Urml: A textual toolkit for teaching model-driven development for reactive systems

by

Keith Yip

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
July 2014

Abstract

Modelling is widely used in traditional engineering. Models also serve as the foundation of theoretical computer science—from computational models to formal languages. However, even though software designers use formal systems of software, a dominant modelling methodology—model-driven development (MDD)—has not yet penetrated into the industry. This can be attributed to the practitioners' flawed understanding of the benefits of MDD, the practitioners' programming-centric mindset, and current complex and expensive tools. Thus, this project aims at providing *education* and *simple tools* for the public to appreciate MDD. The product of this project is *Urml*, which is an educational toolkit for utilizing MDD to develop models for reactive systems. There are four steps in using this toolkit: first, one specifies the metamodel that serves as the language to build models; secondly, one builds a model for reactive systems using the language that one has built; thirdly, one analyzes the model for correctness and consistency; and fourthly, one executes the model for testing through a custom-made interpreter. These four steps—(1) language design, (2) model building, (3) model analysis, and (4) model execution—serve as the pillar of MDD and thus provide students a comprehensive overview of what MDD entails. This toolkit is complemented with identifications of extensions and customizations suitable as student projects in a graduate course in MDD.

Acknowledgments

I would like to thank Dr. Dingel for his guidance throughout the making of this thesis.

Contents

List of Figures

Chapter 1

Introduction

In traditional engineering disciplines, models and modelling practices have been widely used since the earliest days. The benefits of modelling cannot be understated. Modelling provides engineers a scaffold to reason and make prediction about their problems and solutions. Models are also an effective tool to communicate ideas among different stakeholders. Models can serve as as blueprints for realizing the final product.

Perhaps the key benefit of modelling in traditional engineering is reliability. First, we can make sure that our *designs* are reliable, in that our solution will fulfill its key requirements within specified cost, resource, and time constraints. One can assess the feasibility and cost of one's design before investments are made for its realization. Secondly, throughout from specification to realization, we can make sure that our *design intent* is transmitted reliably. If a solution is certified as valid through its analysis, one can be sure that the realization is an accurate rendering of that design [20].

If modelling is heavily used in traditional engineering, how about in software design? Modelling was widely used too during the days when computers and programming languages were being invented. For instance, the use of control flow graphs and railway syntax diagrams can be a precursor of code, and computational models such as finite-state machines and the Turing machine can be considered as precursors of computers.

1.1 Model-driven development and its related methodologies

During the previous decade, there is a number of developed approaches that are grouped under the category called model-driven development. **Model-driven development** (MDD) is a development methodology in which *models*—instead of code—serve as the primary artifact of the development process. Usually, code and documentation are (semi-)automatically generated from the specified models. In this sense, then, models are not merely *descriptive* (serve mainly as part of documentation), but are *prescriptive* (serve as *part* of the programming process).

There are also various MDD-related approaches that have been developed by the research and industrial community. **Model-driven architecture** (MDA) is a particular vision of MDD proposed by the Object Management Group (OMG); this vision relies primarily on OMG standards. MDA is a subset of MDD.

Beyond the development activities of MDD, **model-driven engineering** (MDE) also includes other model-based tasks of a complete software engineering methodology. Such model-based tasks can be a model-based *evolution* of the target system and the model-driven reverse engineering of a legacy system. In this case, MDD is a subset of MDE.

There is also a softer version of MDE, called **model-based engineering**, which is a process in which software models serve as an *important role* though not necessarily the *key artifacts* of development. For instance, during analysis, designers might specify the domain models of the system; however, these models are directly handed out to programmers as blueprints to *manually* write the code. Compare this to MDD, which involves *explicit definition* of some platform-specific model to perform *automatic* code-generation. In this case, the development process is not *driven* by models but somehow *based* on models. MDE is a subset of MBE.

The family of the above approaches can be summarized by the hierarchy, MDA $\subseteq$ MDD $\subseteq$ MDE $\subseteq$ MBE [5]. Despite the numerous approaches above, the focus of this paper will be on MDD.

1.2 The use of MDD today

One might ask how well MDD has been adopted in the industry. Stephan Mellor, a pioneer in MDD research, was asked when MDD would become mainstream practice in industry. Mellor responded, “Three years,” with the caveat that he had been giving the same answer since 1985! Indeed, compared to other paradigms of programming such as functional programming, the industry has been quite behind in adopting MDD approaches.

Why is MDD still not a prevalent practice today? The first has to do with our place in history. Consider the technology development cycle in Figure 1.1. When the technology was first introduced, it reached a “peak of inflated expectations”; this is mostly due to the popularization of UML, which was sold by vendors as the silver bullet for software development. This initial hype was irrational and led to disappointment; hence, UML and its related methodologies such as MDD was falling into the “trough of disillusionment.” This phase occurs during the previous decade, when negative perceptions about MDD were made. The opinion recovered slightly afterwards and we are currently at the beginning of the “slope of enlightenment,” when viable applications of MDD are being discovered, such that companies can bring opportunities where MDD can make a difference [5].

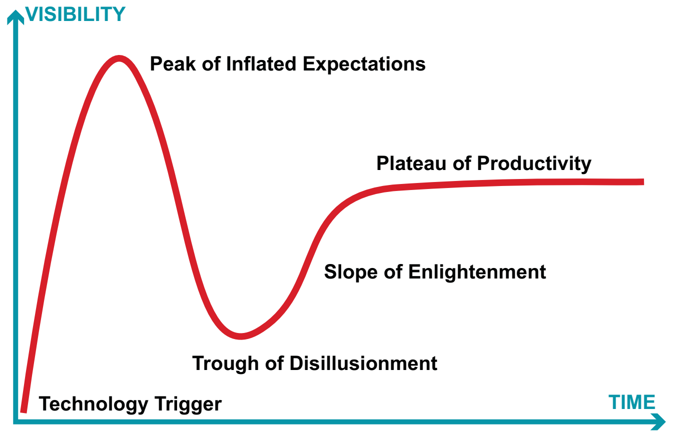


Figure 1.1: The technology development cycle: The “technology trigger” arises when UML was introduced. The “peak of inflated expectations” corresponds to the hype of UML, where UML and its related technologies such as MDD were regarded as *the* silver bullet of software development. The rapid decrease after such hype is represented by the “trough of disillusionment,” which is the time when many misconceptions about MDD are made. Nevertheless, a promising scenario begins during the “slope of enlightenment,” when more realistic applications of MDD are being pursued. The “plateau of productivity” corresponds to the final stage of technology, which is the time when the technology has earned its support from the industry [5].

The “slope of enlightenment” comes with obstacles, however. Programmers must be able to overcome a few major impediments before MDD can become mature. Some of the impediments are summarized below:

Inadequate or flawed understanding of the benefits of MDD

Many programmers perceive MDD as a method in which expensive computer-aided design tools are used to draw “pretty” UML diagrams, which have little relation to the actual implementation. They see models the same way as they do in documentation. Modelling—like documentation—is something they have to do for the sake of following development guidelines and rules, and something that merely distracts programmers from what actually matters: programming. This obsessive attitude towards programming leads to the next point.

Programming-centric mindset

Building software tends to have a highly seductive nature. The cycle of software development can be very short; from conception to realization, it can take only minutes. This is in contrast to the same development cycle for traditional disciplines, which can take months or even years. Such a short development cycle attracts creative individuals because it amplifies creativity. However, at the same time, the same programmers would be attracted to build software for the sake of programming itself, without concentrating on the *specific domain* for which they build a solution. Programmers will invest time practising their skills related to a particular technology or programming style, and will be disinclined to switch, even if the new methods may be better suited for the problem. This can be seen in C++ programmers who might be encouraged to try more recent languages such as Java or C#, but are unlikely to switch to a language of different paradigm, such as a modelling language.

Complex and expensive MDD tools

Commercial MDD tools can be quite expensive. IBM's Rational Software Architect, for instance, costs several ten thousands of dollars. Using it, particularly with massive models that are not uncommon in today's software development, can be quite difficult as well. On the other hand, open-source tools are usually free. However, many of them are not robust or capable enough to cope with technical challenges in complex industrial projects. Alternatively, there are a few industrial funding efforts to develop open-source tools; an example of such efforts is the Eclipse Foundation, which is established by consortia of industrial working groups. Nevertheless, the initial investment required for develop such open-source tools may *exceed* the amount needed to purchase existing MDD tools, not to mention the legal liabilities that arise from intellectual property issues.

What can be done to circumvent or mitigate these hurdles? **Education** might be the best answer to this, so that more open-minded generations of software design graduates are able to select whichever technology or programming technique to *apply* to whatever problem at hand—whether the approach is model-based or based on third-generation general programming languages. One way to enhance such education is to **build simple tools**, such that students are encouraged to explore the concepts and ideas underlying MDD and its toolings.

These two needs—(1) education, and (2) simple tools—provide the motivation to develop the language Urml[1], a toolkit for teaching the use of MDD in reactive systems.

[1]Urml is the abbreviation of the name "*U*ML–*R*T *M*ini-*L*anguage." It refers to the character, Urmel, from *Impy's Island*, a 2006 German-made computer animation. This animation is based on the children's novel by Max Kruse, *Urmel from the Ice Age*.

1.3 Objectives

There are five parts in Urml, with each part corresponding to the major five key components in MDD:

1. **Language Design.** We first design a domain-specific language (DSL) for our potential models we wish to build. For our project, we will only consider *textual* languages.

2. **Model Building.** Then, we build a model using the language that we have designed.

3. **Model Execution.** After that, we execute our model to test and validate it, using a compiler or an interpreter.

4. **Model Analysis.** Next, we analyze our model for correctness and safety.

5. **Code Generation/Interpretation.** Finally, if everything is good, we can generate code that conforms to our model or interpret that model directly.

A typical workflow involving the five steps in Urml is shown in Figure 1.2. First, a BNF grammar for the Urml language is specified in Xtext, which will generate a parser. Second, the user specifies an Urml model in Urml language, which is to be parsed by the parser to transform into an abstract syntax tree (AST). Third, the AST can be interpreted, which displays the output. Fourth, the AST can be analyzed, which outputs the result. Fifth, the AST can be transformed by the code generator into code.

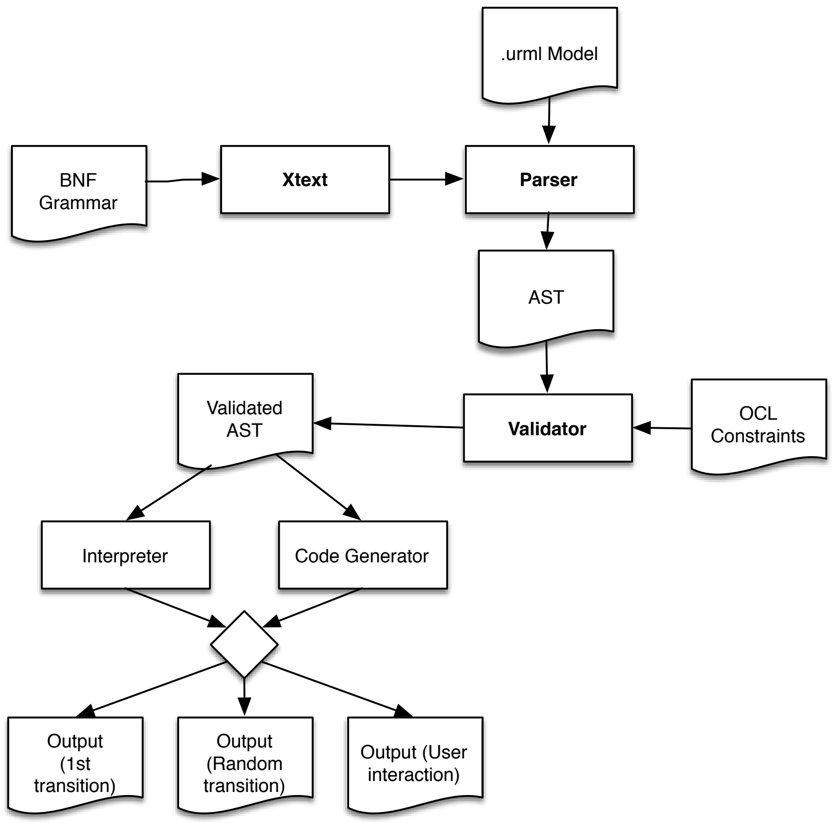


Figure 1.2: A typical workflow in Urml

1.4 Previous Work

The languages Urml is based on are ROOM [21] and UML-RT [19]. These languages are implemented in the MDD tools ObjecTime Developer[2], Rational Rose RealTime[3], and Rational Software Architect Real-Time Edition (RSA-RTE)[4], all of which are the industrial standard for model-driven development. Nevertheless, in these languages, the meta-model is fixed, and hence their DSLs are already defined as profiles. Further,

[2]The documentation for ObjecTime Developer can be found at `ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/objectime.jsp`

[3]The documentation for Rose RealTime can be found at `ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rosert.html`

[4]The documentation for RSA-RTE can be found at `http://pic.dhe.ibm.com/infocenter/rsarthlp/v9/index.jsp`

the user interface is difficult and cumbersome to use. Moreover, their purpose is for industrial use to generate code and not for educational use to learn about MDD. Also, since the action code is written in the target language (i.e., C++ or Java), the semantics of the code can be as complex as the target language, which makes it difficult to analyze for various properties such as correctness. These tools are also expensive and not readily available for students around the world.

Another general tool that has similar targets as Urml is the Eclipse Modelling Framework (EMF) [23]. Unlike ROOM/UML-RT, it is free and under open-source. The user can also edit the meta-model using the Ecore language. However, when the meta-model becomes large and complex, such as that of UML-RT, the user cannot edit the meta-model straight from the editor graphically using tree diagrams and class diagram. Additionally, it can be difficult to use the generated textual editor as they do not have full IDE support.

More recently, a tool that is based on ROOM called eTrice [17] has been developed under the Eclipse Foundation. The language is free, and uses Xtext, which allows for full IDE support for the ROOM models it edits. However, like RSA-RTE, it is for industrial use, and has features that are too advanced to use (for instance, the use of layering). Also, the Xtext meta-model in eTrice is already very advanced, because eTrice is intended to implement ROOM faithfully; thus, it is not ready for the user to directly customize the meta-model, thereby limiting the presentation of the overall MDD process for educational purposes.

Umple [8] is an educational toolkit which is used for teaching MDD through the use of textual class diagrams and state machines; it has been widely used for teaching modelling in University of Ottawa and eastern Ontario. What distinguishes Urml from Umple is that Umple is based on object-oriented programming methodology, while Urml is based on ROOM/UML-RT, which is a methodology for building *real-time and reactive* systems using the terminology similar to that is used in object-oriented methodology. Moreover, like eTrice, the meta-model (i.e., the language) being used in Umple is already set, and thus a user in Umple is not able to modify the meta-model directly. Meanwhile, Urml is designed to teach the *entire* modelling process: (1) meta-modelling through language design, (2) model building, (3) model validation, and (4) model execution. Both eTrice and Umple are designed to target only to (2) model building.

1.5 Contributions

Like eTrice, Urml is also developed in Xtext, and thus has full IDE support. Also, the Xtext meta-model is designed to be minimal and hence can be modified without difficulty. Based on the factors above, Urml makes it easy to teach the MDD workflow that is described in Figure 1.2.

1.6 Organization of the Thesis

The necessary background is provided in Chapter 2, where the concepts of meta-models, DSLs, and their supported tools are discussed. Then, the Urml language is described and specified in Chapter 3. Once the language is described, one needs to validate that the textual model is correct; this analysis and validation stage is described in Chapter 4. The execution of the textual models through an interpreter is illustrated in Chapter 5. Finally, seven examples that use Urml are provided in Chapter 6. The thesis closes with future work along with a conclusion in Chapter 7.

Chapter 2

Background

We first want to design a language for the models that we want to build. Since the language has a specific purpose, the language will also be a domain-specific language (DSL). A **domain-specific language** is a computer programming language of limited expressiveness created specifically to solve problems in a particular problem domain. It is not intended to solve problems outside of that domain. This is in contrast to general-purpose languages (such as C++ programming language and UML modelling language), which are created to solve problems in many domains [11].

There are two kinds of DSLs: external DSLs and internal DSLs. An **internal DSL** is represented *within* the syntax of a general-purpose language; it is a "stylized" use of the general-purpose language for a domain-specific purpose. As an example, Ruby on Rails uses the syntax of the Ruby language to build an internal DSL. An **external DSL** is represented *separately* from the host language; it can have a custom syntax, although it might use the syntax of another common language, such as XML. For instance, regular expressions can be considered as an external DSL because they do not have a host language to base on. Nevertheless, as DSLs are becoming more popular, there are workbenches known as **DSL toolkits** arising from the market.

