine` statement to define the state model are given in the State Model Definition section.

```
machine
    # statements to define a state model for the enclosing class
end
```

population

The `population` statement defines the storage characteristics of the instance population of a class. Class populations can be declared as `dynamic`, `static`, or `constant`. If no `population` clause is present, `population static` is assumed. A class population is `dynamic` if instances are created and deleted at runtime. Class populations that are not created and deleted at runtime are termed `static`. This implies that the number of instances does not change over the course of running the domain. A special case of `static` is a class population that is `constant`. A `constant` population is both static and read-only (its attributes are not updated).

```
population dynamic
```

slots

The `slots` statement defines the number of class instance storage slots that are allocated beyond those occupied by initial instances. Additional slots only be defined only for class populations that are `dynamic`.

```
population dynamic
slots 5
```

class operation

A class may define class-based operations by using the `class operation` statement. Class operations do *not* arise from model elements, but are implementation constructs provided to factor common code into a single function.

```
class operation
findByName(
    char const *name) : (struct Lubrication_Schedule *) {
    ThisClassRefVar(ls) ;
    PYCCA_selectOneStaticInstOfThisClassWhere(ls, strcmp(ls->Name, name) == 0)
    return ls == ThisClassEndStorage ? NULL : ls ;
}
```

instance operation

The `instance operation` statement is used to translate instance-based operations. Instance operations have an implicit first parameter that is a pointer to the instance on which the operation is to be performed. It is not necessary to declare the `self` variable, as pycca will insert it. However, because this is C, it is necessary to supply a value for the implied `self` parameter when invoking an instance operation.

```
instance operation Max_system_pressure() {
    ExternalOp(ALARM_Set_pressure_error)(PYCCA_idOfSelf)      ;
    ClassRefVar(Autocycle_Session, acs) = self->R2 ;
    InstOp(Autocycle_Session, Deactivate)(acs) ;
}
```

polymorphic event

The `polymorphic event` statement declares one or more events to be polymorphic. A polymorphic event can be declared only for a superclass.

```
polymorphic event
    e1
    e2
end
```

constructor

The `constructor` statement defines a constructor function for a class. This function is invoked with only the implicit `self` parameter whenever any instance of the class is created at runtime. If any class that contains a constructor also has an initial instance population specified, pycca will generate a function of the form `<domain name>_Ctor`, where `<domain name>` is replaced by the name of the domain. This function will invoke the constructor for all initial instances of all classes that have defined a set of initial instances and also have defined a constructor. It is up to the user to invoke this function during the application initialization phase (for example, in a domain operation that is invoked at initialization time).

```
attribute (int count)
constructor {
    self->count = 0 ;
}
```

destructor

The `destructor` statement defines a destructor function for a class. This function is invoked with only the implicit `self` parameter whenever any instance of the class is created at runtime.

```
attribute (int count)
destructor {
    reportCount(self->count) ;
}
```

State Model Definition

The statements in this section are used to define a state model for a class. State model definition statements must appear in enclosing machine and end statements. For brevity, we omit the enclosing machine and end statements.

state

The state statement defines a state in a state model. A state has a name and may have parameters that are passed to it as part of the event that causes the transition into the state. These parameters, if any, are listed in parentheses following the state name. A state has C code associated with it that is run when the state is entered. A set of preprocessor macros are provided to facilitate interfacing with the ST/MX runtime. The macros are discussed in the “Activity Macros” section.

```
state BUILDING_PRESSURE()
{
    if (self->Pressure > self->R4->Min_delivery_pressure) {
        PYCCA_generateToSelf(Above_inject_pressure) ;
    }
}
```

transition

The transition statement is used to specify the transition behavior of a state model. This statement lists the current state, event, and new state. Note that there is no separate list of events that are specified. Event names are defined when they appear in transition statements. The state name consisting of a period character (.) is the pseudo-initial state and defines the event in the transition to be a creation event.

```
transition BUILDING_PRESSURE - Above_inject_pressure -> INJECTING_AT_PRESSURE
```

default transition

The default transition statement defines the behavior of any transition not explicitly mentioned in a transition statement. The default transition can be IG and the event is ignored, or CH and the event causes an error condition. If no default transition statement is given in the definition of the state model, CH is assumed.

```
default transition IG      # ignore unexpected events
```

initial state

The initial state statement defines a default initial state for the state machines. If no initial state statement is given in a state model definition, the first state defined is taken to be the default initial state.

```
initial state SLEEPING
```

final state

The `final state` statement defines a state in which the instance of the class is automatically deleted by the runtime after the activity of the state has been executed.

```
final state perish
```

Activity Macros

This section gives a summary of the most frequently used preprocessor macros. This list is not comprehensive, and the full list can be found in the online materials. We divide the interface descriptions into groups according to the model elements upon which they operate.

Instance References

Frequently, there is a need to declare C variables that will hold a reference to class instances or refer to sets of class instances.

ClassRefVar(c, v)

Declare a variable suitable for holding a reference to a class instance.

-
- | | |
|---|---|
| c | The name of the class to which the instance refers. |
| v | The name of a C variable. |
-

ClassConstRefVar(c, v)

Declare a variable suitable for holding a reference to a class instance whose instance population is constant.

-
- | | |
|---|---|
| c | The name of the class to which the instance refers. |
| v | The name of a C variable. |
-

ClassRefSetVar(c, v)

Declare a variable suitable for referring to a set of class instances.

-
- | | |
|---|---|
| c | The name of the class to which the instance set refers. |
| v | The name of a C variable. |
-

ClassConstRefSetVar(c, v)

Declare a variable suitable for referring to a set of class instances whose instance population is constant.

c	The name of the class to which the instance set refers
v	The name of a C variable

Events

Signaling events is a common action for a state activity.

Events with No Parameters

For the common case where the event carries no additional parameters, several macros are provided to handle the three event types.

PYCCA_generate(e, c, i, s)

Signal an event to an instance.

e	The name of the event to signal.
c	The name of the class of the instance receiving the event.
i	A pointer to the instance that is to receive the event.
s	A pointer to the instance signaling the event. This may be <code>NULL</code> if the event originates from outside an instance context.

PYCCA_generateToSelf(e)

Signal an event to self.

e	The name of the event to generate
---	-----------------------------------

PYCCA_generatePolymorphic(e, c, i, s)

Signal a polymorphic event.

e	The name of the polymorphic event to signal.
c	The name of the class of the instance receiving the event.
i	A pointer to the instance that is to receive the event.
s	A pointer to the instance signaling the event. This may be <code>NULL</code> if the event originates from outside an instance context.

PYCCA_generateCreation(e, c, s)

Signal a creation event.

- e The name of the creation event to signal.
 - c The name of the class of the instance receiving the event.
 - s A pointer to the instance that is sending the event. This may be NULL if the event originates from outside an instance context.
-

PYCCA_generateDelayed(e, c, i, s, d)

Signal a delayed event.

- e The name of the event to signal.
 - c The name of the class of the instance receiving the event.
 - i A pointer to the instance that is to receive the event.
 - s A pointer to the instance signaling the event. This may be NULL if the event originates from outside an instance context.
 - d The delay time, in milliseconds.
-

PYCCA_generateDelayedToSelf(e, d)

Signal a delayed event to self.

- e The name of the event to generate.
 - d The delay time, in milliseconds.
-

Events with Parameters

Because events can carry supplemental data, to signal such an event is a three-step process:

Obtain an *event control block* (ECB). A different macro is used to obtain an ecb for each of the three event types.

Fill in the event data.

Post the event to the appropriate event queue.

As an example, assume that dog is an instance of class Dog and that the Bark event takes a single parameter, howLoud. Then the following will signal the Bark event to the dog instance of Dog.

```
MechEcb bark = PYCCA_newEvent(Bark, Dog, dog, self) ;
PYCCA_eventParam(bark, Dog, Bark, howLoud) = 20 ;
PYCCA_postEvent(bark) ;
```

PYCCA_newEvent(e, c, i, s)

Returns a MechEcb for an ordinary event.

- e The name of the event to signal.
 - c The name of the class of the instance receiving the event.
 - i A pointer to the instance that is to receive the event.
 - s A pointer to the instance that is sending the event. This may be NULL if the event originates from outside an instance context.
-

PYCCA_newEventToSelf(e)

Returns a MechEcb for an ordinary event where the target instance and the signaling instance are self.

- e The name of the event to signal.
-

PYCCA_newPolymorphicEvent(e, c, i, s)

Returns a MechEcb for a polymorphic event.

- e The name of the event to signal.
 - c The name of the class of the instance receiving the event.
 - i A pointer to the instance that is to receive the event.
 - s A pointer to the instance signaling the event. This may be NULL if the event originates from outside an instance context.
-

PYCCA_newCreationEvent(e, c, s)

Returns a MechEcb for a creation event.

- e The name of the event to signal.
 - c The name of the class from which an instance is to be created.
 - s A pointer to the instance that is sending the event. This may be NULL if the event originates from outside an instance context.
-

PYCCA_eventParam(ecb, c, e, p)

Retrieve the value of an event parameter.

ecb	A pointer to an event control block.
c	The name of the class of the instance receiving the event.
e	The name of the event to generate.
p	The name of the event parameter.

PYCCA_postEvent(ecb)

Place **ecb** on the non-self-directed event queue.

ecb	A pointer to an event control block.
------------	--------------------------------------

PYCCA_postSelfEvent(ecb)

Place **ecb** on the self-directed event queue. Note that creation events should not be posted to the self-directed event queue.

ecb	A pointer to an event control block.
------------	--------------------------------------

PYCCA_postDelayedEvent(ecb, d)

Post a delayed signal.

ecb	A pointer to an event control block.
d	The delay time in milliseconds.

Delayed Signal Operations

Delayed signals support both canceling and obtaining the remaining delay time.

PYCCA_cancelDelayed(e, c, i, s)

Cancel a delayed signal.

e	The name of the event to cancel.
c	The name of the class of the instance receiving the event.
i	A pointer to the instance that is to receive the event.
s	A pointer to the instance signaling the event. This may be NULL if the event originates from outside an instance context.

PYCCA_cancelDelayedToSelf(e)

Cancel a delayed event that was sent to `self`.

`e` The name of the event to cancel.

PYCCA_remainDelayed(e, c, i, s)

Retrieve the time remaining on a delayed signal.

`e` The name of the event to query.

`c` The name of the class of the instance receiving the event.

`i` A pointer to the instance that is to receive the event.

`s` A pointer to the instance signaling the event. This may be `NULL` if the event originates from outside an instance context.

PYCCA_remainDelayedToSelf(e)

Retrieve the time remaining on a self-directed delayed signal.

`e` The name of the event to query.

Instance Creation and Deletion

Class instances may be created and destroyed at runtime. These macros help with the interface to the mechanisms to handle instance management.

PYCCA_newInstance(c)

Create an instance of a class in the default initial state.

`c` The name of the class of the instance to be created.

PYCCA_destroyInstance(i)

Destroy an instance of a class.

`i` A pointer to the instance to be deleted.

Instance Selection

Pycca provides macros to iterate across instances and select them based on a certain criteria.

PYCCA_selectOneInstWhere(i, c, expr)

This macro expands to a linear search of class named *c* for the first instance where *expr* is true. The expanded code searches the storage pool for *c* stopping at the first instance that is in use and that satisfies *expr*. *Expr* is presumed to contain accesses to the attributes of *c* in the form of *i->a*. The value of the *i* variable is modified and at the end of the loop will either point to the first instance of *c* where *expr* evaluates to nonzero or will point past the end of the storage pool for the class (as given by the *EndStorage(c)* macro, that is, if *i >= EndStorage(c)*, then the search failed). Note that this macro tests whether the instance is currently allocated and therefore is useful only for classes that are either dynamically allocated or have a state machine.

<i>i</i>	The name of a variable that is a pointer to <i>c</i> class instances.
<i>c</i>	The name of the class of the instance corresponding to <i>i</i> .
<i>expr</i>	A C expression that will be interpreted as a Boolean.

PYCCA_forAllInst(i, c)

This macro is a convenience macro that sets up a loop that iterates across all instances of a class. The macro should be followed by a statement (possibly compound and enclosed in braces *{}*). In the statement, *i* will iteratively take on the value of every instance defined for the class *c*.

<i>i</i>	The name of a variable that is a pointer to <i>c</i> class instances.
<i>c</i>	The name of the class of the instance corresponding to <i>i</i> .

```
ClassRefVar(Autocycle_Session, acs) ;
PYCCA_forAllInst(acs, Autocycle_Session) {
    if (IsInstInUse(acs)) {
        PYCCA_generate(Created, Autocycle_Session, acs, NULL) ;
    }
}
```

PYCCA_forAllRelated(v, i, r)

This macro is a convenience macro that sets up a loop that iterates across the instances related to the class. This macro assumes that the related instances were declared using the *->>c*, syntax (the related instances are of the counted type). The macro should be followed by a statement (possibly compound and enclosed in braces *{}*). In the macro, *v* should be declared as *ClassRefSetVar* or a *ClassRefConstSetVar*. Then *v* is iterated over the set of related instances.

<i>v</i>	The name of a reference set variable.
<i>i</i>	The name of an instance reference variable.
<i>r</i>	The name of the relationship across which the iteration occurs.

```

ClassRefConstSetVar(Injector, myinjs) ;
    PYCCA_forAllRelated(myinjs, self, R5) {
        ClassRefVar(Injector, inj) = *myinjs ;
        ClassRefVar(Autocycle_Session, acs) = inj->R2 ;
        InstOp(Autocycle_Session, Deactivate)(acs) ;
    }
}

```

PYCCA_forAllRelatedTerm(v, i, r)

This macro is the same as PYCCA_forAllRelated() except that the relationship must have been declared by using the ->>n syntax (the relationship storage consists of a NULL terminated array of instance pointers).

v	The name of a reference set variable.
i	The name of an instance reference variable.
r	The name of the relationship across which the iteration occurs.