These DSL toolkits are specialized IDEs that are used for defining and building DSLs, in particular, external DSLs. It makes it easy to define parsers and custom editing environments for the DSL [11].

Because there are many approaches in building a DSL, and DSL toolkits are heavily dependent on their development methods, it is better to discuss such approaches first.

2.1 Different Ways to Develop DSLs

A DSL has two “layers” of syntax: (1) the *concrete* syntax, which is the visual notation that represents the objects the language, and (2) the *abstract* syntax, which is the data structure that contains the information about the objects of the language.

2.1.1 Concrete syntax and abstract syntax

The **concrete syntax** (CS) of a DSL is the representation in which the user interacts with the language. This representation can be textual, graphical, tabular, etc. The **abstract syntax** (AS) of a language is the data structure that holds the core information in a program without any notational details that are contained in the concrete syntax; for instance, white-spaces, keywords, and comments are not included in the abstract syntax. Abstract syntax typically has a tree-like structure and thus sometimes people refer it as **abstract syntax tree**, or AST.

2.1.2 CS-first and AS-first development

Since a language can have *concrete* syntax and *abstract* syntax, there are also two ways to develop such a language: *concrete syntax-first* and *abstract syntax-first* development.

In **CS-first development**, the CS is defined by the user first, and then the AS is derived from the CS definition, either automatically or using hints from the CS definition. This is the default in Xtext, where the user first specifies the Xtext grammar and then Xtext automatically derives the metamodel from it. While this is more convenient, the resulting metamodel may not be necessarily clean and require manual adjustment.

In **AS-first development**, the metamodel—or the AS—is defined first, and then we define the CS manually by referring and conforming to the AS. Because the CS has to conform to the AS, there are typically more constraints that are externally imposed by the AS, thus making the construction of the CS difficult.

2.1.3 Parser-based and projectional approach

Once the language is defined, the language has to be used in some way so that programs can be made. There are also two ways in which AS and CS can relate to each other, as the language is used to create programs: *parser-based* approach and *projectional* approach.

The typical programming cycle is parser-based. In this **parser-based approach**, the user enters character sequences to represent programs using an editor. Then, the user launches a parser, which checks the program for syntactic correctness; the parser then transforms the character sequence into an abstract-syntax tree (AST). Finally, the AST contains all the semantic information that is expressed by the program.

The **projectional approach** is the other way around. The user edits the program directly by modifying the AST. The projection engine then creates some representation of the AST which the user interacts. To define such a projection engine, instead of defining a grammar, the user must define:

1. **Projection rules** that map language concepts (in the abstract syntax) to a notation (in the concrete syntax); and

2. **Event handlers** that modify the AST based on user's editing gestures.

Although the projectional approach might be considered somewhat unusual when compared to traditional programming paradigms, it is well-known for defining graphical editors. Consider the case when a user creates a class in a UML diagram. This is not the case when the user draws a rectangle (i.e. directly modifies the concrete syntax) and then some "pixel parser" parses the rectangle and infers the rectangle as a class in the AST. Instead, an instance of a class (i.e. directly modifies the abstract syntax) is made when the user drags the rectangle from the palette into the canvas. The projection engine then renders the diagram, based on the projection rule that a class is represented by a rectangle. This case thus follows the **model–view–controller** (MVC) pattern, in which the "model" (the abstract syntax) is the class instance, the "view" (the concrete syntax) is the rectangle, and the "controller" (the projection rule) maps the class instance to the rectangle diagram.

To summarize, a DSL has two layers of syntax: (1) concrete syntax and (2) abstract syntax. One can develop (1) CS-first or (2) AS-first. The two layers of syntax can relate to each other in two ways: (1) parsed-based or (2) projectional approach. These relations can thus be summarized in the matrix in Figure 2.1. With

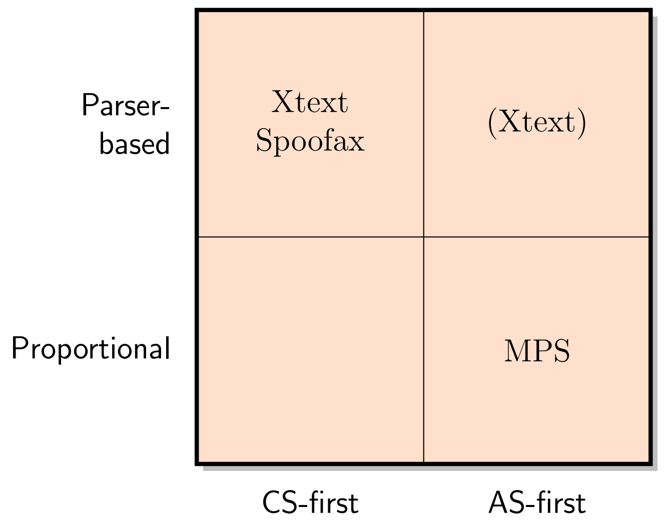


Figure 2.1: The three DSL toolkits introduced in this thesis are placed in the matrix. (1) Xtext is by default a CS-first parser-based toolkit; one can also develop a concrete syntax on top of a metamodel, and hence it is parenthesized in the AS-first column. (2) Spoofax is also a CS-first parser-based toolkit. (3) In contrast, MPS is a AS-first projectional editor.

various approaches of developing DSL discussed, it is an appropriate time to present DSL toolkits.

2.2 DSL Toolkits

As for the DSL workbench, we will use tools from Eclipse Modelling Project (EMP). The EMP is an ecosystem of tools and frameworks that are used for everything related modelling. The EMP is a huge network and is beyond the scope of this paper. However, in this paper, we will focus on the dominating DSL toolkit in EMP—Xtext.

Xtext is a framework for building textual DSL. It is now a major component of the Eclipse downloadable package *Eclipse IDE for Java and DSL Developers.* Xtext contains Xtend, a Java-like language that is used primarily for code generation. The internal data structure in Xtext is based on EMF/Ecore, the underlying meta-metamodel used to represent model data. Being developed as part of the now defunct openArchitectureWare project for around a decade, the tooling in Xtext has become very mature and has a huge user base. As the result, Xtext also comes with quite a few extensions that we will discuss further in this thesis.

Besides Eclipse and Xtext, there are a few popular DSL toolkits available on the market. One of them is JetBrains Meta-Programming System (**MPS**). MPS, in contrast to Xtext, is a projectional editor—there is no grammar and parser in MPS. Instead, the user changes the underlying abstract syntax tree (AST), and then MPS *infers* the concrete notation through various means (textual, symbolic, tabular, graphical) and through a range of other compositional features. Another DSL toolkit is **SDF/Stratego/Spoofax**. As implied from the name, the tool has three components. First, SDF is a formalism for defining parsers for context-free grammars. Secondly, Stratego is a term-rewriting system for AST transformation and code generation. Thirdly, Spoofax is an Eclipse IDE that is used for working with SDF and Stratego [26].

2.3 Xtext

The basic use case in Xtext is to compile a source grammar, written in an `.xtext` file. The source contains the grammar of the DSL, written in a BNF-like notation. Xtext then transforms that source grammar into the working code of an editor. This editor can be used for writing and updating the user's code written in that DSL.

2.3.1 Workflow

The typical workflow for developing a DSL in Xtext is fairly simple.

First, an invocation of Eclipse (with Xtext) workbench is launched and a new Xtext project is created.

Then, the grammar for the DSL is specified. The grammar is for the concrete syntax of the textual language, and is written in a grammar language that resembles the Backus-Naur Form (BNF). The grammar file has the extension `xtext`. This grammar language provides operations such as "`?`" (zero-to-one-occurrence), "`+`" (one-or-more times repetition), and "`*`" (zero-or-more times repetition). Other than these, Xtext adds many features to the grammar language to annotate extra information with respect to the abstract syntax; one of such features is cross-referencing.

After the grammar has been specified—or at least, the first iterative development cycle is done—the artefacts of the DSL can be generated. These artefacts—known as **fragments** in Xtext's parlance—of the language are generated through the Modelling Workflow Engine (MWE2). All the fragments are defined in the `GenerateUrml.mwe2` file, with the interesting ones as follows [12]:

1. `XtextAntlrGeneratorFragment`, which converts the Xtext grammar source into an ANTLR3 grammar, and then runs the ANTLR3 generator in order to create an ANTLR3 parser.

2. `EcoreGeneratorFragment`, which saves the generated Ecore models in a persistent format and then creates EMF generators. It then launches the EMF generator to create the EMF classes for the generated Ecore models.

3. Other generator fragments that make up an Eclipse IDE, including scoping, formatter, labelling, outlining, quick-fix, content assist, JUnit testing, rename refactoring, a preference page for code template, and a compare view.

2.3.2 Generation Gap Pattern

Every time the grammar is modified, the generator must be run in order for the model to conform to the grammar. This creates a problem: When one customizes the generated code, subsequent generations might overwrite one's handwritten customizations; consequently, the generated code may sometimes not do exactly what is needed. While the generated code may be *correct* and *efficient*, it may not be as *functionally complete* and as *maintainable* as one wishes.

This problem is mitigated by the **Generation Gap Pattern** [24] (see Figure 2.2), which separates handwritten and generated code through inheritance. The framework generates an abstract base class that contains all generated code, and then generates empty subclass of such abstract base class in a separate file. These empty subclasses are then handwritten by the user. That way, the framework can always regenerate the abstract base class with any generated code, without overwriting the subclass file which contains handwritten code. In Xtext, the code that is generated is placed in the `src-gen` folder, and the contents of this folder should not be modified because it will eventually be overwritten by the generator. Instead, the code that the programmer can modify is placed in the `src` folder [3].

2.4 Grammar Specification with LL(k) Engines: Some Constraints

ANTLR3 [16]—the parser generator that Xtext uses—supports specific classes of grammar, and in this case, LL(k) and LL(*) grammars. The naming scheme of LL(k) grammars tells us the way how the parser scans the input:

1. the first L represents left-to-right scanning;

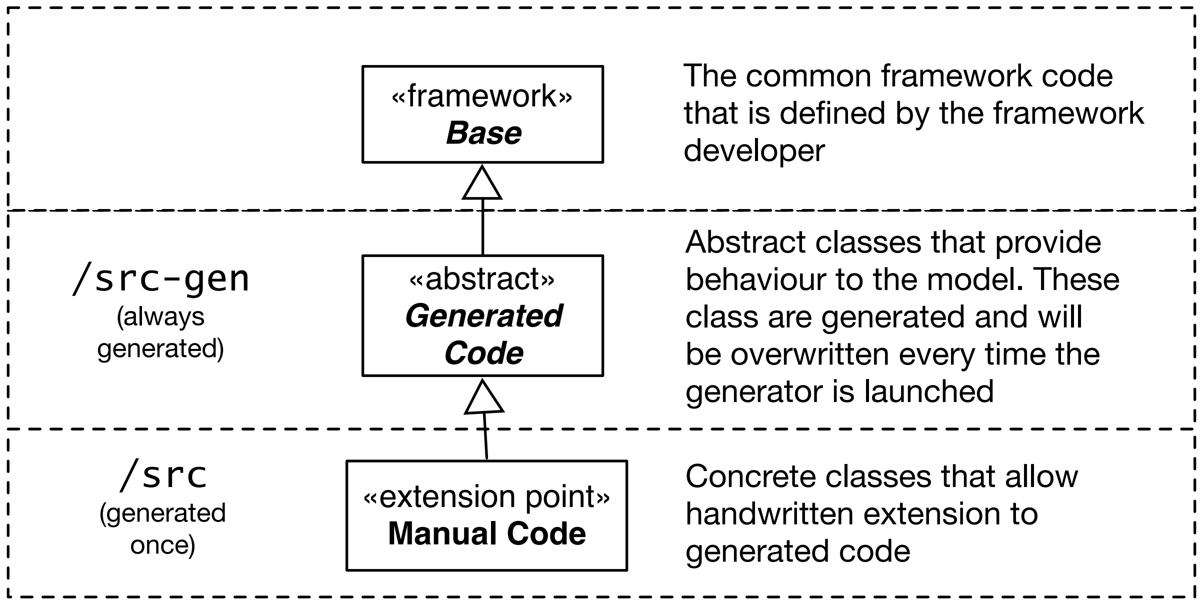

«framework»
Base
The common framework code that is defined by the framework developer
/src-gen
(always generated)
«abstract»
Generated Code
Abstract classes that provide behaviour to the model. These class are generated and will be overwritten every time the generator is launched
/src
(generated once)
«extension point»
Manual Code
Concrete classes that allow handwritten extension to generated code

Figure 2.2: Description of the generation gap pattern [1]. As an example, Xtext can be the framework that provides the common base code. As the user launches the generator, Xtext generates abstract classes in the `/src-gen` folder. Xtext also generates concrete classes that are subclass of those generated abstract class in the `/src` folder, which can be modified by hand by the user. This way, when the generator is launched again, the handwritten code from the concrete subclass will not be overwritten by the generated code, because the generated code is located in the abstract classes.

2. the second L stands for leftmost derivation of the input string; and

3. the k indicates the maximum number of tokens the parser will look ahead to decide what production rule it recognizes. In the special case—LL(*)—k is unbounded and the parser will look ahead arbitrarily many tokens to make decisions.

The tool supports only a particular subclass of BNF grammars. To specify its constraints, a few definitions are needed. Let A be a nonterminal; define:

- *first*(A) to be the set of all terminals that can appear at the beginning of any string derived from A.
- *follow*(A) to be the union of *first*(B) where B is a nonterminal contained in the production rule: $S \rightarrow \alpha AB$ for some terminal α.

For an LL(1) grammar, the following constraints must hold [18]:

1. If $X \rightarrow V_1 \mid V_2 \mid \ldots \mid V_n$, then *first*$(V_i)$ and *first*(V_j) are disjoint for all distinct i and j (i.e., $i \neq j$). Otherwise, a **first/first conflict** occurs.
2. If $Z \rightarrow X^*$, then *first*(X) and *follow*(Z) are disjoint. Otherwise, a **first/follow conflict** occurs.
3. No left-recursion occurs.

Because the tool supports only particular subclasses of context-free grammars, it is possible to write grammars that do not conform to these subclasses.

2.4.1 Left-recursion

For instance, consider the following grammar for expressions:

```
Exp ::= ID
      | STRING
      | Exp "." ID
```

Note that the leftmost symbol of one of the `Exp` definitions invokes `Exp`; this grammar is therefore left-recursive. Unfortunately, left-recursive grammars does not conform to LL grammars, and hence are not supported by ANTLR3.

One way to remove left-recursion is by **left-refactoring**; the grammar can be rewritten such that all recursive production rules consume *at least one token* before going into recursion or repetition.

Left-refactoring often (1) adds intermediate rules to the original left-recursive grammar, and (2) makes repetition explicit using `+` and `*` operators. Take the example above, replacing the recursion with repetition results to the following:

```
Exp ::= ID
      | STRING
      | Exp ("." ID)+
```

The grammar is still left-recursive; however, we can add an intermediate rule to remove the recursive call, as follows:

```
Exp ::= ID
      | STRING
      | ExpPart ("." ID)+


ExpPart ::= ID | STRING
```

2.4.2 First/first conflict

Unfortunately, we have overlapping rules: the `ID` and `STRING` terminal symbols match more than one production rule; that is, $\mathit{first}(\texttt{Exp}) \cap \mathit{first}(\texttt{ExpPart}) = \{\texttt{ID}, \texttt{STRING}\} \neq \emptyset$, resulting to a first/first conflict. We can remove the `ID` and `STRING` symbols from the `Exp` production rule and change the repetition from one-or-more repetition (`+`) to zero-or-more repetition (`*`), as follows:

```
Exp ::= ExpPart ("." ID)*
ExpPart ::= ID | STRING
```

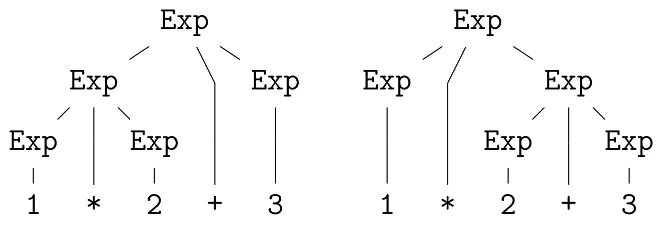
This grammar now conforms to the LL(1) subclass of grammars and can be parsed by ANTLR3 parsers [26]. The problem with “re-factoring” a left-recursion with left-refactoring is that left-refactored rules can be difficult to read; the intermediate rules have the tendency to hide their purpose.