PYCCA_forAllLinkedInst(o, r, l)

Iterate over the instances of a one-to-many relationship implemented using linked lists. This macro is similar to PYCCA_forAllRelated except that the relationship must have been declared by using the ->>1 syntax (the relationship storage is implemented with linked lists). This macro expands to a loop construct in which all the instances related to o across r are visited. The link variable, l, is successively assigned values of the links related on the many side to o. The value of the link variable, l, is *not* a pointer to an instance. The instance pointer must be recovered by using the PYCCA_linkToInstRef() macros described here.

o	A pointer to a one-side instance.
r	The name of the relationship.
l	The name of a variable of type rlink_t *.

```

rlink_t *czlink ;
PYCCA_forAllLinkedInst(ondc, R2, czlink) {
    ClassRefVar(Control_Zone, found) =
        PYCCA_linkToInstRef(czlink, Control_Zone, R2) ;
    if (strcmp(rcvd_evt->zone, found->Name) == 0) {
        hoff_zone = found ;
        break ;
    }
}

```

PYCCA_linkToInstRef(l, c, r)

This macro converts a link pointer of type `rlink_t *` that is a link in relationship `r` to a pointer to an instance of class `c`.

<code>l</code>	The name of a variable of type <code>rlink_t *</code> .
<code>c</code>	The name of the class on the many side of the relationship to which <code>l</code> refers.
<code>r</code>	The name of the relationship.

Instance Identifiers

When invoking external operations, it is useful to use have a means to identify an instance of a particular class outside a domain. The pointer value of the instance is *not* suitable for this purpose, but the array index of the instance in its storage pool is satisfactory. The macros in this group provide a means of generating a small integer value for an instance that can be used as an identifier external to the domain or to translate an instance identifier into a pointer reference to the instance.

PYCCA_idOfSelf

Generate an integer identifier for the `self` reference.

```
ExternalOp(ALARM_Clear_lube_level_very_low)(PYCCA_idOfSelf) ;
```

PYCCA_idOfRef(c, r)

Generate an integer identifier for a reference `r` of class `c`.

<code>c</code>	The name of the class.
<code>r</code>	The reference value for a member of class <code>c</code> .

PYCCA_refOfId(c, i)

Return an instance reference to an instance of class `c` that is identified by the integer identifier `i`.

<code>c</code>	The name of the class.
<code>i</code>	An integer identifier for an instance of the class.

PYCCA_checkId(c, i)

Generate an invocation of the `assert()` macro to test that the identifier `i` is valid for class `c`. This is useful if a domain operation accepts an integer identifier for an instance and wants to assert its validity.

<code>c</code>	The name of the class.
<code>i</code>	An integer identifier for an instance of the class.

```
PYCCA_checkId(Autocycle_Session, sessionId) ;
ClassRefVar(Autocycle_Session, session) =
    PYCCA_ref0fId(Autocycle_Session, sessionId) ;
if (IsInstInUse(session)) {
    InstOp(Autocycle_Session, Suspend)(session) ;
}
```

Navigating Generalizations

When a generalization relationship is implemented as a union data type, the instances of the subtypes do not use a pointer to navigate to the supertype. Rather, it is necessary only to *up cast* the `self` pointer to obtain the pointer to the supertype.

PYCCA_unionSupertype(sub, supc, r)

Navigate to the supertype for instances contained in a union.

<code>sub</code>	A pointer to the subtype instance. This is frequently <code>self</code> .
<code>supc</code>	The name of the supertype class.
<code>r</code>	The name of the relationship.

■ Warning Up casting in this context is just as dangerous and error prone as up casting in any other context. If `sub` is not a subtype of the class `supc`, this macro will silently yield an incorrect result because it involves an explicit cast that the compiler cannot check.

Navigation to the subtype is achieved by taking the address of the subtype member within the generated structure. This macro hides the naming conventions for the subtype member.

PYCCA_unionSubtype(*sup*, *r*, *subc*)

Navigate to the subtype instance contained in a union.

sup A pointer to the supertype instance. This is frequently `self`.

r The name of the relationship.

subc The name of the subtype class.

```
ClassRefVar(On_Duty_Controller, new_ondc) =
    PYCCA_unionSubtype(new_controller, R1, On_Duty_Controller) ;
```

PYCCA_referenceSubtype(*sup*, *r*, *subc*)

Navigate to the subtype instance by pointer reference.

sup A pointer to the supertype instance. This is frequently `self`.

r The name of the relationship.

subc The name of the subtype class.

PYCCA_isSubtypeRelated(*sup*, *supc*, *r*, *subc*)

Determine whether a supertype instance is currently related to an instance of the given subtype. Returns `true` if *sup* is related to an instance of the subtype class *subc* across the relationship *r*.

sup A pointer to the supertype instance. This is frequently `self`.

supc The name of the supertype class.

r The name of the relationship.

subc The name of the subtype class.

PYCCA_migrateSubtype(*i*, *supc*, *r*, *subc*)

Change the subtype of a supertype instance. For a generalization relationship, the supertype maintains an encoded value of the subtype to which it is currently related. This macro changes that encoded value.

i Instance pointer to a supertype.

supc Name of the supertype class.

r Name of the generalization relationship.

subc Name of the subtype class.

```
PYCCA_migrateSubtype(self, Air_Traffic_Controller, R1, Off_Duty_Controller) ;
```

PYCCA_initUnionInstance(sup, r, subc)

Initialize a subtype instance that is contained in a union-based supertype to its default initial state. When subtype migration happens in subtypes contained in a union, if the subtype has a state model, then it is necessary to initialize the architectural structures to be those of the new subtype. This macro accomplishes that. Note that this macro does not run any constructor of the subtype. For subtypes that have a constructor, an explicit call is required.

sup A pointer to the supertype instance. This is frequently `self`.

r The name of the relationship.

subc The name of the subtype class.

PYCCA_relateSubtypeByRef(s, supc, r, t, subc)

Relate a supertype instance to a subtype instance when the relationship is being stored by reference.

s A pointer to the supertype instance. This is frequently `self`.

supc The name of the supertype class.

r The name of the relationship.

t A pointer to the target subclass instance.

subc The name of the subtype class.

APPENDIX D



Bibliography

Books

- Stephen J. Mellor and Marc J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley (2002), ISBN 0-201-74804-5.
- Chris Raistrick et al., *Model Driven Architecture with Executable UML*, Cambridge University Press (2004), ISBN 0-521-53771-1.
- Leon Starr, *Executable UML: How to Build Class Models*, Prentice Hall (2001), ISBN 0-13-067479-6. Note: This edition is out of print, but a new, extended edition covering all the model facets is in the works for 2018, publisher to be determined.
- Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall (1988), ISBN 0-13-629023-X.
- Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall (1991), ISBN 0-13-629940-7.
- Lex de Hann and Toon Koppelaars, *Applied Mathematics for Database Professionals*, Apress (2011), ISBN 1-43024-248-1. 1430242841.
- Fredrick P. Brooks, *The Mythical Man Month (Anniversary Edition with four new chapters ed.)*, Addison-Wesley (1995).
- ACM Letters on Programming Languages and Systems 1, 3 (Sep. 1992), 213-226.

Papers

- E.F. Moore, “Gedanken-Experiments on Sequential Machines,” *Automata Studies*, Princeton University Press, Princeton, N.J. (1956), pp. 129–153.
- George H. Mealy, “A Method for Synthesizing Sequential Circuits,” *Bell System Technical Journal* (1955), pp. 1045–1079.
- Christopher W. Fraser, David R. Hanson and Todd A. Proebsting, *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems 1, 3 (Sep. 1992), 213-226.
- <http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf>.

Articles

- Leon Starr, “[How to Build Articulate UML Class Models and Get Real Benefits from UML](#)” (2008) Originally published at www.uml.org/news.htm. Also at www.modelint.com/mbse and www.executableuml.org.
- Leon Starr, “[Time and Synchronization in Executable UML](#)” (2008) Paper and video available at www.modelint.com/mbse and www.executableuml.org

Index

Symbols

`#=`, 256, 258
`..=`, 258
`.=,` 258
`=`, 258

A

`abort()`, 75
`abort()` C function, 75
Action
 model script, 29, 30, 39, 183
 parallel execution, 228
 on sets, 229
 translation, 39, 48, 52–53
Action function
 pointer to, 70, 71
Action language
 Alf (Action Language for Foundational UML), 27, 227
 ASL (Action Specification Language), 227
 to C, 233
 vs. coding, 227
 desirable characteristics, 227–230
 implementation bias, 228
 mapping to code, 220
 OAL (Object Action Language), 227
 predicate logic, 230
 vs. programming language, 227
 relational algebra, 229, 230
 Scrall (Starr’s Concise Relational Action Language), 255–263
sequence
 small, 227, 228
sequence of computation, 227
set at a time, 229
Small (Shlaer-Mellor Action Language), 227
syntax and semantics, 226
text vs. diagram, 219, 220
text vs. graphics, 219

Action translation
 commented out, 48
 preprocessor macros, 40
Activity
 class based operation
 for implementation only, 107
 class method, 93
 concurrency, 26
 data flow diagram, 26, 228
 derived attribute definition, 248
 domain operation, 248, 249, 254
 entry/, 22
 external entity operation, 248
 state, 21, 26, 49, 52, 53, 55, 63, 64, 70–74, 76, 95, 104, 108, 118, 162, 172, 174, 188, 191, 221, 251, 262, 277
 synchronization, 21
 translation, 52–57
ADC (analog to digital converter)
 block diagram, 114
Ad hoc queries
 database management system, 107
Aggregation, 2
Agile
 iteration, 214
 Manifesto, 213
Aircraft classes
 airplane, 161
 Fixed Wing aircraft, 161
 helicopter, 161
 Rotary Wing aircraft, 161
 VTOL aircraft, 163
Air Traffic Control. *See* ATC (Air Traffic Control)
ALS (Automated Lubrication System)
 code size, 190–191
 description, 78–79
 initial population, 86
ALS classes
 Autocycle Session, 85, 96–98, 101–103
 injector, 82–84
 Injector Design, 82–83

■ INDEX

- ALS classes (*cont.*)
 - Lubrication Schedule, 85
 - machinery, 82
 - Reservoir, 82
- ALS domains
 - Alarms, 80, 190
 - Lubrication
 - description, 81
 - translating, 80
 - SIO (Signal I/O), 80
 - user-interface, 80, 81
- ALS state models
 - Autocycle Session, 89–92, 102
 - Injector, 86–89
 - Reservoir, 91
- ALS state tables
 - Autocycle Session, 93, 96
 - Injector, 94, 95
 - Reservoir, 97
- Analysis paralysis, 77
- Analysis skill
 - vs.* modeling, 218
- Application logic, 3, 4, 7, 10, 192, 219, 226, 227
- Aspect oriented programming (AOP), 221
- Assigner
 - example, 252
 - implementation, 120–121
 - multiple, 252
 - state model
 - definition, 252
- Assignment operator (.=), 256–258
- Association
 - access direction, 33
 - conditionality, 17
 - decomposition, 33
 - definition, 245
 - implementation choices, 32, 33
 - implementation considerations, 32
 - link, 260
 - model script, 30
 - multiplicity, 17, 33, 200, 233, 262
 - phrase, 16
 - referential attribute, 16, 32
 - translation, 97–98
 - verb phrase *vs.* role, 16
- Association class. *See also* Associative relationship
 - decomposition, 98, 99
 - definition, 246
 - translation, 97–98
- Association translation
 - to many linked list, 45
 - unconditional data integrity, 56
- Associative relationship
 - code, 100–101
 - decomposition, 99
 - navigating, 99–101
- ATC (Air Traffic Control)
 - application, 13–15
 - class model, 15–17
 - execution scenario, 15
 - limitations, 77
- ATC classes
 - Air Traffic Controller, 44, 53, 55, 61, 192, 193
 - Control Zone, 14
 - On Duty Controller, 18, 247
 - Duty Station, 14, 43, 44
 - Off Duty Controller, 18, 247
 - Shift Specification, 17
- ATC domains
 - Radar Tracking, 77
- ATC state models
 - Duty Station, 21
- ATC translation
 - Air Traffic Controller state model, 51–52
 - Air Traffic Controller superclass, 34
 - conditional association using NULL, 42
 - On Duty Controller subclass, 44
 - Duty Station class, 51
 - Duty Station class to C structure, 43
 - Duty Station state model, 48–51
 - generalization to C union, 34
 - identifier, 39
 - Logging In activity, 53
 - Logging Out activity, 55
 - to one association to pointer, 43
- Attribute
 - definition, 242
 - descriptive, 16, 32, 35, 47, 97
 - I and R tags, 242
 - identifying, 16, 32, 42, 120, 121, 136, 198, 208, 209
 - notation, 5
 - referential, 16, 24, 32, 97, 202, 206, 207, 209, 219, 225, 226, 244, 246, 260
 - uninitialized, 229

■ B

Battery-operated environment, 69

Berkeley DB

Btree, 197, 203

cursor

for navigation, 171

database, 195–198

environment, 197

foreign key index, 206, 210

join

for navigation, 206

key, 188

mapping domain data, 195–207

model alternate identifiers, 200

model identifiers, 32

- navigation, 171
- secondary index, 199, 200, 202, 204–206, 210
- using C struct, 193
- Bridge**
 - assertions in code, 190, 219
 - asynchronous signals, 138, 150, 261–262
 - bridge table, 134–137, 139–141, 144–151, 160
 - code, 106, 137, 146, 147, 156, 160, 190, 223, 233, 234
 - counterpart instance, 132
 - definition, 239
 - half tables, 134–137, 144, 145, 160, 223
 - ID parameters, 137, 139, 145
 - implementation, 146, 153
 - implicit *vs.* explicit, 221
 - instance bridge table, 137, 148
 - mapping optimization, 132–134
 - mapping tables, 136, 152, 223
 - pull strategy, 153
 - push strategy, 153
 - semantic gap, 131, 135, 137, 160
 - table search
 - array indexing, 148, 150, 151
- Bridging, 1, 10, 78, 112, 113, 129, 132, 140, 146, 147, 191, 220, 221, 233, 266
- bsearch(), 109, 193
 - for performance, 193

C

- C**
 - familiarity, 1, 26
 - files, 38
 - passed through, 57, 266, 267, 269, 271
 - preprocessor macros, 30, 38, 40, 55, 57, 107, 265, 275, 276
 - in this book, 1
- C++**
 - elaboration, 2
 - high-level, 2
 - why not, 8
- Can't happen. *See* Transition
- C99 designated initializer
 - for bridge initialization, 136
- Class**
 - abstraction, 185
 - definition, 185
 - diagram, 14, 15, 81, 114, 124, 165, 166, 170, 196, 200, 257
 - implementation, 186
 - model script, 29, 30, 38, 39, 183
 - notation, 257
 - translation, 32
 - unassociated, 24
 - without state model, 19–20

- Class based operation**
 - none in xUML, 107
- Class collaboration diagram**, 91, 110