2.4.3 Ambiguous grammars

Another problem with defining grammars for a parser is that grammars can be ambiguous. A well-formed sentence in the language can be constructed in *more than one way* from the production rules, resulting to multiple possible parse trees (and hence ASTs). For instance, consider the following grammar:

```
Exp ::= NUM
      | Exp "+" Exp
      | Exp "*" Exp
```

For this grammar, notice that the string `1 * 2 + 3` can be parsed into the following two parse trees:



Because LL parsers are deterministic, they can return only one possible output; thus, they cannot handle grammars with ambiguities. One can resolve ambiguities in the presence of associativity and precedence. To encode left-associativity and higher precedence for the `*` operator, we can write the LL grammar as follows:

```
Exp ::= Mult ("+" Mult)*

Mult ::= NUM ("*" NUM)*
```

Note how this grammar puts the `+` operator *higher* in the layer, giving it *lower* precedence; in other words, the `*` operator is more “sticky” to its terms because it is closer to the layer with the terminal symbol `NUM` [26].

Chapter 3

Description and Specification

The Urml language is quite simple; its grammar is shown in Appendix A on page 160. Its language concepts can be subdivided into three categories:

Types	**Structure**	**Behaviour**
Capsules	Attributes	Operations
Protocols	Ports (Log and Timer)	Signals
	Connectors	State Machines

The language also defines its own action code. Such action code can be used at (1) the body of operations, (2) the action code for transitions, and (3) the entry and exit code for states.

The expression part of the action code can also be used at (1) transition guards, and (2) default values for capsule attributes.

Note that although Urml is a textual language, it is possible to express an Urml model graphically. In this chapter, both the textual language and its graphical representation will be shown.

In the rest of the chapter, each concept of the language will be described in detail.

The Urml language is largely similar to ROOM [21] and UML-RT [19]

3.1 Capsules

Like classes in object-oriented languages, the fundamental component in an Urml model is the **capsule**.

The capsule encapsulates data (attributes), structure (capsule instances, ports, connectors) and behaviour (state machines, operations) into a single unit. This shell of encapsulation hides the capsule's implementation in both ways: It prevents the external environment from seeing the internal implementation of the capsule, but also prevents the capsule's internal components from having direct contact with the external environment. What makes such strong encapsulation useful? First, it *abstracts* out the potentially complex functionality of a capsule such that it can be seen as a simple single unit. Secondly, it *decouples* the capsule's environment from its internals, and thus shields any changes from one another, making the components more reusable. Thirdly, it *secures* the capsule's internal object to prevent undesirable access [21].

Each capsule instance represents as an **active object**. Thus, each instance of a capsule that has a state machine has its own flow of control. A capsule instance can be executed as a logical thread and can therefore operate concurrently with other active objects.

Like classes, a capsule can have **attributes** and **operations**; however, unlike classes, instances of a capsule cannot be passed like objects in object-oriented languages. This is done in order to keep a capsule within its own flow of control, hence avoiding potential concurrency problems that can arise in typical real-time systems.

While capsules can have attributes and operations, these contained elements have private accessibility and cannot be accessed or modified by other capsules. The only way to communicate between capsules is by sending **messages** to them. However, messages cannot be sent directly to instances of capsules. These messages must be sent through **ports**, which must be routed between capsules through **connectors**. The ports hence serve as the only external interface for the capsule.

A capsule may contain behaviour in the form of a **state machine**. The action code in this state machine may have access to attributes and operations in the capsule instance.

A capsule can be **composite**, in that a capsule instance can contain other instances of capsules inside it. This allows the programmer to encapsulate related functionalities of a capsule into sub-capsules, and hence reduce the complex structure into a number of simple components.

3.1.1 Code Sample

A capsule can be declared through the `capsule` keyword. To illustrate how a capsule can be defined, consider the following example:

```
root capsule BlinkingLight {
  attribute blinkDelayTime := 1000
  operation doSomething() {
    noop
  }
  timerPort timer
  logPort logger
  port externalPort : ExternalProtocol
  capsuleInstance controller : Controller
  capsuleInstance light : Light
  connector controller.toLight and light.toController
  stateMachine {
    final state ending {}
    transition init: initial -> ending {}
  }
}
```

The `BlinkingLight` has a `root` keyword preceding it. This indicates that the root capsule has the type `BlinkingLight`. Be aware that in a model, there can be only one capsule with the `root` keyword.

A capsule can contain attributes, operations, timer ports, log ports, message ports, other capsule instances, connectors and state machines. These are all the possible items that a capsule may contain.

As for the graphical notation, a capsule may have three kinds of diagrams to present its structure and behaviour: (1) the class diagram, (2) the structure diagram, and (3) the state diagram.

The class diagram is shown in Figure 3.1. The first compartment of the diagram is the capsule's stereotype and its name. The second compartment contains the attributes of the capsule; the sub-capsules are also included there. The third compartment contains the operations. The fourth compartment contains the ports (timer port, log port, and message port) the capsule has.

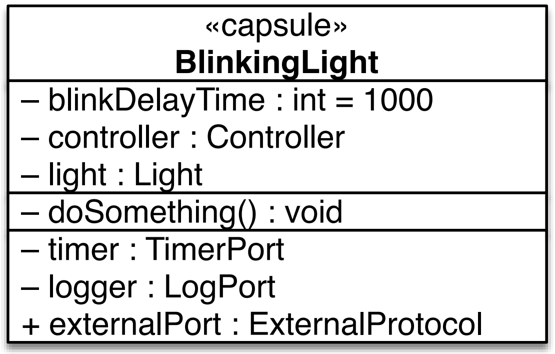


Figure 3.1: Class diagram of ExampleCapsule

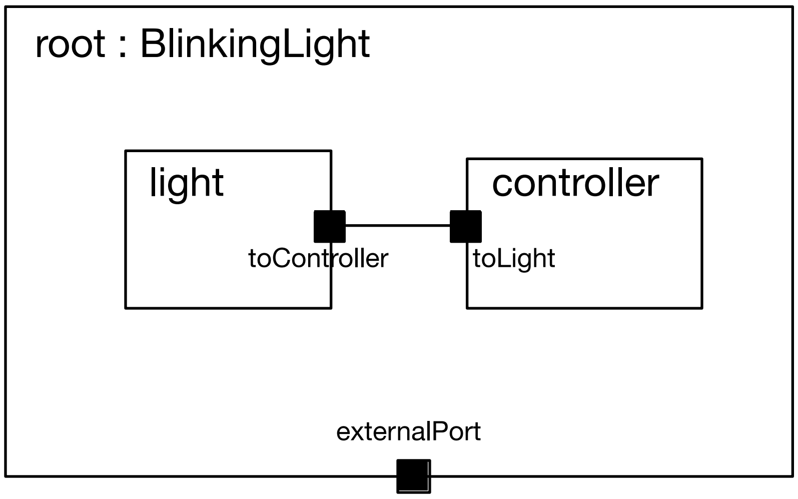


Figure 3.2: Structure diagram of ExampleCapsule

The structure diagram is shown in Figure 3.2. It shows the structure of the ExampleCapsule and its subcapsules. Note that the connector that connects between the two subcapsules is realized in the structure diagram.

The state diagram for the capsule is shown in 3.3.

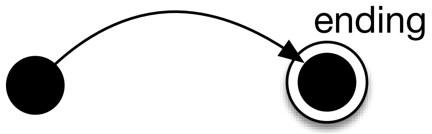


Figure 3.3: State diagram of ExampleCapsule

3.2 Protocols

A protocol defines what kind of messages to send into or out from a port. It is a pattern to which the set of message exchanged between two capsules must conform. Since there are two directions in which capsules can communicate, the protocol specification has **polarity**. So, the direction of messages are specified from the perspective of one of the two participating capsules: messages that are coming into a capsule are called **incoming signals**, and messages that getting out of the capsule are called **outgoing signals**[1].

3.2.1 Code Sample

To illustrate protocols as a concept, consider the following code snippet:

```
protocol ComputeResultProtocol {
  incoming { compute() }
  outgoing { result()  }
}
capsule Computer {
  port requestPort : ComputeResultProtocol
}
```

The code above can be represented in the class diagrams in Figure 3.4.

[1]Note that the terms “messages,” “signals,” and “events” are to be used interchangeably in this thesis.

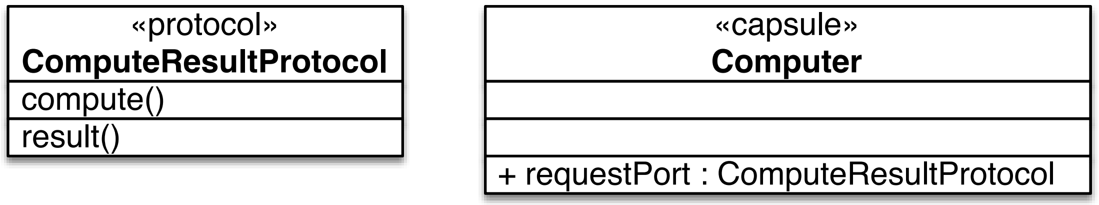


Figure 3.4: Class diagrams for the ComputeResultProtocol and Computer.

In the code snippet, the `Computer` contains an external port named `requestPort` that has the type `ComputeResultProtocol`. The `ComputeResultProtocol` has two types of signals: (1) incoming signals called `compute()` and (2) outgoing signals called `result()`. Since ports serve as the only interfaces for capsules, if one wishes to communicate with the `Computer`, one can only send signals to it through `compute()`; likewise, the only signals that the `Computer` can send to other capsules are those that are sent through `result()`.

3.2.2 Base and Conjugated Ports

Suppose that the port p_0 connects to another port p_1. The signals that p_0 receives are exactly those that can be sent by p_1, and likewise the signals that p_0 sends are exactly those that can be received by p_1. Originally, one can define two protocols with one for each side of the connection. This duplication can however be avoided by defining a *common* protocol (known as **binary protocol**) where both ports have the same type. One of the ports is set to be **conjugated**, in which the incoming and outgoing signals are swapped. So, messages specified as incoming signals are now outgoing and similarly messages specified as outgoing signals are now incoming. The port that has the default protocol is called the **base** port.

As a convention, components that take the role as *clients* receive the base side of the protocol, whereas components that take the role as *servers* receive the conjugated side of the protocol.

3.2.3 Code Sample

Consider the following example using the same ComputeResultProtocol defined above:

```
root Capsule Container {
  capsuleInstance requester : Requester
  capsuleInstance receiver : Receiver
  connector requester.requestPort and receiver.receivePort
}
capsule Requester {
  port requestPort : ComputeResultProtocol
}
capsule Receiver {
  port ~receivePort : ComputeResultProtocol
}
```

The structure diagram for the code above is depicted in Figure 3.5; the class diagram is shown in Figure 3.6. The type of the ports requestPort and receivePort is ComputeResultProtocol. However, receivePort is conjugated, as specified by the tilde symbol (~) before its name in the code. In the class diagram, the tilde symbol is specified after the protocol's name; in the structure diagram, base ports are shown as black small squares while conjugated ports are shown as white small squares.

3.3 Attributes

Attributes are slots where run-time data can be stored as values.

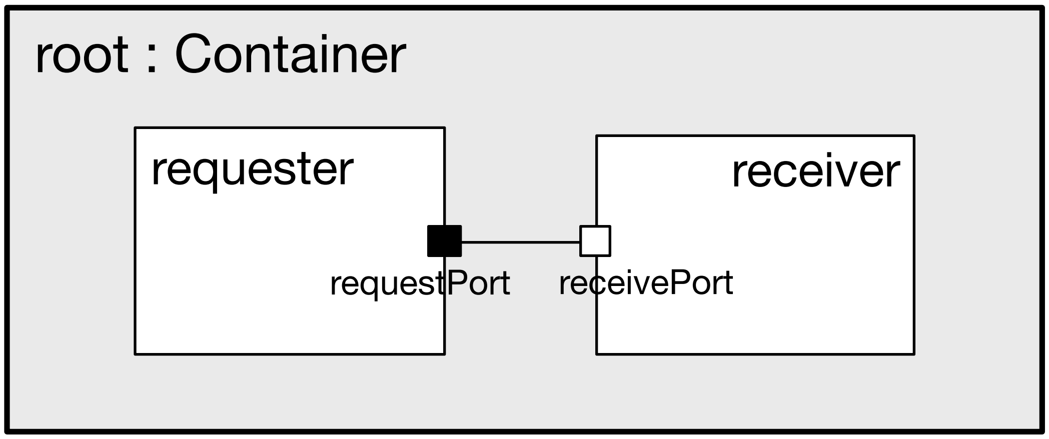


Figure 3.5: Structure diagram for Container

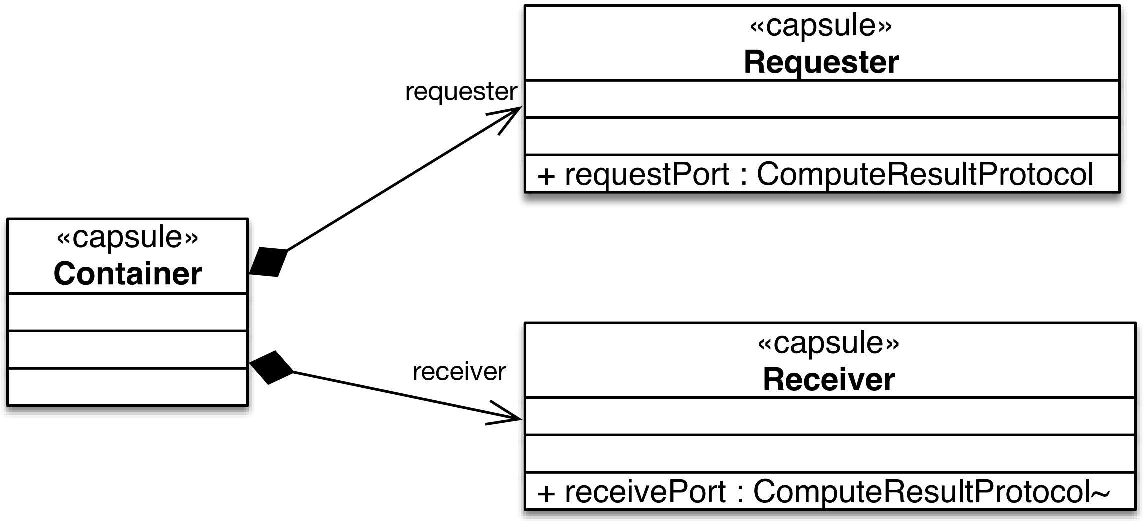

«capsule»
Requester
+ requestPort : ComputeResultProtocol
requester
«capsule»
Container
receiver
«capsule»
Receiver
+ receivePort : ComputeResultProtocol~

Figure 3.6: Class diagrams for Requester, Receiver, and Container

Attributes are local within the capsule, in that they cannot be accessed by other capsule instances; however, all the action code and expressions in the capsule instance have access to them.

The typing system for attributes is very simple; there are only two types of attributes for use in Urml: (1) boolean variables and (2) integers. The user does not need to declare the type of an attribute; its type is inferred by the evaluator.

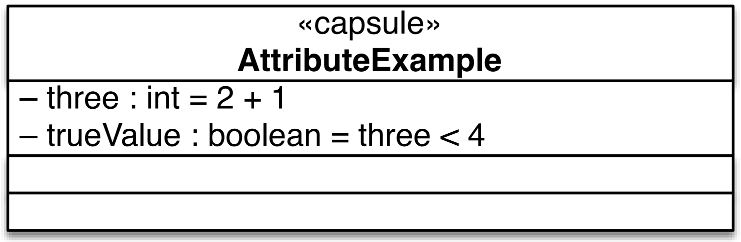


Figure 3.7: A class diagram representing the AttributeExample capsule

The user can define the default value for an attribute through an assignment symbol (:=). The expression followed by the assignment symbol will be evaluated at runtime and the evaluated result will be assigned to the attribute.

3.3.1 Code Sample

In textual form, attributes are declared as part of a capsule. Consider the following snippet, which is a capsule containing several attributes:

```
capsule AttributeExample {
   attribute three := 2 + 1
   attribute trueValue := three < 4
}
```

Note that after an attribute is defined, expressions from other places of the capsule can have access to the attribute.

In graphical form, attributes are shown in the class diagram of a capsule. Shown in Figure 3.7 is a class diagram representing the code snippet above. Note that attributes are located in the second compartment of the capsule's class diagram.

3.4 Ports

A port is an object that serves as a boundary for a capsule instance; it mediates the interaction of the capsule instance to the outside world. By forcing each capsule to communicate only through ports, one decouples the capsule's internal implementation from any knowledge about the external environment, which increases the reusability of the capsules.

Each port plays a specific role in the protocol, in a sense that each port must have a type that is a protocol. The port then implements the behaviour specified by that protocol role.

A port generally has one of two purposes: to relay signals (called **relay ports**) or to serve as the ending point for signals (called **end ports**).

3.4.1 Relay Ports

A port does not need to send or receive any signals; it may simply pass the signals through. This port is known as a **relay port**. A relay port is typically connected to one end of a sub-capsule, and on the other end usually to some other "peer" capsule in the outside world.

Relay ports generally do not require any intervention from state machines of capsules. Messages entering a relay ports are simply funnelled directly to a sub-capsule without interacting with other behavioural elements. Thus, relay ports are purely structural elements and are irrelevant when modelling the behaviour of the program.

3.4.2 End Ports

Ports that *do* send or receive signals are known as **end ports**, which are boundary objects of the state machines in the capsules.

End ports are essentially the source and sink of all signals between capsules. Consider the lifetime of a signal: A state machine first sends a signal at one of its end ports. The signal is then relayed through the connector, and possibly passed through one or more relay ports and connectors. The signal finally reaches another end point of another capsule, possibly triggering some transition in the other capsule's state machine to execute some behaviour potentially through the transition's action code.

To store the signal being received, each end port has a message queue that stores asynchronous messages that have been received but not consumed by the state machine.

As for their visibility, end ports can be **external** or **internal**. An external port has public visibility because it appears at the boundary of the capsule and is visible outside the capsule. An internal port has private visibility because it is completely inside the capsule as part of its internal implementation structure. Internal ports are useful in that the state machine inside the capsule can interact with the internal port in order to send messages to sub-capsules whose external ports are connected to that internal port.

3.4.3 Code Sample

In textual form, ports can be defined as part of a capsule definition:

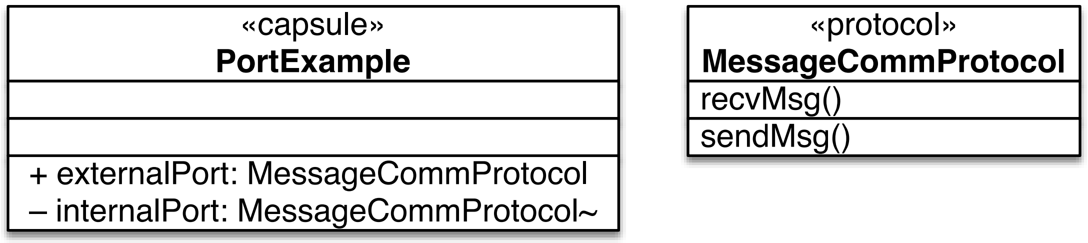

«capsule»
PortExample
+ externalPort: MessageCommProtocol
– internalPort: MessageCommProtocol~
«protocol»
MessageCommProtocol
recvMsg()
sendMsg()

Figure 3.8: Class diagrams for PortExample capsule and SomeProtocol

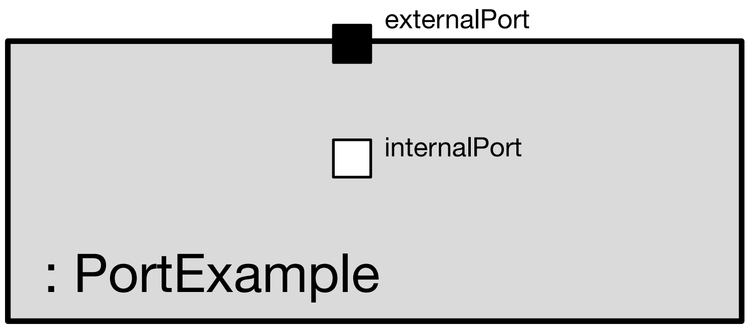


Figure 3.9: Structure diagram for PortExample capsule

```
capsule PortExample {
  port externalPort : MessageCommProtocol
  internal port ~internalPort : MessageCommProtocol
}
protocol MessageCommProtocol {
  incoming {  recvMsg()  }
  outgoing {  sendMsg()  }
}
```

If there is a tilde symbol (˜) before the name of the port, that port is a conjugated role of its protocol. Additionally, the ports are external by default; one can specify an internal port by using the "`internal`" keyword in front of the "`port`" keyword.

In graphical form, a port appears in the class diagram and the structure diagram. The class diagram and the structure diagram representing the code snippet above are shown in Figure 3.8 and Figure 3.9, respectively.

In the class diagram, the list of ports has an individual compartment that appears after the attribute and operation list compartments. External ports (relay and public end ports) have public visibility while internal ports (private end ports) have private visibility. For binary protocol, a suffix tilde (˜) symbol is used to identify conjugated role and the base role's name is implicit with no special annotation.

In the structure diagram, ports are represented by small black or white squares. Public ports are placed between the outside and the inside of the edges of the capsules—this represents that they are visible in public; private ports are placed inside the capsules and hence have private visibility. As for binary protocols, conjugates roles are identified by white squares, while base roles are identified by black squares.

3.5 Special Ports: Log Ports

Log ports provide functionalities that allow one to log text onto console output. This is mostly useful for debugging and tracing code.

In textual notation, the user can define a log port in a capsule definition. An example follows:

```
capsule LogPortExample {
  logPort logger
}
```

Although log ports appear in the ports compartment in class diagrams, they are usually omitted in structure diagrams.

In the action language, the programmer can print something out through a log port with the **log** command:

log [*log port name*] **with** [*string expression*]

where

- [*log port name*] is the name of the log port, and
- [*string expression*] is the string expression to be printed.

For instance, the following statement:

log *logger* **with** "$HelloWorld!$"

prints

```
root logging to logger with Hello World!
```

3.6 Special Ports: Timer Ports

A timer port serves as a timer which will send timeout signals after a specified time of duration. The timer port will send timeout signals *only once.*

In textual notation, the user can define a timer port in the capsule definition. An example follows:

```
capsule TimerPortExample {
  timerPort timer
}
```

Although timer ports appear in the ports compartment in class diagrams, they are usually omitted in structure diagrams.

In the action language, the programmer can start the timer with the following **inform** command:

inform [*timer port name*] **in** [*number of milliseconds*]

where

- [*timer port name*] is the name of the timer port, and
- the timer will timeout after a number of milliseconds as specified in [*number of milliseconds*].

For instance, the statement

inform *timer* **in** 5000

will set a timer to fire a timeout event after 5 seconds. Be advised again that the timeout message will fire only once. If the user wants to fire the timeout event repeatedly, the user must reset the timer through the **inform** command repeatedly.

3.7 Connectors

A connector is a communication channel that allows signals to be transmitted through a specified protocol from one port of a capsule instance to another. Connectors may interconnect only ports that are of the same or conjugated protocol.

3.8 Operations

An operation is a method or a function that can be performed by a capsule instance. An operation has the following syntax:

operation [*operation name*]([*parameter list*]) { [*body*] }

where

- [*operation name*] is the name of the operation, which must be unique;
- [*parameter list*] is the common-delimited list of parameters
  - The parameters provide a way to pass data into the operation body. Since the language only supports integers and boolean values, only passing-by-value mechanism is implemented. Note that the parentheses are *not optional*, even when there are no parameters.
- [*body*] is the body of action code that will be executed upon invocation

In the textual language, the user can define an operation as part of the capsule definition:

```
capsule OperationExample {
  operation helloWorld() {
    noop
  }
}
...
action {
  call helloWorld()
}
```

If used as a function, the operation body must have a **return** statement:

```
capsule FunctionExample {
  attribute hello := helloWorld()
  operation helloWorld() {
    return 0
  }
}
```

If an operation is invoked as a function call but the body of the operation does not have a **return** statement, an exception will be raised.

As for the graphical notation, in the capsule class diagram, operations will be listed in the operation compartment.

3.9 Messages

A message specifies communication sent form one capsule to another. This message can be sent or received by a capsule instance.

Every message belongs to a protocol, and a protocol may have two kinds of messages: incoming messages and outgoing messages.

In Urml, one can specify messages as part of the protocol definition:

```
protocol ExampleProtocol {
  incoming { receiveMessage(parameter) }
  outgoing { sendingMessage(parameter) }
}
```

This code translates to the class diagram in Figure 3.10. Like operations, one can pass data with messages by specifying parameters in it; this is done by specifying a list of parameters, delimited by commas and surrounded by parentheses. Note that the parentheses are not optional; they are required even when there are no parameters.

In action code (for transitions and entry/exit code for states), one can send a message through the **send** command. To do that, one must specify:

1. The port to which the message is sent. The type of the port must be a protocol that specifies that message to be sent as an outgoing message (or incoming message if that port is conjugated).
2. The name of the message itself; and
3. The data to be sent with the message, if there are parameters defined for the message.

The syntax for sending message is:

send [*port*].[*outgoing message*]([*parameter list*])

where

- [*port*] is the port of which the outgoing message goes through,
- [*outgoing message*] is the signal that is compatible to the port's protocol (i.e., outgoing signal for base port, incoming signal for conjugated port), and
- [*parameter list*] is the list of parameters that is specified in the protocol definition.

3.10 State Machines

A state machine specifies the behaviour of the capsule.

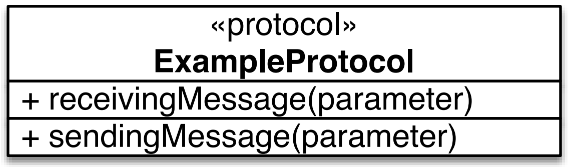


Figure 3.10: Class diagram for ExampleProtocol

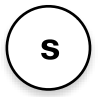


Figure 3.11: Graphical representation of a state

3.10.1 States and Transitions

A state machine can either be in a *state* or in a *transition* that is possibly triggered by a *message*. These three elements—state, transition, message—are the fundamental building blocks of a state machine.

In the language, **state** is specified by the keyword `state`, like as follows:

$$\textbf{state } s$$

where s is the name of the state; note that the name must be unique amongst all other states in the capsule. The graphical representation of a state is shown in Figure 3.11.

When a capsule is in a state, the capsule is ready to process incoming messages inside the machines. It looks at all the transitions that are outgoing from the current state. If there is an outgoing transition whose trigger matches an incoming message, the state machine begins to execute that transition if the state machine's condition satisfies the guard condition for that transition. If not, no change takes place. Until

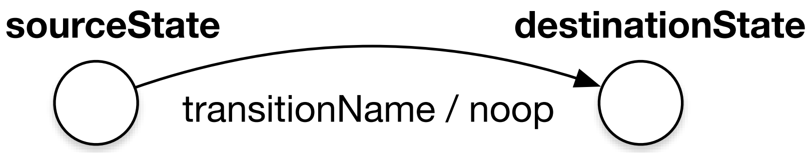


Figure 3.12: Graphical representation of a transition

the state machine receives another message at one of the end ports from the capsule, the state machine will remain at the current state.

During a **transition**, the state machine moves from one state (known as the *source state*) to another (known as the *destination state*). Each transition may have an optional block of **action code** attached to it, and the state machine will execute this block of action code. The environment in the action code is extremely limited, in that it can only have access to: (1) any attributes from the current capsule, (2) any operations from the current capsule, (3) local variables that can be declared and defined in the action code, and (4) data that is passed through the incoming trigger. Additionally, inside the action code, one can send messages through ports that are defined by the the current capsule in order to communicate with other capsules.

A transition is specified by the keyword `transition`, like as follows:

```
transition transitionName : sourceState -> destinationState {
  action { noop }
}
```

Note that the action code, which is distinguished by the `action` keyword, is optional and thus that block of action code is omitted. The graphical representation of the transition code is shown in Figure 3.12.

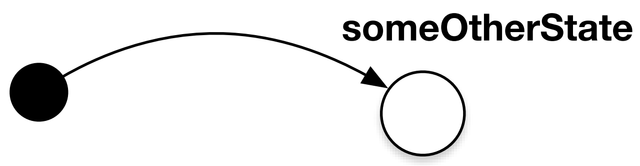


Figure 3.13: Graphical representation of an initial transition

3.10.2 Initial Transition

When a capsule instance starts, the state machine will look for an initial transition, as indicated by the `initial` keyword at the source state, like as follows:

```
transition t : initial -> someOtherState {}
```

The graphical representation is shown in Figure 3.13. Note that the initial transition is distinguished by the black dot at the source of the transition. Like all other transitions, the optional action code for the initial state is run to completion. After that, the state machine will enter the target state of that initial transition.

3.10.3 Triggers and Guards

Except the initial transition, a transition may be triggered by an incoming message (or an outgoing message if the port the message is coming through is conjugated). Such a trigger is specified by three elements: (1) the name of the incoming message that causes the transition to be triggered, (2) the end port from which the signal comes through, and (3) an optional guard condition. The first two is distinguished by the `triggeredBy` keyword, like as follows:

$\textbf{transition}\ \textit{outgoingTransition} : \textit{someOtherState} \rightarrow \textit{anotherState}\ \{$

$\quad \boxed{\textbf{triggeredBy}}\ [\textit{somePort}].[\textit{someIncomingMessage}]([\textit{parameterList}])$

$\}$

where

- [*somePort*] is the port at which the incoming message arrives,
- [*someIncomingMessage*] is the message that triggers the transition, and
- [*parameterList*] is the parameters of the incoming message.

If these two are not specified, the transition will be triggered by *any* valid incoming message.

The third element, the **guard condition**, is a boolean expression—written in action code—that must be evaluated true in order to trigger the transition. The guard expression is optional, and is assumed to be true if not specified. The guard condition is distinguished by the **`guard`** block, like as follows:

```
transition outgoingTransition : someOtherState -> anotherState {
  triggeredBy somePort.incomingMessage()
  guard { 3 < 4 }
}
```

The graphical representation of the code above is depicted in Figure 3.14.

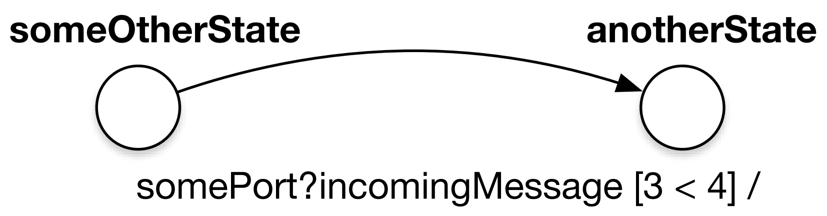


Figure 3.14: Graphical representation of an transition with a guard

In graphical notation, the tag for a transition has the following format:

[*trigger port*] ? [*trigger message*] ([*trigger parameter list*])

[[*guard condition*]]

/ [*action code*]

In summary, a transition is said to be **enabled** if and only if:

1. it has a trigger for an incoming message that is received on the port specified after the **triggeredBy** keyword, and
2. the guard condition evaluates to true.

3.10.4 Transition Chain by Run-to-Completion

When a state is entered, the state machine may execute another piece of code, this optional snippet is called the **entry code**. Note that the entry code will always be executed when entering the state, regardless of which transition is triggered. Entry code is defined by a block followed by the **entry** keyword, like as follows:

```
state someOtherState {
  entry {
    noop
  }
}
```

When a message arrives, one of the outgoing transitions from the current state may be triggered. If it is triggered, the state machine will execute the optional **exit code** of the state being left. Note also that the exit code will always be executed every time when the state is exited. An exit code is defined by a block preceded by the **exit** keyword, like as follows:

```
state someOtherState {
  exit {
    noop
  }
}
```

The execution of the source state's exit code is followed by the **action code** of the outgoing transition. The action code is executed when the transition is triggered. The action code of the transition is like as follows:

```
transition outgoingTransition : someOtherState -> anotherState {
  action {
    noop
  }
}
```

Once the transition is executed and the state machine finds the next state, the **entry code** of the next state will be executed.

The execution of the entire transition chain—from (1) executing the exit code of the leaving state, (2) executing the action code of the transition, to (3) executing the entry code of the destination state—will not be interrupted by any other message or any context switch to other capsule instances. The execution is said to **run-to-completion**. Hence, to ensure good responsiveness, the behaviour in this chain should not take too much time (and should not have blocking behaviour such as waiting for input), because during this time, no other messages are processed in any of the capsule instances in the interpreter. Such long-running tasks should be delegated to another capsule instance so that they can be executed separately.