Class method, 29, 30, 38, 39, 183

Class model

- model script, 30

Class operation

- implementation specific
- code, 192–194

C main function, 63

Code

- derivation, 230

Code generation

- template driven, 185, 188

Code generator

- task of designing, 3, 149, 210

Competitive association

- first come first served, 121

Computations

- long duration, 64

const, 41

- for immutable values, 41

Constraints

- relational *vs.* OCL, 226

Creation

- asynchronous, 101

event

- model script, 101

- translation, 101

Cross platform development, 74

D

- Database**, 5, 8, 16, 60, 104, 107, 188, 189, 195–202, 204, 206, 207, 209, 210, 219, 222–225, 229, 234, 236

Data flow

- UNIX pipe syntax, 288

Data flow diagram. *See* Activity

Data type

- definition, 243

- 2D Point, 230, 243

- examples, 243

- matrix algebra, 230

- model script, 29, 30, 38, 39

- model *vs.* implementation, 31

- operations, 230

- typedef, 198

Data type translation

- typedef, 198

Debugging, 74, 190, 231

Delayed event

- queue, 66–69

Deletion

- asynchronous, 91, 101, 103–104

Desktop computer *vs.* micro-controller environment, 8, 9
 Device access entity, 113
 Device units, 113
 Diagnostic lubrication schedule, 78, 85, 216
 Documentation literate programming, 46, 220 useful, 219
 Domain application, 77 application level, 77, 81 benefits, 129 as black box, 130–132 client and service roles, 220, 239 configuration data, 221 delegation, 77, 109, 220 dependency, 106, 132, 220, 221 distinct subject matter, 59, 80, 81 element identifier, 147–148 high level and low level, 2, 81 interaction, 81 layering view, 81 lubrication, 81 class model, 81 external entity, 133 Lubrication domain marking, 147 populate mapping, 143 portal, 147, 156, 158 requirements source, 156 reuse, 147 satisfies requirement for service, 136 semantic gap, 160 separation of concerns, 111, 112, 127, 129 specifies need for service, 133 subject matter *vs.* functional partitioning, 81 vocabulary, 238
 Domain chart for ALS, 78–82 dependency arrows, 80, 132 platform independence, 4, 215, 227 refactoring, 15, 28, 217, 218
 Domain operation C function, 52 defined, 93 naming convention, 38, 105, 109, 146, 156, 285 Write point, 132, 134–137, 149, 223
 DSL (domain-specific language) pycc, 8
 Tcl or Python, 223 internal, 223

■ E

Edge detection falling, 123 rising, 123 EEPROM, 229 Elaboration abrupt failure, 2 action language, 227 failure of, 2 gradual failure, 2 model destruction, 3 Encapsulation domain level, 220 Engineering units, 113, 151 Error fatal custom handler, 75–76 handling, 75–76 types, 75 Event calculate sequences, 74 cancel delayed, 67, 69, 88 consuming, 164, 170, 250, 251 data to dispatch diagram, 69, 71 delayed, 67–69 delayed event queue, 66–69 dispatch data, 69, 72 delayed, 69 generated code, 70 locating action table, 69, 70 locating transition table, 70 in flight error, 67 ignored Boolean attribute trick, 168 self-directed, 167 software lock, 167 transitory state, 167 local and non-local, 94, 95 numbering, 65, 67, 70 parameters pass by value, 67 pointer to, 125 pending, 27 polymorphic (*see* (Polymorphic event)) response possibilities, 36, 37 self and non self-directed, 67, 68, 167, 191, 259 signaling, 66–67 too late or too early, 36 tracing dispatch, 73 types, 65, 175, 176, 178 enum, 174, 175 unexpected, 27, 35

Event control block (ECB)
 MechEcb, 65
 polymorphic event, 174

Event, delayed, 27, 66–69, 88–90, 94, 96, 147, 191, 278, 281

Event map
 generated code, 176
 index into, 179
 polymorphic event dispatch, 179, 181, 183

Event queue
 free, 66
 imminent, 66–70
 main loop, 64

Event specification, 20, 139, 225

Example external entities
 NOTIFY, 144, 145, 151, 154
 SIO
 functions, 148, 149

Executability, 4, 15

Execution cycles
 limited, 60

External bus, 113

External entity
 asynchronous signals, 138, 150, 261–262
 definition, 131
 functions
 injection control, 148–150
 pressure alert, 154–156
 pressure attribute, 151–154
 pressure monitor, 150–151
 implicit ID, 139
 operation, 131, 133–137, 146, 147, 149, 151, 154, 221, 223, 230
 naming convention, 146
 portal function, 147, 150, 153, 154, 156, 160
 as proxy for service domain, 131, 146
 synchronous operation, 137

External entity function
 implementation of EE operation, 148–151

External entity operation
 C function, 147, 149
 inject, 136, 137
 parameter, 133
 mapping, 135, 137
 return value, 133
 synchronous and asynchronous, 146

External operation
 defined, 106

F

Facet
 actions, 15
 class model, 14, 15
 pycca workflow, 232

state model, 14, 15
 translation order, 30

Faults
 modeled, 78

Final pseudo-state
 instance deletion, 103

Final states
 array, 71

fUML (Foundational UML)
 and xUML, 225

Functional
 decomposition, 81
 partitioning, 81

G

Gap
 bridging, 1

Generalization
 composed form
 compound generalization, 164
 multiple generalization, 163
 repeated specialization, 162
 disjoint/complete tags, 17, 161
 disjoint set interpretation, 163
 disjoint union, 17
 example, 247
 implementation alternatives, 34
 migration
 state model, 172
 model script, 34
 navigation
 empty reference, 162
 pycca, 171
 pointer references
 pycca, 170
 polymorphism, 161–164, 166, 169–171, 174–176
 reference, 34
 reference implementation, 170
 referential attribute, 247
 relating super and subclass instances, 169, 170, 174, 183
 spanning subclass, 163
 state models in subclasses, 166
 subclass determination
 Launch type, 166
 translation, 43, 44, 47, 54
 type inheritance, 161
 union, 34
 union implementation, 44, 172
 XOR (exclusive or) constraint, 247
 xUML and UML, 161, 183

Go language
 target for concurrency, 228

H

Half table

- attribute to attribute, 136
- generic model elements, 137
- inject ID parameter, 137
- inject instance, 136
- inject operation, 135
- instance to instance, 135
- instance to instance mapping, 136
- parameter to attribute, 137
- parameter to parameter, 136, 137
- population, 223
- pressure monitor, 138–139
- pressure update, 140–141
- runtime and static population, 136

Hardware access library, 190, 191

Hardware controls, 113

Hardware engineering

- modeling analogy, 3
- workflow, 3

Hash table

- for performance, 194

HierarchyDispatch, 175, 179–181

- typedef, 175

Hierarchy dispatch block (HDB), 175

Hop operator (/), 24

Human intervention

- pycca workflow, 233

I

Identifier

- definition, 244
- discarded, 29
- implementation-defined, 29, 32, 35
- invented *vs.* UML implicit, 244
- notation, 244
- real world meaning, 32
- real world *vs.* invented, 244