3.10.5 Composite States

As a state machine becomes large and complex, it is possible to reduce this complexity by grouping multiple states together and packing them as a lower-level state machine inside a state. Such a lower-level state machine is known as a **sub-state machine**. The states inside a sub-state machine are called **sub-states**. The state that contains the sub-state machine is known as a **composite** or **hierarchical state**. These composite states are useful in that it can abstract away detailed behaviour.

Sub-states in a composite state are visible inside the *entire* capsule; that is, a transition—regardless of its nesting level—has access to all the states of all nesting level inside that capsule. This is the reason why the name of a state must be unique throughout the entire capsule.

It is possible to have a *common* outgoing transition (that responds to the same message in exactly the same way) among all sub-states in a composite state; such a transition is called a **group transition**, which can be simply indicated by a transition whose source state is a composite state.

3.11 Action Code

Urml has a set of action code whose syntax resembles to Java. The action code can be *statements* or *expressions*. The statement part of action code can be used at (1) the body of operations, (2) the action code for transitions, and (3) the entry and exit code for states. Meanwhile, the expression part of the action code can also be used at (1) transition guards, and (2) default values for capsule attributes.

3.11.1 Types of Statements

The action code in Urml can have eight types of statements: (1) send messages, (2) declaring variables, (3) logging, (4) assigning variables, (5) informing the timer, (6) while loop, (7) if statement, and (8) invoking an operation. Below is a discussion for each type of statements.

Sending messages

The send command has the following syntax:

send [*port*].[*outgoing message*]([*parameter list*])

where

- [*port*] is the port of which the outgoing message goes through,
- [*outgoing message*] is the signal that is compatible to the port's protocol (i.e., outgoing signal for base port, incoming signal for conjugated port), and

- [*parameter list*] is the list of parameters that is specified in the protocol definition.

Declaring Variables

Local variables can be declared via the following syntax:

var [*name*]

or

var [*name*] := [*expression*]

where

- [*name*] is the name of the variable
- [*expression*] is the expression to be evaluated.

There is also a boolean variable named *assign* that determined if the second syntactical form is used.

Logging String Expressions

Log statements have the following syntax:

log [*logPort*] **with** [*stringExpression*]

where

- [*logPort*] is an existing port that is defined in the capsule

- [*stringExpression*] is some expression that is evaluated as a string.

Assigning Variables

One can assign attributes in the capsules and local variable in the current call stack via the following syntax:

[*assignable*] := [*expression*]

where

- [*assignable*] is the name of either an existing variable or a capsule attribute
- [*expression*] is an expression to be evaluated and stored into the variable/attribute.

Informing the Timer

The syntax for informing the timer is as follows:

inform [*timer port name*] **in** [*number of milliseconds*]

where

- [*timer port name*] is the name of the timer port, and
- the timer will timeout after a number of milliseconds as specified in [*number of milliseconds*].

For instance, the statement

inform *timer* **in** 5000

will set a timer to fire a timeout event after 5 seconds.

While Loop

The syntax for loops is as follows:

while [*condition*] { [*statement*]$^+$ }

where

- [*condition*] is an expression which is required to be evaluated to a boolean variable at runtime[2]
- [*statement*]$^+$ is at least one statement that is to be executed when [*condition*] evaluates to true.

If Statement

The syntax for **if** statements is as follows:

if [*condition*] { [*thenStatement*]$^+$ }

or

if [*condition*] { [*thenStatement*]$^+$ } **else** { [*elseStatement*]$^+$ }

[2]No type check is performed until runtime.

where

- [*condition*] is an expression which is required to be evaluated to a boolean variable in runtime[2]
- [*thenStatement*]$^+$ and [*elseStatement*]$^+$ are lists of elements to be executed if [*condition*] evaluates to true or false, respectively.

Invoking an Operation

The syntax for invoking an operation is as follows:

call [*operation name*]([*parameter list*])

where

- [*operation name*] is the name that refers to an existing operation in the current capsule
- [*parameter list*] is a list of arguments that is to be evaluated as expression. The list is expected to have the same number of arguments as the number of parameters defined in the operation. This will be checked in both validation and at runtime.

3.11.2 Types of Expressions

Unlike statements, which occur only in action code, expressions may appear in many different places in Urml. Expressions may appear in statements such as message-sending statements, assignments, **if**/**while** statements, and invocations. Expressions also occur in guard conditions.

There are four types of expressions: (1) literals, (2) unary and binary operators, (3) identifiers, and (4) function calls.

Literals

Literals can be integer literals (which can be any integers) or boolean literals (which can be `true` or `false`).

Unary and Binary Operators

Unary operators include boolean negation (`!bool`) and integral negation (`-num`).

Binary operators include conditional-or (`||`), conditional-and (`&&`), addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), modulo (`%`), less-than-equal (`<=`), less-than (`<`), greater-than-equal (`>=`), greater-than (`>`), equal (`==`), and not-equal (`!=`).

Identifiers

An identifier can be either

1. a capsule attribute of the current capsule instance,
2. a local variable in the local environment from the call stack, or
3. an incoming variable from a trigger of the current transition.

Function call

The syntax for a function call is as follows:

[*function name*]([*parameter list*])

where

- [*function name*] is the reference to an existing operation in the current capsule
- [*parameter list*] is a comma-delimited list of arguments to be evaluated each by the ExpressionEvaluator.

3.12 Educational Benefits and Possible Extensions

The primary purpose of the language is to help facilitate the learning of more common real-time languages such as ROOM and UML-RT. First of all, the student can learn the benefits of *abstraction* which is used widely in object-oriented development. Unnecessary complications are factored out through higher abstractions such as capsules and state machines. Secondly, students are able to appreciate the subtleness of concurrent and reactive systems. Thirdly, the action code is made to be minimal so that students can extend it through the addition of other statements and support for other types.

The language is made to be extensible, as more concepts from UML-RT can be directly transferred to Urml:

3.12.1 Non-Wired Ports

Recall that ports from the capsules need to be wired with a static connector structure in order to communicate with each other. These wirings are not needed for special ports known as **non-wired ports**. Instead of using connectors, non-wired ports are programmatically connected at run-time by the use of services provided by the language.

A common use case for non-wired ports is to model clients and servers—a service provided by a server are shared among a number of clients. In such architecture, the server is known as a **publisher**, and the clients are called **subscribers**.

The programmer is able to add a special non-wired port called **service provision point** (SPP) onto the publisher, with each SPP uniquely identified by a name (the **service name**).

The programmer can also add a special port called **service access point** (SAP) onto the subscribers. A SAP is connected to a SPP through the service name. To do that, the SAP performs a lookup from the language's services to find an SPP with a matching service name. If such a name is found, the SPP is connected to the SAP; if not, the connection request is queued until an SPP with a matching name is found in the future.

3.12.2 Message priority

Right now, messages are ordered in the message queue with respect to their time of arrival. The first message that is enqueued is the first message to be dispatch (i.e., first-in-first-out). It is possible to add a feature such that a message with higher priority will be placed before a message with lower priority.

3.12.3 Internal and local self-transitions

Right now, all self-transitions are **external**, in that the state is exited (with the execution of exit code), the transition is execution (the action code), and then the state is entered again (with the execution of entry code). For **internal transitions**, however, the state's exit and entry code blocks are not executed; only the action code of the transition is executed. A variation of internal transitions are **local transitions**, in which all the exit and entry of the *sub-states* are executed, but the exit and entry codes for *the state itself* are omitted.

3.12.4 Deferring and Recalling Messages

A state machine may sometimes process a sequence of multiple messages. During the processing of such a message sequence, if some other message that *does not* belong to the message sequence happens to arrive, it would be better to *defer* that non-belonging message until the entire message sequence is processed.

After a transition is triggered, the plugin developer could make it possible to *defer* that trigger through the **defer** command. The deferred message will then be saved in a defer queue, and will be there until it is recalled through the **recall** command. If the deferred message is recalled, it will be transferred to the back of the message queue, so that it can be dispatched at later point in time.

Chapter 4

Analysis and Validation

Other than being able to execute the models so that they can be tested for their execution semantics, the models also need to be *analyzed* for their correctness and consistency.

In Urml, correctness of a model can be analyzed through various means:

1. Custom validation rules;
2. Object Constraint Language; and
3. Type system development

4.1 Custom validation rules

One of the common use cases for DSL development is to statically validate constraints. In Xtext, customized validation can be added through the `JavaValidatorFragment`. This generator fragment will create the following two classes [12]:

1. From the `src-gen/` directory, an abstract class `AbstractDeclarativeValidator` will be added. This class registers the necessary EMF Validators one can provide constraints for.

2. From the `src/` directory, a class `UrmlJavaValidator` in which the developer can provide customized constraints.

Inside the `UrmlJavaValidator` class, the developer can implement declarative constraints to make sure that the model conforms to his/her need. The constraint method must be preceded by the `@Check` annotation, which signals Xtext to invoke automatically (through Java's Reflection API) when validation takes place. The typed parameter for the constraint method is the type to be validated.

For example, to define the constraint where the name of a `Type` instance must start with a capital, one can implement the following `@Check` method in the validator class [12]:

```
@Check public void checkEntityNameStartsWithCapital(Type type) {
  if (!Character.isUpperCase(type.getName().charAt(0))) {
    warning("Name should start with a capital",
    UrmlPackage.TYPE__NAME);
  }
}
```

From the code above, the `warning()` method will create a warning under the type name in question. A yellow line will appear underlining the type name in question, which is indicated by the constant UrmlPackage.TYPE__NAME. To display an error instead, the developer can use the `error()` method; a red line will appear underlining the type name in question.

Each of the warnings or errors that are discussed above can be associated with a problem-specific code. For instance, for the code above, if one wants to assign the problem-specific code INVALID_TYPE_NAME to the warning, one can have the following code instead [12]:

```
String INVALID_TYPE_NAME = "ca.queensu.cs.mase.urml.InvalidTypeName";
@Check public void checkEntityNameStartsWithCapital(Type type) {
  if (!Character.isUpperCase(type.getName().charAt(0))) {
    warning("Name should start with a capital",
    UrmlPackage.TYPE__NAME, INVALID_TYPE_NAME);
  }
}
```

Once a unique problem-specific code is defined to identify the problem that violates the constraint, the developer can link this code to an appropriate quick fix, which will be described below.

4.1.1 Quick Fix

Quick fixes can be generated through the QuickfixProviderFragment; this will add a class UrmlQuickfixProvider to the UI project. As stated, one can link a quick fix through the problem-specific code. The developer can add the quick fix method to the UrmlQuickfixProvider class that is annotated with @Fix(PROBLEM_SPECIFIC_CODE), where the PROBLEM_SPECIFIC_CODE represents the problem-specific code. For instance, the constraints above might have the matching quick fix method as follows [12]:

```
@Fix(UrmlJavaValidator.INVALID_TYPE.NAME)
public void fixTypeName(Issue issue,
                        IssueResolutionAcceptor acceptor) {
  IModification modif = (IModificationContext context) -> {
    IXtextDocument doc = context.getXtextDocument();
    String firstLetter = doc.get(issue.getOffset(), 1);
    doc.replace(issue.getOffset(), 1,
                Strings.toFirstUpper(firstLetter));
  }
  acceptor.accept(issue,
    "Capitalize name",  // quick fix label
    "Capitalize name of '" + issue.getData()[0] + "'",
                        // quick fix description
    "upcase.png",       // quick fix icon
    modif);
}
```

The `accept()` method that is being invoked in the code above belongs to an `IssueResolutionAcceptor` object and has the following signature:

```
public void accept(Issue issue,
          String label,
          String description,
          String image,
          IModification modification)
```

The three parameters—`label`, `description`, and `image`—are the GUI representation of the quick fix. The actual modification is done in the `modification` parameter. There, the code gets access to the document through the `context` parameter, and then uses the API from Eclipse's `IDocument` to replace the erroneous text [12].

There are other ways to implement the quick fix. For instance, instead of using an `IModification` object, one can use `ISemanticModification` to edit the semantic model directly. For instance, the developer can invoke the method displayed above using the following code instead:

```
ISemanticModification modif =
  (EObject element, IModficationContext context) -> {
    ((Type) element).setName(
      Strings.toFirstUpper(issue.getData()[0]));
  };
acceptor.accept(issue,
  "Capitalize name",  // quick fix label
  "Capitalize name of '" + issue.getData()[0] + "'",
                      // quick fix description
  "upcase.png",       // quick fix icon
  modif);
```

Here, the erroneous object is passed into the `ISemanticModification`, and the erroneous object is of type `Type` [12].

4.1.2 Opportunities for Extension

There are many opportunities to add `@Check` and its subsequent `@Fix` method to make the Urml language more robust. For instance, one can check whether there are cycles in declarations of capsule instance. That is, the following cannot be valid:

```
root capsule A {
   capsuleInstance one : B
}

capsule B {
  capsuleInstance two : A
}
```

This will create a cycle in the capsule instance tree, and will cause an infinite loop when attempting to create such a tree while the interpreter is being initialized. The developer can check the existence of a cycle by visiting each node in the capsule instance tree and check if that node has been visited before. If the node has been visited, this means that the node is causing a cycle; the validation method can underline the offending node and then raise an error. A quick fix method then can be made to create an option to eliminate the offending node.

4.2 Educational Benefits and Possible Extensions

Students can add further validation through other means (1) by using the Object Constraint Language (OCL) and (2) by building a type system with Xtext/TS and Xsemantics.

4.2.1 Object Constraint Language

Since all the models that are generated by Xtext are EMF models, the models can be checked through the use of **Object Constraint Language** (OCL). OCL is an add-on to UML; it is a formal language that is used to express constraints without causing side-effects. The developer can thus use OCL to declare these constraints and attach them to his/her models.

There are a few ways to use OCL in EMF model. The first way by using the OCLinEcore language, which embeds OCL constraints directly in the Ecore files of the EMF models. The problem of using the OCLinEcore language is that the Ecore file is *generated* by Xtext; once the Xtext grammar is modified and the EMF model is generated again, the OCL constraints will be lost. Thus, the EMF model can no longer be generated and must be manually maintained [10].

The second way is by using the Complete OCL, which provides a language in which the OCL constraints *complements* the EMF models in a separate file. There is no need to edit the generated Ecore file of the EMF model; however, one must register the Complete OCL file through a new `CompleteOCLEObjectValidator` object. The registration can be done by modifying the `register()` method from the `UrmlJavaValidator` class [10]:

```
public class UrmlJavaValidator extends AbstractUrmlJavaValidator {
  @Override public void register(EValidatorRegistrar registrar) {
    super.register(registrar);
    UrmlPackage package = UrmlPackage.eINSTANCE;
    URI ocl = URI.createPlatformResourceURI(
         "/pathToProject/model/UrmlValidation.ocl", true);
    registrar.register(package,
         new CompleteOCLEObjectValidator(package, ocl));
  }
}
```

4.2.2 Type system development

Another common case for DSL development is type checking. To do this, one can develop a type system for the DSL. A type system is a syntactic framework for classifying each phrase, according to the types of values they compute. Consider the following expressions:

var int $a,\ b,\ c$;

calc int $x = a$;

calc int $y = a + c$;

calc bool $z = a * a$; // *wrong*

Note that this program is structurally and syntactically correct. What is incorrect is the type of the last statement. The results of $a * a$ is of type **int**, but one cannot assign this integer value to the variable z, which is of type **bool**. A type system is built so that proper type inference like this is made according to the typing constraints that is defined by the developer [4].

```
axiom BooleanLiteralType
  G |- BooleanLiteral lit :
      XsemGuiDslFactory::eINSTANCE.createBooleanType

rule AttributeRefType
  G |- AttributeRef attrRef : Type type
from { G |- attrRef.attr : type }

rule LengthOfType
  G |- LengthOf len : XsemGuiDslFactory::eINSTANCE.createIntType
from { G |- len.expr : var StringType stringType }

rule WidgetContentType
  G |- WidgetContent widgetContent : Type type
from { G |- env(G, 'widgetcontent', Attribute) : type }
```

$$\Gamma \vdash \mathtt{true} : \mathtt{boolean}$$

$$\frac{\Gamma \vdash \mathtt{attr} : \mathtt{T}}{\Gamma \vdash \mathtt{ref\ attr} : \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{exp} : \mathtt{string}}{\Gamma \vdash \mathtt{lengthOf(exp)} : \mathtt{int}}$$

$$\frac{\Gamma \vdash \Gamma(\mathtt{widgetcontent}) : \mathtt{T}}{\Gamma \vdash \mathtt{widgetcontent} : \mathtt{T}}$$

Figure 4.1: Examples of rules and axioms in Xsemantics [2]

In Xtext, there are two type system frameworks available (each is an Xtext add-on): (1) Xtext/TS [25] and (2) XSemantics [2].

Xtext/TS

Xtext/TS is a Java API[1]–based type system framework that generally has the following three building blocks:

1. **Type assignments**, which are used to define various types. For instance, the types `int` and `bool` have a fixed type and need to be defined.

2. **Type calculation rules**, which are used to compute a type, which are typically inferred from the types of its constituent elements.

3. **Typing constraints**, which are used to verify that the types of certain elements conform to the expectations as defined by the developer.

Once all these three building blocks are completed, the target type system is then generated to integrate into Xtext's validation framework.

[1]The second version of Xtext/TS also comes with a DSL.

Xsemantics

Xsemantics is a DSL that is used to implement type system for an Xtext language. A definition in Xsemantics consists of a set of **judgments** and a set of **rules**, which have a **conclusion** and a set of **premises**. The details about these syntactic rules can be referred from the documentation [2]. The nice thing about Xsemantics is that it resembles formal deduction rules in a formal setting; refer to Figure 4.1 for examples of Xsemantics rules. Thus, it is suitable for developers who are familiar with formal type systems and operational semantics. After an Xsemantics definition has been made, the framework then automatically generates Java code that can be used in Xtext for scoping and validation.

Summary

There are many educational benefits that are derived from this step of Urml. For instance, one can assign various `@Check` methods for students as study questions, so that static analysis can be studied in small pieces. Also, students can learn OCL by, for example, completing an assignment on extending the model validation for Urml models using OCL. Additionally, type inference can be studied by building type systems. It is especially fortunate in this case because multiple frameworks are developed for Xtext so that different frameworks can be tested.

Chapter 5

Implementation of the Interpreter

5.1 Code Generation and Interpretation

After the AST of the program is created, the *execution semantics* can be implemented in two ways:

1. **Interpretation:** by traversing through the AST and performing actions based on the data in the AST; or

2. **Code generation:** by transforming the data in the AST into a program in a lower-level language, such as a general-purpose programming language like C++ or Java.

This alternative between interpretation and code generation is shown in Figure 5.1 [26].

In Urml, model execution is primarily done by an interpreter, and hence, in this project, we will implement an interpreter to map the AST of a program into its execution semantics.

Since the structure of expressions and statements in Urml is largely recursive, for simplicity, we can recursively traverse through the expressions/statements tree and

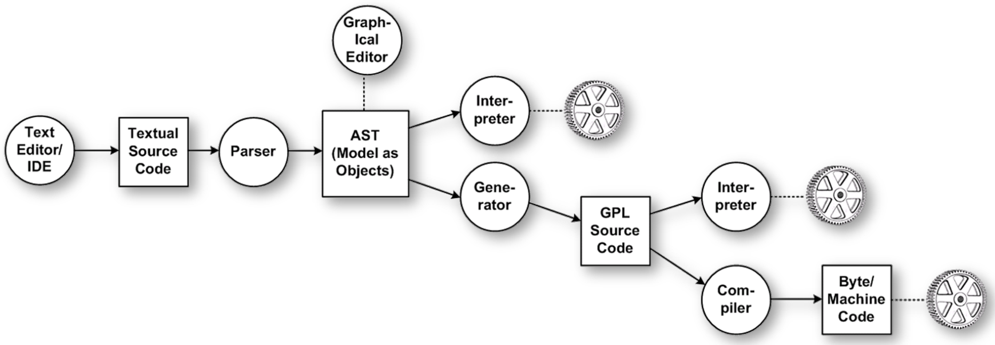


Figure 5.1: An AST can be either (1) interpreted directly or (2) code-generated into a lower-level programming language [26].

evaluate them.

5.2 Intermediate Representation of the Model

Once the text has been parsed by Xtext, the relevant data from the text must be stored in memory. How to represent this data in memory? This in-memory object graph—also known as the *abstract syntax tree* (AST), the *document object model* (DOM), the semantic model, or simply the model—is stored as an EMF model. Xtext *infers* these EMF models from the grammar, which is specified in the Chapter 3. Basically, the EMF model abstracts out all the syntactical information from the textual file; it is the essence of the model.

The concepts that an EMF model must conform are summarized in a UML class diagram in Figure 5.2 [12]. Every element in the EMF model is an instance of the **EObject** class, and an EObject is also an instance of an **EClass**. Like original classes, each EClass can have a set of **EAttributes**; the domain of values for an EAttribute is defined by its **EDataType**. EMF predefines some EDataTypes, which are essentially

Java primitive types such as boolean (defined by EBoolean) and immutables such as String (defined by EString). The “E-” prefix provides a distinction from Java standard types, decoupling the model from a Java implementation.

Other than EAttributes, an EClass can also have a set of **EReferences**. An EReference contains a *containment flag* which determines if that EReference is a *containment reference* or a *cross-reference*. For containment references[1], the referenced object is declared or *defined* inside the referencing object. Each element that is contained can have at most one container. In cross-references[2], the referenced object is defined elsewhere; the referencing object simply use the cross-reference to link to the referenced object [12].

Note from Figure 5.2 that both EAttributes and EReferences are subclasses of **EStructuralFeature**; an EStructuralFeature has an attribute *name*, which specifies the structure’s name, and the attributes *lowerbound* and *upperbound*, which specifies the structure’s cardinality.

Note also from Figure 5.2 that both EClasses and EDataTypes are subclasses of **EClassifier**, which defines the name of the EClass/EDataType [12].

5.3 Accessing the EMF Model

The EMF model of a file can be accessed through the **EMF Persistence API**, in which the basic unit of persistence is called a **resource**, of which a class diagram is shown in Figure 5.3. A Resource interface is a persistent container of EObjects (stored in an EList), the actual location of which is identified by its URI. One can access the EObjects from its containing Resource via the getContents() method; in

[1]A containment represents a composition (“A has-a B”) relation in UML.

[2]A cross-reference represents an association (“A knows B”) relation in UML.

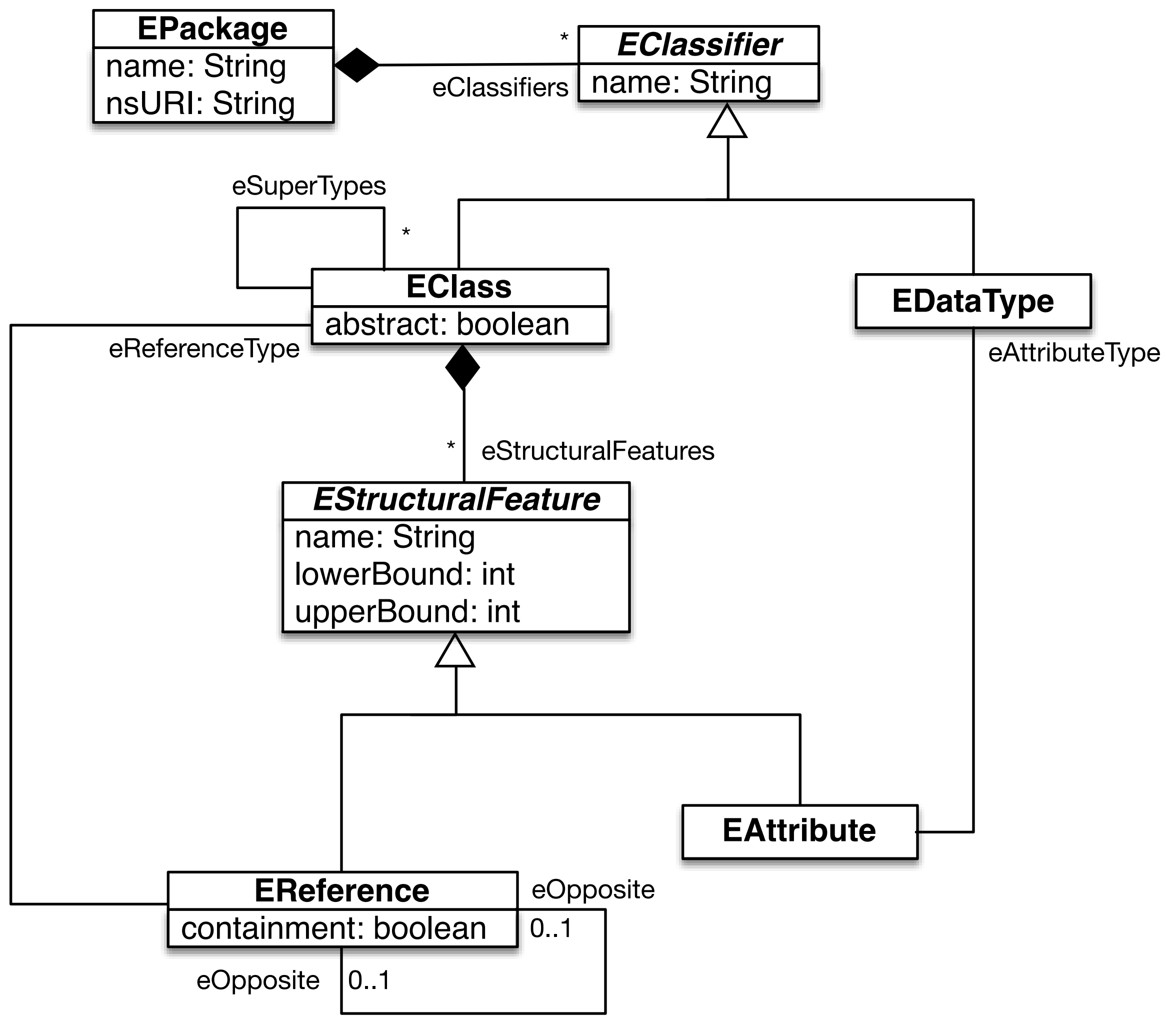


Figure 5.2: Concepts of an EMF model [12]

the opposite direction, an EObject can access its container Resource via eResource() [23].

The highest level of the persistent API is the **resource set**. It serves as the container of its resources, allowing references between each other. Contained in the ResourceSet interface is the getResource() method, which normalizes the given URI, checks if any resource in the resource set has a matching normalized URL, and if so, returns the existing resource.

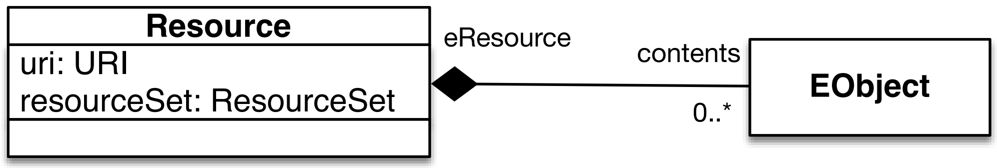


Figure 5.3: Conceptual model of resource and its contents [23]

Once the getResource() method returns the resource successfully, and the getContents() method from the resource gets the EObject of the root model successfully, the program has direct access to the EMF model of the parsed file [23].

After the persistent copy of the EMF model is accessed, the interpreter can be started to use this model.

5.4 Starting the Interpreter

The interpreter is initialized by instantiating the UrmlInterpreter, with the model to be interpret, an input stream, an output stream, and user configuration as its parameters:

```
UrmlInterpreter(Model m, InputStream in, OutputStream out,
    ExecutionConfig config);
```

The **user configuration** consists of the following:

- **Multiple Transitions.** The model becomes non-deterministic if and only if more than one transition is enabled at the same time. If such a case happens, what can the interpreter do? The interpreter allows the user to select one of the three options:

  1. Run the first transition occurring in the text.
  2. Select the transition randomly.

3. Interactively ask the user which of the enabled transitions to take (hence the need of using an input stream by the interpreter).

- **Exit Conditions.** Models without a final state will run indefinitely, and hence, we can set some limit on when the interpreter would exit. There are three options for such exit conditions:

  1. Exit interpretation after a certain number of seconds.
  2. Exit interpretation after a certain number of transitions are executed.
  3. Interpret indefinitely.

  There is a duration (as an integer) which determines how many seconds or transitions may elapse before exiting the interpreter.

Once the interpreter is initialized, one can start the interpreter by calling the its `interpret()` method.

5.4.1 Preprocessing: Building a capsule instance context tree

The interpreter first delegates the preprocessing of the model to the CapsuleContext-TreeGenerator class.

The CapsuleContextTreeGenerator creates a tree of capsule instance context—known as **CapsuleContext** in code—which is a wrapper of a capsule instance. This tree is built to store information about the relationships between a composite capsule and its sub-capsules. A node in the tree (implemented as `TreeNode`) contains two things: (1) a capsule instance which is a composite node called "parent", and (2) a list of nodes that represents the "children" capsule instances that the parent composite capsule contains.

To start building the tree, we need to find the root of the tree first. Find the root capsule from then model, which is distinguished by the `root` keyword in its definition. Then we create a capsule instance that has the root capsule as its type.

Note that Xtext does not instantiate the root capsule instance in the object graph; that is, only non-root capsule instances are instantiated. Thus, we must assume that every model must have one (and only one) capsule instance having the root capsule type. Then, the interpreter creates the root capsule on its own from the UrmlFactory singleton:

```
CapsuleInstance root = UrmlFactory.eINSTANCE.createCapsuleInstance();
```

We set that capsule instance as the root, and then wrap that root capsule instance with a root capsule instance context.

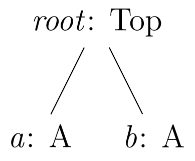
Once the root of the capsule instance context tree is created, we can deal with the children. We do the following recursively, letting the root capsule instance context as the parent capsule instance first:

1. First, get the capsule instance that the parent capsule context contains.
2. Then, for each child capsule instance that is contained within the parent capsule instance:
   (a) Wrap that child capsule instance into a child capsule context
   (b) Assign that child capsule context as a child of the parent capsule context
   (c) Perform this algorithm for each child capsule context, now using the child capsule context as the parameter for parent capsule instance.

For instance, consider the following model:

```
root capsule Top {
  capsuleInstance a: A
  capsuleInstance b: A
}
capsule A {}
```

This will produce the following tree:



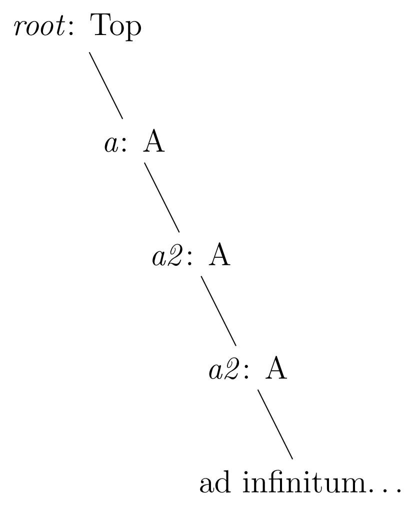
Once the recursive tree structure is formed, we return to the root capsule context, which is returned by the CapsuleContextTreeGenerator.

Error: Tree of infinite height

Note that a legal tree cannot form a cycle; otherwise, an infinite loop will occur when building the tree. For instance, consider the following model:

```
root capsule Top {
  capsuleInstance a: A
}
capsule A {
  capsuleInstance a2: A
}
```

This will produce the following tree of infinite height:



This is an error; validation code should be made to make sure such cycles do not occur in the model.

5.4.2 The main interpreter loop: Scheduling the execution of capsule instances

After the model is preprocessed, the interpreter delegates the execution of the capsule instances to the `CapsuleLoop` class. The interpreter first runs the initial transition of all the capsule instances so that the current state will not point to null. Then, in an infinite loop:

1. For each capsule instance, use the selection method[3] in the configuration file to find[3] the next enabled transition.
2. Once the next transition is selected, execute it[3].
3. If one of the following occurs, break out from the infinite loop:

[3]Finding the next transition, selecting the transition, and executing the next transition will all be discussed in Section 5.5.

(a) **User configuration.** The user might specify a number of seconds to be elapsed before the interpreter quits. If this number of second is reached, break out from the loop.

(b) **User configuration.** The user might specify a number of transitions to be executed before the interpreter quits. If this number of transitions is reached, break out from the loop.

(c) **Final states.** A capsule instance context will stop going to the next state if it reaches a final state. If all the capsule instance contexts in the tree (other than those capsule contexts that do not have a state machine) have reached a final state, then break out from the loop.

After the interpreter breaks out from the loop, it finishes its execution.

5.5 Running the Interpreter State-by-State

Recall that the interpreter loop needs to (1) execute the initial transition for all capsule instances, (2) find the next transition from the current state and (3) execute the next transition itself. These three tasks are performed by the StateExecuter class through the following three methods, respectively:

1. `public void executeInitialTransition(CapsuleContext);`

   This method is described in Section 5.5.1.

2. `public List<Transition> findNextTransitions(CapsuleContext);`

   This method is described in Section 5.5.2.