Identifying attribute

- referenced, 209

Identity constraint, 16

Ignore event. *See* Transition

Impedance mismatch elimination

- TclRAL, 189

Implementation detail, 2, 185, 252

Implementation technology

- choosing, 210

in., 24, 257, 259

- input parameter prefix, 24

Inheritance, 2, 5, 17, 34, 161, 183, 227

Initial instance population

- DSL, 222, 223

- non-modelers, 222

- tables *vs.* action language, 221

Initial pseudo-state

- instance creation, 101

Injection sequence, 83

Instance

- class data, 62
- current state, 61
- data structure, 62
- dynamic, 63
- memory, 61, 63, 67
- pre-existing, 18, 19
- slot allocation, 35
- slot in use, 75
- ST/MX view, 61–63

Instance creation

- asynchronous, 172
- migration, 172

Instance mapping

- initialized C array variable, 150

Instance operation

- translation of class method, 105

Instance search

- generic *vs.* specific strategy, 107

Intellectual property, 4, 220

Interchange

- format, 219
- relational database, 219
- UML *vs.* xUML, 219
- XMI, 219
- XML or JSON, 225

Interrupts

- handling, 60

J

Java

- elaboration, 2
- high-level, 2
- target language, 2

K

Key/value pair

- store, 229

L

#line directive, 187, 231, 265

Link action (&), 26, 260

Linked list

- iteration, 101

Link type

- linked list, 45
- in reference statement*, 42, 43

Lockout

- machinery, 79, 87, 90

Lubrication cycle, 79, 84–85

M

- Magic
 - modeling tools, 1
 - model translation, 10
- Main loop
 - activity diagram, 64
- Management *vs.* technical process, 214
- Mapping
 - input points, 145
 - range limits, 144
- Marking
 - actuator control, 133
 - alert event, 143
 - annotations, 215, 224
 - platform model, 224
 - pressure monitor, 138
 - in pycca, 224, 234
 - repository, 215
 - sensor attribute, 140
 - task, 214, 234
 - threshold attribute, 142, 144
 - transparent sheet, 224
- Marking and mapping, 132–137, 146, 160
- Mars climate orbiter, 113
- Mars probe classes, 164
 - Mars Probe, 164
 - Nuclear Powered, 164
 - Rover, 164
 - Solar Powered, 164
 - Stationary, 164
- Mathematical foundations, 9, 226, 229
- MBSE (model-based software engineering), 215
- me, 24, 53, 259, 262
 - self instance action keyword, 2
- MechEcb, 279
 - typedef, 65
- MechEventType, 65, 174, 176
 - enum, 174
- MechInstance, 61, 174
 - struct, 61–63
 - typedef, 61
- Memory
 - limited, 60
- Memory map, 113
- Memory usage
 - example application, 190, 191
 - model execution domain, 191
 - RAM and flash, 190–191
- Metamodel
 - BridgePoint-xtUML, 225
 - constraints, 224, 225
 - database schema, 224, 225
 - iUML, 225
 - miUML
 - tool independent, 225
 - relational theory, 225
 - SQL, 225
 - tool independent, 225
 - xUML, 224–225
- Migrate, 24, 26, 56, 166, 261
 - action, 24
- Model
 - application logic, 3, 4, 7, 192, 226, 227
 - deriving code, 4
 - destruction of, 3
 - detail, 4, 10
 - executable, 4, 6, 7, 13–29, 77, 185, 186, 213, 214, 235
 - high level, 2, 5
 - intellectual property, 4
 - language, 1, 4, 5, 7, 14, 77, 224–226
 - mental execution, 5, 6
 - platform-independent, 1, 4, 224
 - running, 1, 4, 9
 - translation, 1, 4, 7–11
 - uses, 14, 219
 - value of, 3
- Model descriptions, 219, 220
- Model editor
 - draw tools, 219
- Model execution
 - configurable, 60
 - domain
 - implementation, 59, 60
 - microcontroller constraints, 60, 63, 74
 - no universal domain, 60
 - performance, 60
 - persistent non-volatile storage, 195
 - rules
 - invariance, 210–212
 - runtime
 - C preprocessor macros, 38
 - subject matter of, 59, 60
 - Model execution run-time. *See* MX run-time
 - Model facets, 26, 39, 255
 - Modeling
 - required skills, 213, 215, 216, 218
 - Model parsing
 - analogy, 231
 - Model repository, 214, 215, 225
 - Model script, 29, 30, 38, 39, 183
 - Model script statements. *See* Pycca statements
 - Model validation
 - using metamodel database, 225
 - Monitoring interval
 - lubrication, 84
 - Monitor pressure, 78
 - Moore state diagrams, 22, 49
 - Multiplexer, 114
 - MX. *See* Model execution
 - MX run-time, 104, 215, 275

N

Navigation

- generalization, 34
- pointer arithmetic, 34
- pointers, 34

Next state

- determine, 64

Non-xUML models, 214

Notation and complexity, 226

OObject. *See* Instance

ObjectDispatchBlock, 50, 70, 72

- typedef, 70

Object files, 190, 215

Object-oriented

- vs.* mathematical foundations, 229
- perspective, 226
- programming, 2, 5, 7, 9, 62, 107, 226, 227
- semantics, 5

OCL (Object Constraint Language), 226

ODB (object dispatch block), 70, 180

P, Q

Pair programming, 234

Panic

- on can't happen, 71

Parallel processing, 4

Parsing actions

- AST (abstract syntax tree), 233

PDB (polymorphic dispatch block)

- locating, 178

Perfect hash function, 109

Performance

- code size, 190, 191
- execution speed, 192
- immutable values, 41
- initialization, 35
- measurement *vs.* guessing, 109
- model execution domain, 60
- optimization, 109
- pointers, 54
- pycca, 189–211
- scale, 211
- trade-off, 141

Platform

- ARM Cortex-M3, 74, 189
- Linux, 75, 188
- macOS, 75, 188
- other targets, 190, 213
- POSIX, 74–75
- server/desktop target, 195