3. `public void executeNextTransition(Transition, CapsuleContext);`

   This method is described in Section 5.5.3.

5.5.1 Execute the initial transition

First, go through the state machine[4] and look for the initial transition. If the initial transition cannot be found, we assume that there is no states at all in the state machine. On the other hand, if an initial transition is found, we run the action code for the initial transition, run the entry code for the first state, then set the destination state of the initial transition as the current state.

5.5.2 Find the next transitions

First, check if the current state is a final state; if so, we assume the state machine has stopped and return an empty list.

After that, we accumulate the possible next transitions. We first find all the outgoing transitions from the current state; we include the *group* outgoing transitions from the *composite* states we are in too. Then, we filter the transitions based on the following two conditions:

1. We accept only transitions whose guard conditions return true; for those without guards, we assume the guards returning true.

2. We accept only transitions that are triggered by some message.

The transitions that satisfy these two conditions are known as **enabled transitions**. If we have more than one transition after the first two filtering processes, we will have more than one enabled transitions and the behaviour of the interpreter will be non-deterministic. We resolve this non-determinism by using one of the strategies the user chooses from the execution configuration, as described in Section 5.4.

[4]A capsule can have at most one state machine. The grammar could not enforce this and hence a validation rule should be customized for this constraint.

This filtering process is delegated to the predicates classes, which we will discuss in Section 5.7.

5.5.3 Execute the next transition

To execute the next transition, we run the exit code of the source state, the action code for the transition, and the entry code of the destination state. Note that we can have transition chains that exit and enter multiple composite states. We execute this transition chain by using the least-common ancestor algorithm (which we will discuss in Section 5.6). Also, note that a transition might be triggered by an incoming message containing some marshalled parameters (called `IncomingVariables` in code). Once the transition has executed its action code, such parameters are no longer relevant, and thus we want to eliminate these parameters after the transition has been executed.

After the related action code is executed, we set the previous state as the current state and the current state as the destination state of the transition.

Now, we encounter either of the two cases with regards to the current state: (1) the current state is a simple state, and (2) the current state is a composite state (i.e., the current state has a sub-state machine). If we are in the first case, then we are done. Otherwise, we try to find the initial transition of the sub-state machine, then execute its action code and the entry code of its destination state. Then set the destination state as the current state. We keep performing these executions of initial transitions until the destination state is no longer a composite state.

5.6 Least Common Ancestor Algorithm

Recall that a transition can be chained. A transition can be connected from a sub-state machine in one composite state to a state machine in another composite state. Executing this transition requires executing the exit codes for multiple source state machines in cascade, and then executing the action code of the said transition, and then executing the entry codes for multiple destinations state machines in cascade. Running such multiple exit, action, and entry code blocks can be done easily through the least common ancestor algorithm.

The algorithm can be summarized as follows:

1. Let `fromState` be the source state, and `toState` be the destination state.

2. Find the ancestor states of `fromState` and store the list into `fromAncestors`; do the same for `toState` and store the result in `toAncestors`. Note that one can store the elements of `fromAncestors` and `toAncestors` in form of deques, such that their elements can be pushed into and popped out at both sides of the list.

3. Remove the common ancestors of both `fromAncestors` and `toAncestors` by doing the following loop:

   (a) First, we peek the ancestor side of both `fromAncestors` and `toAncestors` and compare the peeked elements.

   (b) If the two are not equal, or if either `fromAncestors` or `toAncestors` is empty, break out from the loop.

   (c) Otherwise, the peeked elements are equal; we pop off the peeked elements from both `fromAncestors` and `toAncestors`, and then we continue running

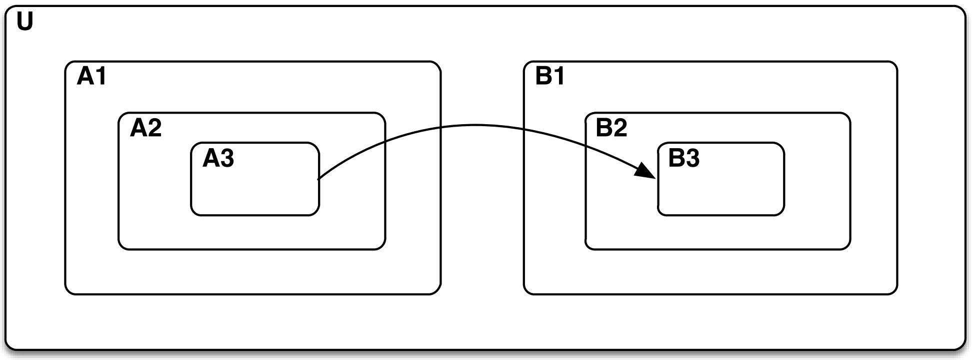


Figure 5.4: A transition chain

the loop.

4. Then, from the remaining states in `fromAncestors`, we pop off one state from the children side of `fromAncestors` and then run the exit code of the popped state. We keep doing this step until `fromAncestors` is empty.

5. Once the exit codes are executed, we execute the action code for the transition, noting that the incoming variables of the triggers are relevant only for this action code, and thus these variables are all flushed out after the action is executed.

6. From the remaining states in `toAncestors`, we pop off one state from the ancestor side of `toAncestors`, and then run the entry code for the popped state. We keep doing this yep until `toAncestors` is empty.

Once these steps are followed, we finished executing the exit-action-entry code of the transition chain.

5.6.1 An Example of a Transition Chain

Consider a transition chain as shown in Figure 5.4. Consider also the only transition in the diagram; it is a transition chain because it cross multiple composite states.

The source state of the transition is A3, and so the source configuration—including its ancestor states—is $(\mathrm{U}, \mathrm{A1}, \mathrm{A2}, \mathrm{A3})$; the destination state of the transition is B3 and so the destination configuration—including its ancestor states—is $(\mathrm{U}, \mathrm{B1}, \mathrm{B2}, \mathrm{B3})$.

First, we eliminate the common ancestors from both source and destination configurations, so that the source configuration becomes $(\mathrm{A1}, \mathrm{A2}, \mathrm{A3})$ and the destination configuration becomes $(\mathrm{B1}, \mathrm{B2}, \mathrm{B3})$.

Then, we begin to pop out the states from the children side of the source configuration. Here, we pop out A3 and run its exit code, pop out A2 and run its exit code, and pop out A1 and run its exit code.

After that, we run the action code of the transition.

Finally, we begin to pop out the states from the ancestor side of the destination configuration. Here, we pop out B1 and run its entry code, pop out B2 and run its entry code, and pop out B3 and run its entry code.

In summary, we execute the action code in the following order:

1. A3 exit code
2. A2 exit code
3. A1 exit code
4. transition action code
5. B1 entry code
6. B2 entry code
7. B3 entry code

5.7 Filtering for Enabled Transitions

Here we discuss which conditions the transition must meet in order to be enabled. The two conditions that need to be met are as follows:

1. The transition either:
   (a) has no guard, or
   (b) meets the guard condition.
2. The transition either:
   (a) Has no message trigger and has no timeout triggers, or
   (b) Has a message trigger and that trigger is activated for the said transition, or
   (c) has a timeout trigger and that trigger has been expired for the said transition.

We now explain separately the implementation of how each condition is evaluated.

Condition 1(a) *Has no guard.* This is trivial to implement: simply check if the guard is null; if so, the transition has no guard.

Condition 1(b) *Meets the guard condition.* This is implemented in the `GuardPredicate` class. We first access the guard expression, and then interpret it with the `ExpressionEvaluator` class, which will be discussed in Section 5.11. Then, we return the result.

Condition 2(a) *Has no message trigger and has no timeout triggers.* This is trivial to implement and is implemented in the `DefaultTriggerPredicate` class. We check if the number of received triggers is zero, and the timer port for the trigger is null.

Condition 2(b) *Has a message trigger and that trigger is activated for the said transition.* This is implemented in the `MessageTriggerPredicate`. For each incoming triggers from the transaction, we check if that incoming trigger is contained in the message queue of the capsule.

Condition 2(c) *Has a timeout trigger and that trigger is activated for the said transition.* This is implemented in the `TimeoutTriggerPredicate` class. We check if the timer port from the said transition is contained in the lookup table that stores timeout instants. If it exists, we get the timeout instant from the hash table and check if the current time has passed beyond the specified timeout instant.

5.8 Selecting Multiple Enabled Transitions to Resolve Non-Determinism

Once all enabled transitions are determined, we can return a list of enabled transitions. If there are more than one transition in the list, the interpreter has entered a non-deterministic state. At this stage, there are ways to resolve such non-determinism.

This is implemented in the `TransitionSelector` class, whose only public method is:

```
public Transition select(List<Transition> transitions);
```

The method gets the multiple transitions as a list. If there is no transitions in the list, the method returns null; if there is only one transition in the list, the method return that transition. However, if there are multiple transitions in the list, the program is diverted into three execution paths based on the user configuration that is discussed in Section 5.4:

1. **First transition.** Simply return the first element of the list.
2. **Random transition.** Generate a random number in the range from 0 to the size of the list minus 1, then take the element that has the random number as index from the list and return it.
3. **Interactive mode.** Display a list of selection on the screen and let the user choose one, then return the transition that the user chooses.

5.9 Interpreting Action Code: Multiple Dispatch

The interpretation of statements and expressions heavily relies on a feature called **multiple dispatch**—also known as multimethods [6]—from Xtext, which is not available in the default Java language.

5.9.1 Motivation

By default, Java supports single dispatch for its method calls to support polymorphism. The method call is dispatched dynamically based on the runtime type of *only* one argument. This "special" argument is distinguished syntactically by being placed before a dot in making a method call. For instance, in the invocation `specialArg.call(other, arguments)`, the `call()` method is invoked based on the runtime type of `specialArg` but on the *compiled* types of `other` and `arguments`. As the result, it is possible for the invocation `lion.call()` to produce a roar and the invocation cat.call() to produce a meow. The methods that the two calls dispatch are different. Having such special semantics for the special argument is particularly useful to implement polymorphic behaviour in object-oriented programming.

On the other hand, when one is *overloading* a call and wants the behaviour to depend on the runtime types of the other arguments, in plain Java, one must use a chain of **instanceof** expressions to dynamically delegate the call to other methods. In other words, such chain of **instanceof** expressions and dynamic typecasts as shown in the following code will occur:

```
public void doThings(Type objOfType) {
  ...
  if (objOfType instanceof SubTypeA)
    doForA( (SubTypeA) objOfType );
  else if (objOfType instanceof SubTypeB)
    doForB( (SubTypeB) objOfType );
  ...
}
void doFor(SubTypeA obj) { ... }
void doFor(SubTypeB obj) { ... }
```

In this case, the difference of types for SubTypeA and SubTypeB and the invocations of doFor() methods are tightly coupled to the **if**-**instanceof**-cast cascades in the doThings() method, which makes the code potentially more error-prone.

There are several ways to mitigate this problem: one by using the visitor pattern (Section 5.9.2) and another by using multiple dispatch (Section 5.9.3).

5.9.2 Visitor Pattern

The **Visitor Pattern** [14] allows the programmer to separate the code in doFor() methods from its caller, without significantly modifying the structure of the caller. For instance, consider the following code as an alternative code structure:

```
interface IVisitor {
  public void doFor(SubTypeA obj);
  public void doFor(SubTypeB obj);
}
interface IVisitable {
  public void accept(IVisitor v);
}
class SubTypeA extends Type implements IVisitable {
  public void accept(IVisitor v) { v.doFor(this); }
}
class SubTypeB extends Type implements IVisitable {
  public void accept(IVisitor v) { v.doFor(this); }
}
private class DoForVisitor implements IVisitor {
  public void doFor(SubTypeA obj) { /* code for A */ }
  public void doFor(SubTypeB obj) { /* code for B */ }
}
class Client {
  void doThings(IVisitable objOfType) {
    ...
    objOfType.accept(new DoForVisitor());
    ...
  }
}
```

In the client code in the Client class, when the coder wants to call the `doFor(obj)` method, where `obj` is of type `SubTypeA` or `SubTypeB`, the coder calls the `IVisitable.accept()` method, which in turn calls the `doFor()` method in the visitor. In general, the visitor pattern uses single-dispatch from Java *twice* and hence is sometimes called "double-dispatch": (1) the client dispatches the `accept()` method into the `IVisitable` in runtime, and (2) the `IVisitable` object then dispatches to `doFor()` method into the visitor [14].

5.9.3 Introducing Multiple Dispatch

In dynamic languages such as Xtend, Groovy, and Scala, multiple dispatch can be used instead, where the method is dynamically dispatched based on the runtime types of *all* its arguments, including the arguments *after* the dot of the method call. This is hence a generalization of the double-dispatch as shown in the Visitor pattern. For instance, consider the following code in Xtend:

```
class Type {}
class SubType extends Type {}
def dispatch foo(Type val) { "foo" }
def dispatch foo(SubType val) { "bar" }

Type t = new SubType()
```

The expression `foo(t)` would evaluate to "`bar`" rather than "`foo`".

Multiple dispatch provides flexibility to group related code based on their subtypes, so extending a class becomes very trivial. It does not require one to prepare the arguments through chains of **instanceof** and dynamic casts, and so it requires less code to write. Nevertheless, because all the arguments must be checked against their types this form of dispatch has a significant overhead—with a median overhead of 13.7% to the maximum of 47%. There are not many use cases where double or triple dispatch is needed [15].

In Urml, the evaluation of expressions and the execution of statements are done via an Xtext library that implements multiple dispatch: `PolymorphicDispatcher`$\langle$`RT`$\rangle$, where `RT` is the return type of the methods to be dispatched.

5.9.4 How Multiple Dispatch is Used in Xtext

The `PolymorphicDispatcher` class is poorly documented; however, it provides several factory methods and constructor are provided for public to generate a dispatcher. One particularly useful factory method is the following:

```
public PolymorphicDispatcher createForSingleTarget(
   String methodName, int minParams, int maxParams, Object target);
```

The parameters of this factory method is as follows:

- the `methodName` is the name of the methods to be dispatched;
- the `target` is the class in which the methods are contained;
- the `minParams` is the minimum number of parameters the dispatched methods may contain; and
- the `maxParams` is the maximum number of parameters the dispatched methods may contain.

To invoke a call, one can simply call the `invoke()` method onto the `PolymorphicDispatcher` object that is created by the factory method.

5.9.5 What is Multiple Dispatch Used for in Urml?

In Urml, multiple dispatch is used primarily in `ExpressionEvaluator` and `StatementExecuter`. Each of the two has a class attribute that is a polymorphic dispatcher. Each also has a number of "`compute(T stmt, CapsuleContext ctx)`" methods where T is a subtype of Statement or Expression. We rely the `invoke()` method to dispatch to the appropriate interpretation method for each subtype of `Statement` or

`Expression`. Discussed below is how different types of statements (Section 5.10) and expressions (Section 5.11) are interpreted in Urml.

5.10 Interpreting Statements

Statements are all implemented in the `StatementExecuter` class. Discussed below is how each of the statements are interpreted in Urml.

5.10.1 Sending messages: interpretataion

A signal travels from the source capsule S through some end port p_S, and possibly through some connections, and the signal is eventually captured by the target capsule instance T through a target end port p_T. A problem arises: how can one travel through the connections from $\langle S, p_S \rangle$ to $\langle T, p_T \rangle$? Recall that a capsule instance tree is prepared during pre-processing; we can use this tree to keep track of travelling through the connections by following some path from $\langle S, p_S \rangle$ to $\langle T, p_T \rangle$.

One can divide the connection path into two parts: first to "zoom-out" the structure diagram from the sub-capsule to its parent capsule, and secondly to "zoom-in" the structure diagram from a composite capsule to one of its children capsules:

1. the **zoom-out** part, where we are traversing *up* the tree from a child capsule to its parent; and

2. the **zoom-in** part, where we are traversing *down* the tree from a parent capsule to one of its children capsules.

Additionally, one can define two kinds of ports in a structure diagram. Refer to the structure diagram in Figure 5.5. Ports that are sitting at the boundary diagram

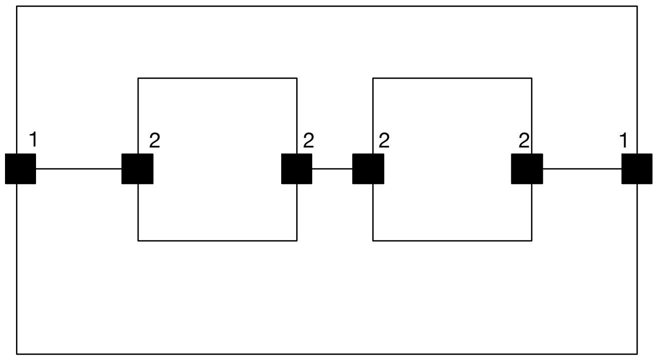


Figure 5.5: The ports labelled "1" are diagrammatically external ports and those labelled "2" are diagrammatically internal ports.

are said to be **diagrammatically external ports** and those that are sitting inside (and not at the boundary) of the structure diagram are said to be **diagrammatically internal ports**[5].