supporting diverse and challenging, 226

TI MSP-430, 74

Windows, 75, 188

Platform constraints

- microcontroller, 60

Platform independence, 4, 9, 77, 215, 224, 227

Platform-independent

- execution rules, 4

model, 4

synchronization, 4

Platform-independent and specific tasks, 214

Platform model

Attribute, 186

Berkeley DB, 207, 209, 210

class, 186–188, 208

ClassRef, 186

code generation

- two step process, 231

Data Element, 186

domain, 186

vs. executable model, 185

File and Line attributes, 187

vs. metamodel, 185

populate with DSL, 185, 188, 211

relational schema, 189

SingClassRef, 186, 187

SubtypeRef, 186

task, 195

Platform-specific features, 224

PolyDispatchBlock

- struct, 175

PolyEventMap

- typedef, 176

Polymorphic event

- definition, 162

- delegation, 169

dispatch

- data used, 177

- event map, 178, 179

- eventType, 175

- generated code, 179

PDB (polymorphic dispatch block), 174

enumeration, 175

event map, 178, 179

example

Arming distance, 168

Cleared tube, 166, 167, 169

Fire, 166–169

Hitch load, 163

Hover, 163

Park, 161

Pass, 161

Recall, 166–169, 172, 180

Sleep mode, 164

Taxi, 163

- model script, 183
- parameters, 174, 183
- signaling
 - pycca, 174
- translation, 169–174
- Polymorphism**
 - model *vs.* programming, 161
 - simple dynamic, 161
- PolyStorageType**
 - enum, 175
- Populating models**, 188, 234
- Population**
 - attribute initialization, 46
 - constant, 35, 273
 - dynamic, 35, 46, 194
 - identifying attribute
 - Berkeley DB example, 207
 - initial instance
 - C array, 35
 - model script, 30
 - translation, 46
 - relationship initialization, 46
 - runtime *vs.* translation time, 223
 - static, 273
- Population instance name (@)**, 35
- Population instance reference**, 35
- Portal**
 - capabilities, 147
 - class information, 157
 - element numbering, 147–148
 - function, 147, 148, 150, 153, 154, 156, 158, 159, 191
 - how it works, 134, 135
 - limitations, 154
 - variable, 147, 156, 159
- Postgres**
 - miUML metamodel, 225
- Programming language**
 - as a domain, 107
- Pycca**
 - availability, 9, 188
 - DSL (domain-specific language), 9, 185, 187, 188, 233, 265
 - file, 9, 40
 - implementation
 - lex and yacc equivalents, 189
 - Tcl language, 185, 188, 189
 - TclRAL (Tcl relational algebra language), 189
 - language overview, 265–287
 - overview, 39
 - parser, 187
 - pass your C code along, 30
 - platform specificity, 9
 - program
- design, 188
- DSL (domain-specific language), 187, 188
- platform model, 185–189, 234
- source code and documentation, 185
- syntax**
 - code, 39
 - comments, 39
 - symbol name, 39
 - variable declaration, 39
 - transparency, 9
- Pycca language**
 - comments, 266
 - C variables, 267
 - keywords and tokens, 267–268
 - pass along C code, 267
 - whitespace, 266
- Pycca macros**
 - ClassConstRefSetVar, 277
 - ClassConstRefVar, 276
 - ClassRefSetVar, 276, 282
 - ClassRefVar, 54, 276
 - PYCCA_cancelDelayed, 280
 - PYCCA_cancelDelayedToSelf, 281
 - PYCCA_checkId, 285
 - PYCCA_createInstance, 101
 - PYCCA_destroyInstance, 103, 281
 - PYCCA_eventParam, 280
 - PYCCA_forAllInst, 282
 - PYCCA_forAllLinkedInst, 283
 - PYCCA_forAllRelated, 282–283
 - PYCCA_forAllRelatedTerm, 283
 - PYCCA_generate, 277
 - PYCCA_generateCreation, 103, 278
 - PYCCA_generateDelayed, 278
 - PYCCA_generateDelayedToSelf, 278
 - PYCCA_generatePolymorphic, 170, 277
 - PYCCA_generateToSelf, 277
 - PYCCA_idOfRef, 284
 - PYCCA_idOfSelf, 284
 - PYCCA_initUnionInstance, 287
 - PYCCA_isSubtypeRelated, 286
 - PYCCA_linkToInstRef, 283, 284
 - PYCCA_migrateSubtype, 286
 - PYCCA_newCreationEvent, 279
 - PYCCA_newEvent, 279
 - PYCCA_newEventToSelf, 279
 - PYCCA_newInstance, 281
 - PYCCA_newPolymorphicEvent, 279
 - PYCCA_postDelayedEvent, 280
 - PYCCA_postEvent, 280
 - PYCCA_postSelfEvent, 280
 - PYCCA_referenceSubtype, 286
 - PYCCA_refOfId, 284
 - PYCCA_relateSubtypeByRef, 287
 - PYCCA_remainDelayed, 281

■ INDEX

Pycca macros (*cont.*)

- PYCCA_remainDelayedToSelf, 281
- PYCCA_selectOneInstWhere, 282
- PYCCA_selectOneStaticInstWhere, 53, 54
- PYCCA_unionSubtype, 286
- PYCCA_unionSupertype, 285

Pycca program

- options, 265–266
- output file, 266

Pycca script

- lexical conventions, 266–268

Pycca statements

- attribute, 43, 97, 271
- class, 31, 32, 271–272
- class operation, 273
- constructor, 274
- default transition, 275
- destructor, 274
- domain, 40, 41, 268–270
- domain operation, 268–269
- external operation, 269
- final state, 276
- grouping
 - cumulative, 40
 - order, 40
- implementation epilog, 269–270
- implementation prolog, 42, 269–270
- initial state, 275
- instance, 270
- instance operation, 274
- interface epilog, 269–270
- interface prolog, 269
- machine, 273, 275
- polymorphic event, 274

population

- constant, 273
- dynamic, 273
- static, 273

reference

- multiplicity, 42

state, 48

- subtype
 - reference, 272
 - union, 272
- table, 270
- transition, 275

Python, 223, 234

■ R

Range

- in and out of states, 123, 124

Raw device value, 140

RDMS (relational database management system),
104, 225, 229

Referential attribute

- pointer substitution, 32

Referential integrity

- foreign-key index, 206

Relational theory

- complex values, 230
- model of data, 207

Requirements

- modeling and translation, 7

Response time

- real-time constraints, 60

Reuse

- bridging, 220, 221
- domains, 127
- rule of three, 223
- xUML, 226

Risk analysis

Round trip, 2

RTOS and ST/MX

- memory comparison, 191

RTOS (real-time operating system), 64, 191

Run to completion, 21, 63, 191

■ S

Sampling

- hysteresis, 124
- jitter, 123
- period, 115

Satellite classes

- Satellite, 100–101
- Station, 100

Track

- association class, 99

Scale

- binary search, 109, 193
- hash table, 152, 194

Schematic, 3

Scall (Starr's concise relational action language)

- assignment operators, 257–258

- attribute references, 257

- class method, 263

- data types, 256, 257

- error handling, 260–261

- event to assigner, 259, 262–263

- event to me, 259, 262

- external entity

- asynchronous signal, 261–262

- synchronous operation, 262

- instance selection

- with criteria, 258–259

- with no criteria, 258

- link and unlink, 260

- navigation, 259

- online reference, 263

- signaling, 259
- subclass migration, 261
- symbol names, 255
- values
 - Boolean, 257
 - enumerated, 257
 - literals not supported, 257
- variable
 - instance set, 255
 - relation, 256
 - scalar, 256
 - system, 256
 - variable types, 255
- Self C variable
 - translation of `me` keyword, 24, 53, 262
- Sequence diagram
 - signal converter assignment, 116–122
- Shlaer-Mellor Action Language, 227
- Shlaer-Mellor methodology, 225–226
- Signal. *See* Event
- Signal an event action (->), 59, 75, 76, 147, 150, 277
- Simulation
 - desktop model execution, 74
- SIO classes
 - Continuous Input Point, 124, 126, 130, 132, 140, 141, 143–145, 151, 154
 - Conversion, 114–120
 - Conversion Group, 115–122, 126, 150, 151, 153
 - I/O Point, 112, 126, 149, 222, 223
 - Point Threshold, 124, 127, 130, 142–145, 154, 156
 - Range Limitation, 124, 125, 127, 143–145, 154–156
 - Signal Converter, 115–122, 126
- SIO (Signal I/O domain)
 - sensing and controlling signals, 111
 - sensors and actuators, 10, 111
 - side effects, 111
- SOC (system on a chip), 113, 189, 191
- Software architecture, 2, 59, 81
- Specialization
 - project teams, 234
- Specification class
 - modeling pattern, 83
 - subclass determination
 - Torpedo Spec, 166
- Spreadsheets
 - for bridge tables, 134–137, 139–141, 144–151, 154, 160
 - for initial population, 35, 46, 146, 221
- State
 - event signature rule, 20, 139
 - model script, 29, 30, 38, 39, 183
 - naming, 35, 38, 47, 57
- State activity
 - as C function, 52, 54, 59, 146, 147, 149, 160, 268
- preprocessor macros and hand code, 30, 38, 40, 55, 57, 107, 265, 275–276
- State coverage
 - spanning tree, 74
- State machine
 - diagram, 14
 - for each instance, 14, 67, 120, 253, 262
- State model
 - action, 15, 21, 27, 28, 48, 167, 168
 - activity, 76, 81, 171, 248, 251
 - behavior, 15, 19, 28, 251, 273, 275
 - definition, 38, 54, 55, 273, 275–276
 - entry activity, 22
 - event, 38, 40, 73, 92, 102, 148, 161, 162, 166, 253
 - event parameter, 21
 - execution, 103, 162
 - generated code, 10, 31, 32
 - lifecycle, 19, 73, 89, 167, 250
 - model script, 29, 30
 - polymorphism, 225
 - signal, 28, 38, 92
 - state, 19–20, 21, 35, 101, 188, 275
 - table, 27, 232
 - transition, 37, 49–50, 75, 233, 250, 275
 - translation, 36, 39, 48–52, 70
- State model diagram, 73, 76, 232, 250
 - Moore and Mealy, 21, 49
- State transition table
 - transitory state, 94–97
 - wait state, 94–96
- ST/MX (single threaded model execution)
 - C library, 61, 75, 109
 - instance management, 222, 281
 - memory usage, 191
- Subclass
 - state model, 161, 167, 169, 183
 - translation, 170–174
- Subclass migration
 - preprocessor macro, 57, 147, 285–287
- Subject matter experts, 217–218
- Superclass
 - model script, 183
- Synchronization
 - activities, 21
 - concurrency, 22
 - events, 21, 67, 191, 250
 - platform independent, 4, 251, 252
- Sync queue
 - main loop, 64, 191
- System executable, 215
- System on a chip (SOC). *See* SOC (system on a chip)
- System partitioning
 - functional *vs.* subject matter, 81
 - platform independent *vs.* specific, 1, 214, 224, 240, 248

T

Target platform

hardware

Arm Cortex-M3, 74, 189, 191

Giant Gecko, 189

implementation language, 1, 2, 8, 29, 30, 32, 59, 107, 189, 195, 210, 211, 231, 234

micro-controller, 4, 8, 9, 35, 60, 63, 65, 69, 74–76, 113, 185, 189, 191–192, 195, 197, 206, 210, 222, 231

programming language, 1, 4, 5, 7, 8, 19, 27, 31, 59, 60, 107, 161, 195, 215, 225–229, 234–235

reactive applications, 8, 65

software

Silicon Labs Simplicity Studio, 190

technology, 8, 132, 216, 234

Textual *vs.* graphical layout, 219

Third party code, 214, 216, 240

Threshold limit, 123–124, 126–127, 144–145, 154

Time

platform specific, 54

POSIX, 45, 54, 74–75, 195

Timeout

modeling, 25, 27, 90

Timer. *See also* Event, delayed

platform resource, 69, 191

Tools

complexity, 235

misuse of "real-time" term, 235

monolithic *vs.* tool chains, 235–236

Torpedo classes

Active Launch Torpedo, 166, 168, 171

Fired Torpedo, 166–168, 180

Loaded Torpedo

model script, 166, 167, 169–174, 177, 179, 180

Passive Launch Torpedo, 166–168, 171

storage rack, 165–167

Stored Torpedo

model script, 166, 167, 171–172

Torpedo

pycca, 169–174, 179

torpedo design, 165

Torpedo Spec, 166, 174

torpedo tube, 165, 167, 171

Tube, 165–169

Torpedo class method

arm, 168

disarm, 165

Torpedo launch example

polymorphism, 169–182

Torpedo state model

Active Torpedo, 167, 169

Fired Torpedo, 166–168, 180

Passive Launch Torpedo, 166–168, 171

Stored Torpedo, 166, 167, 171, 172

Traceability, 231

Trace execution, 73–76, 122, 221, 266

Transaction

data model integrity, 104

Transducer, 111, 113, 126

Transition

can't happen (CH), 36, 51, 71, 94, 250

ignore (IG), 36, 71, 76, 90, 102, 167, 169,

250, 275

matrix, 50

as array, 70, 73

normal, 101, 176

table, 50, 59, 61, 70, 76, 176

Translation

actions, 27, 28, 38, 48, 59, 61, 66, 97, 102, 107, 221, 223, 226

basic workflow, 9

class model, 37, 39–48, 233

data types, 40–42

decisions, 29–38, 219, 233

environment, 7–9, 219

human intervention, 233, 236

processing, 1, 10, 59, 221, 231

pycca workflow, 11, 232–236

three facets, 30, 39, 97

working backwards, 235

Translation workflow

pycca, 11, 232–236

U

UML

lingua franca, 9

standards, 9–10, 219, 225, 227, 257

Unicorns, 1

Unlink action (!&), 24, 260

V

Value threshold, 123–127

Variable initializers, 35

Vehicle classes

Box Truck, 163

Bus, 162

Car, 162

Semi Truck, 163

Truck, 162

Vehicle, 10, 78, 82, 162

Vendor tools

target technology, 7–8

W

Watchdog timer, 76

Web application, 7–8

Wind turbine, 78–79

Workflow

pycca, 8, 29, 39, 61, 77, 111, 129, 169, 185, 213, 265
reference, 213–215, 232–236

X, Y, Z**xUML**

executability, 225, 226, 255
language, 219, 224, 226, 227, 236, 240, 255

lean notation, 226

mathematical foundation, 9, 226, 229

notation, 5–7, 13, 225, 237

platform independent execution

semantics, 5, 237

semantics, 5, 9, 13, 225, 237, 255

synchronization, 4, 39, 252, 290

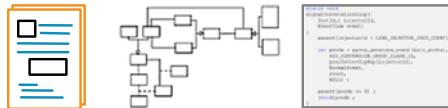
and UML, 1, 9, 17, 20–21, 22, 161, 225

why use it, 6, 22

Online Resources and Author Contact Info

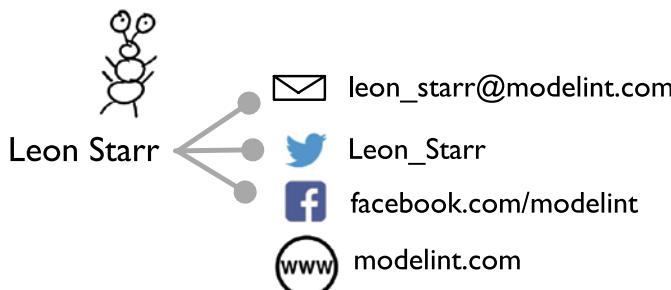
The Book Site

modelstocode.com



Complete models and source code presented in this book are available to freely download.

Author Contacts



Andrew Mangogna ——●———✉ andrew@modelrealization.com



Stephen J. Mellor



Other Sites

executableuml.org

xUML books, websites,
examples, lessons, reference
materials and more

modelint.com

We deliver expert training, requirements
modeling, pre-built and custom built domain
models and model translation solutions for
model driven software projects.