Naming of diagrammatically external and internal ports. Note that when defining a connector in the capsule, the naming of diagrammatically external ports is different, in that the name of the capsule instance is omitted because it is already inferred as part of structure diagram of the parent capsule instance. On the other hand, while defining a connector in the capsule, one *must* name the capsule instance for a diagrammatically internal port because one has to specify *which* sub-capsule the connector is bound to.

While one is travelling through the connectors by zooming out and zooming in the structure diagram, one should keep track of the the pair $\langle C, p_C \rangle$, where C is the current capsule instance and p_C is the current end point. This pair tells one exactly which port of which capsule instance one is currently travelling at.

[5]Note that the ports through the zoom-out segment must be all diagrammatically external; having an diagrammatically internal port in the middle of the zoom-out segment would interrupt the connector segment, making all the connectors invalid.

Zooming out the structure diagram

The zoom-out algorithm may be done only once or done repeatedly. Let the current capsule instance be the source capsule instance and the current port be the source port. If the current port is a diagrammatically internal port, then zoom out just once. If the current port is a diagrammatically external port, keep repeating the zoom-out algorithm until one hits a diagrammatically internal port[6].

How is the zoom-out algorithm done? First, one sets the current capsule instance to its parent, and then from the parent, look for a connector that contains the current port. If such a connector is not found, the connector as a whole is invalid and thus we need to raise an error; otherwise, set the current port to the port at the opposite site of the connector.

If the current port one has reached is a diagrammatically external port, one continues repeating the zoom-out algorithm; otherwise, if one has reached a diagrammatically internal port, then one is done and can proceed to the zoom-in algorithm.

Zooming in the structure diagram

The zoom-in algorithm is done repeatedly until one hits the target capsule instance.

First, find the child node that contains the current capsule instance; if such a child node cannot be found, raise an error. Set that child node as the current capsule instance. Then, find the connector in the capsule instance that contains the current port. If such a connector is not found, the capsule one is in *is indeed* the target capsule; return this target capsule instance and the current port as the resulting

[6]Note that the path of this zoom-out segment must be finite (unless there is some cycle in the capsule instance tree, which is illegal); that is, one *will* eventually hit a diagrammatically hit a diagrammatically internal port because the structure diagram of the root capsule *cannot* have a diagrammatically external port.

location pair. Otherwise, find the port on the opposite side of the connector and set that port as the current port, then continue repeating this zoom-in algorithm.

Marshalling parameters and enqueuing the message to message queue

Once one has found the target capsule instance-port pair, the interpreter evaluates the arguments of the signal using the `ExpressionEvaluator` class (which will be discussed in Section 5.11), packages or marshals the target port, the signals, and the evaluated parameters into a `MessageDesc` object, and then inserts the resulting `MessageDesc` object into the message queue of the target capsule instance.

5.10.2 Declaring Variables: Interpretation

To store the local variables being used, one needs to maintain the state of the capsule instance. The `CapsuleContext` captures the state of the capsule currently running. Inside the state, there is a hash table called `callStackOfLocalVariables` that stores the environment that maintains the state of the variables at the current call. The hash table is of type `String` $\rightarrow$ `Value`, where the string contains the name of the variable, and the value is the current evaluated value of the variable. If the boolean variable `assign` is true, the expression is first evaluated by `ExpressionEvaluator`, and the result of the evaluation is of type `Value`. Thus evaluated result is then stored in the hash table. On the other hand, if boolean variable `assign` is false, no evaluation is done and a null result is stored in the hash table.

5.10.3 Logging String Expressions: Interpretation

The mechanism for log statements is simple. One simply prints the name of the log port and the evaluated string expression onto the output console. Note that this is the only place in Urml grammar that currently supports string expressions and string concatenation. Inside the `StateExecutor` class, another multiple dispatcher is created to perform these operations.

5.10.4 Assigning Variables: Interpretation

If [*assignable*] is a local variable, then the interpreter stores the evaluated result in the hash table which is the current environment in the call stack of the current capsule instance. If [*assignable*] is a capsule attribute, then the interpreter stores the evaluated result in the hash table which contains the capsule attributes of the current capsule instance.

5.10.5 Informing the Timer: Interpretation

The implementation of this command is by adding a timeout event to the timeout hash table in the `CapsuleContext`. In there, the current time plus the number of milliseconds is added, and the state machine will check whether the current time has exceeded the added time when the timeout message is triggered.

5.10.6 While Loop: Interpretation

The implementation of a **while** loop is simple. One runs the following in an infinite loop. Interpret the condition expression into a `Value` result. Check if the result is boolean; if not, throw an exception. Then, check if the result is false; if it is, break out from the infinite loop. Otherwise, execute the list of statements, and continue with the loop.

5.10.7 If Statement: Interpretation

Like **while** loops, the implementation for **if** statements is simple. First, evaluate the condition expression into a `Value`, and then check if the value is a boolean variable; if not, raise an exception. If the result is true, however, then execute the list of statements from $[thenStatement]^+$; if the result is false, then execute the list of statements from $[elseStatement]^+$.

5.10.8 Invoking an Operation: Interpretation

The implementation for invoking an operation is as follows. First, the interpreter checks if the number of formal parameter in the operation definition is equal to the number of actual arguments in the invocation; if not, then the interpreter raises an exception.

After that, the interpreter creates a new environment as a hash table of type `String` → `Value`, and then stores the arguments as local variables in that environment. The interpreter pushes that environment into the call stack, and then runs the statements in the operation. If a **return** statement is detected, a `ReturnStatementSignal` is thrown and the invocation is complete; the interpreter catches the signal and then pops the used environment off the stack.

5.11 Interpreting Expressions

Expressions are evaluated by the `ExpressionEvaluator` class. Discussed below is how each type of expressions is to be interpreted in Urml.

5.11.1 Literals: Interpretation

The implementation simply wraps the literal value into a `Value` object.

5.11.2 Unary and Binary Operators: Interpretation

The implementation is simple. The interpreter unwraps the `Value` objects, evaluates the unwrapped values in Java terms, wraps the result, and returns the wrapped `Value` result.

5.11.3 Identifiers: Interpretation

First, the identifier is checked for its existence. If it exists in one of the hash tables, the interpreter returns it; otherwise, the interpreter raises an exception about its non-existence.

5.11.4 Function call: Interpretation

The implementation is similar to that for invocation calls. The interpreter checks if the number of formal parameters defined in the operation is equal to the number of actual arguments in the function call; if not, the interpreter raises an exception. The interpreter then creates a new stack frame for the function being called. The actual arguments are evaluated and assigned to their formal parameters as local variables in the new stack frame, and the new stack frame is pushed into the call stack.

Once the stack frame is prepared, the interpreter executes the statements in the body of the function. When a **return** statement is detected, the interpreter throws a return signal, which will be caught by the caller. The return value is stored and the stack frame for the old function call is popped from the call stack.

If the interpreter reaches the end of the operation without detecting a **return** statement, it is assumed that the programmer has forgotten to write a **return** statement. This is an error, and an exception is thrown.

5.12 Educational Benefits and Possible Extensions

There are many benefits for providing an interpreter for students. First of all, students can learn about different ways of giving execution semantics to a language—either by compilation or by interpretation. Secondly, students can learn the mechanics of an interpreter and different approaches to implement an interpreter. Thirdly, students can extend the interpreter in different ways. One may build a compiler and use it instead of the interpreter, so that the compiled code can be used in niche Java platforms such as Lego Mindstorms and Arduino. One may extend the interpreter with various model checking features such as exhaustive exploration, assertion checking, collection of execution traces, etc. All in all, students will learn the principles, patterns and practices in executing models through building an interpreter and a compiler.

Chapter 6

Examples and Case Studies

This chapter contains examples of Urml models that are included for this thesis. The first section of this chapter discusses how to bring these examples into the Eclipse workspace and how to launch each of them. After that, each example is going to be discussed in its own section.

6.1 Summary of Examples

There are seven examples to be included in this thesis. Provided below is a list of these examples and some Urml features that each of the examples introduces.

	Example	Desc.	Code
1.	A Simple Handshake	p. 103	p. 168
	• Send signal from one capsule to another		
2.	Consumer-Producer Problem	p. 106	p. 169
	• Pass data through the parameter of a signal		
3.	Widget Production Factory	p. 111	p. 172
	• Provide a “centralized” capsule for synchronization		

	Example	Desc.	Code
4.	Blinking Yellow Light	p. 119	p. 176
	• Control through composite capsules		
5.	Dining Philosophers	p. 125	p. 179
	• Illustrate a real-life example		
6.	Parser Router	p. 135	p. 186
	• Illustrate an example with complicated structure		
7.	Readers-Writers Problem	p. 141	p. 192
	• Illustrate an example using a database		

6.2 Preparing the Examples

6.2.1 Extracting

Unpacking the Examples from the Urml user interface is simple. From the menu, select **File** → **New** → **Example**... A **New Example** wizard dialog (Figure 6.1) appears; from the tree-view of wizards, select **Urml** → **Urml Examples**, and press **Next** and **Finish** to end the wizard. A project named `ca.queensu.mase.urml.example1` should be created and it contains the examples to be discussed in this chapter.

6.2.2 Running

There are two ways to run an Urml example: (1) through the Launch Configuration Dialog and (2) through a Launch Shortcut.

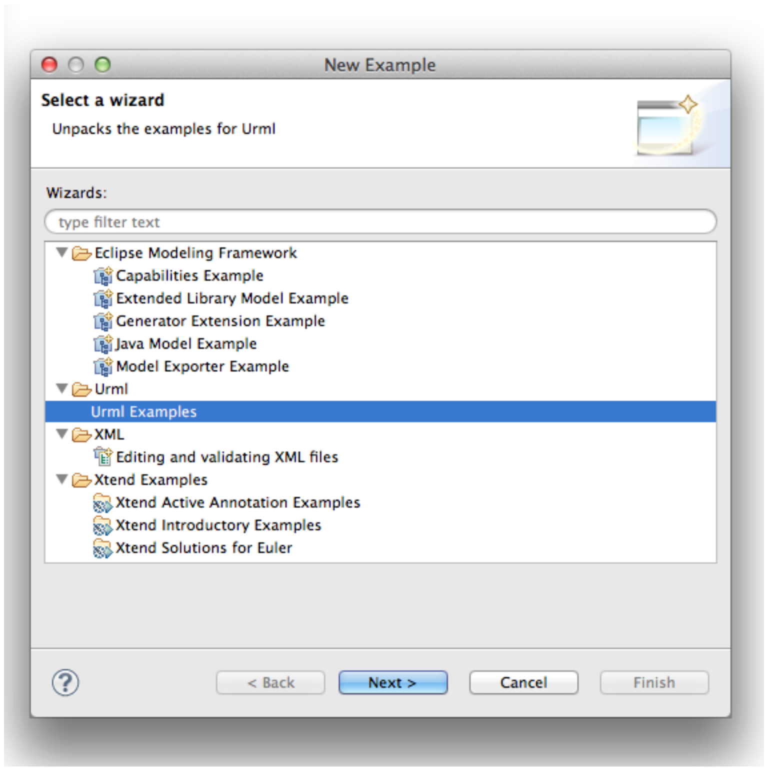

New Example
Select a wizard
Unpacks the examples for Urml
Wizards:
type filter text
Eclipse Modeling Framework
Capabilities Example
Extended Library Model Example
Generator Extension Example
Java Model Example
Model Exporter Example
Urml
Urml Examples
XML
Editing and validating XML files
Xtend Examples
Xtend Active Annotation Examples
Xtend Introductory Examples
Xtend Solutions for Euler
< Back
Next >
Cancel
Finish

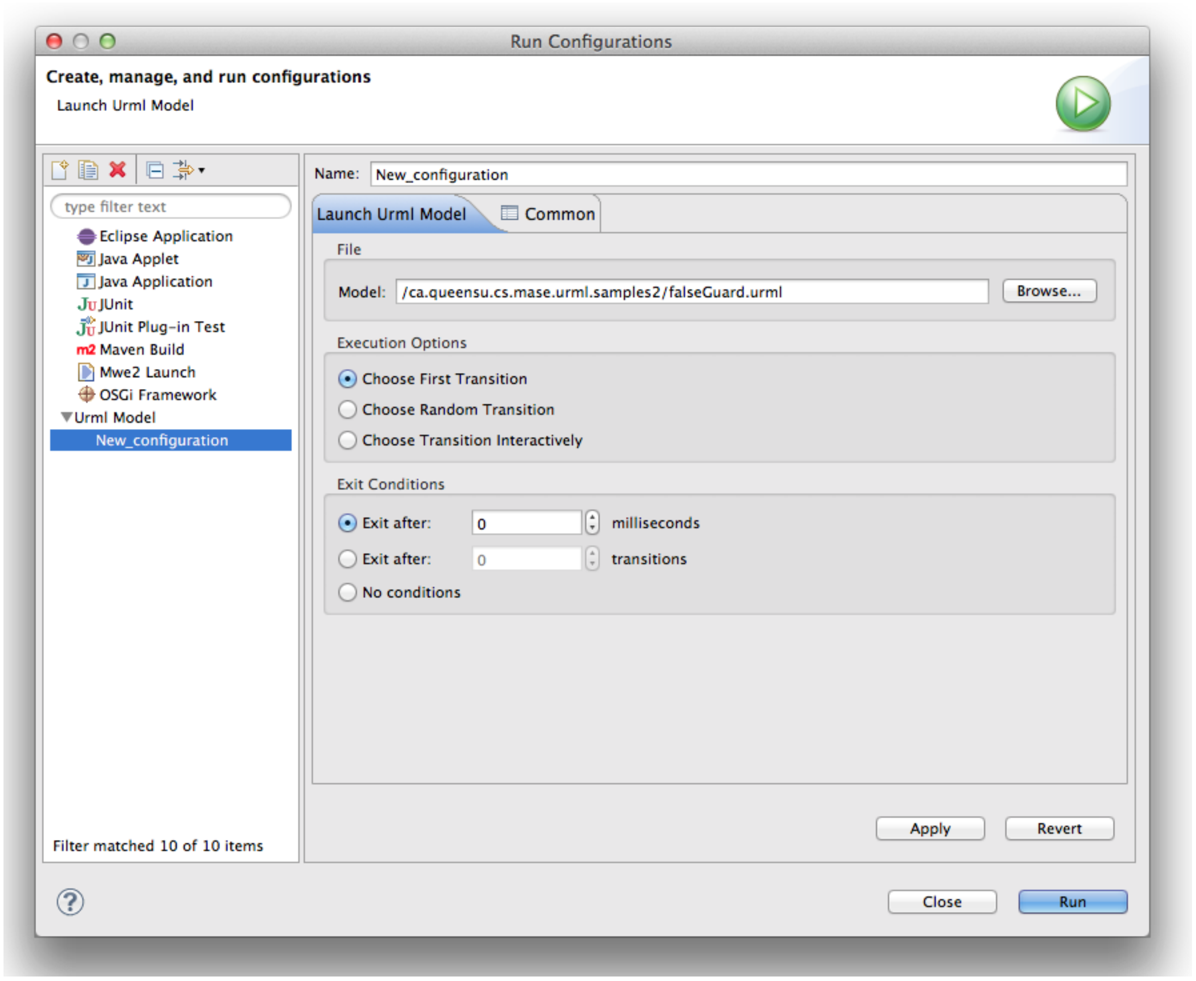
Figure 6.1: A screenshot depicting the New Example Wizard

Through the Launch Configuration Dialog

From the menu, select **Run** → **Run Configurations**... A launch configuration dialog (Figure 6.2) should appear. From the list at the right, right-click on **Urml Model** and select **New**. Then, a page containing a list of available launching options appears, and the user is required to specify three things:

1. **The file to launch.** (Model:) From the Model, browse through the workspace and select the appropriate file to launch.

2. **Execution options for nondeterminism.** (Execution Options:) This option specifies what the interpreter will do if it encounters non-determinism: either (1) select the first transition, (2) select a random transition, or (3) ask the user.



Name: New_configuration

Model: /ca.queensu.cs.mase.urml.samples2/falseGuard.urml

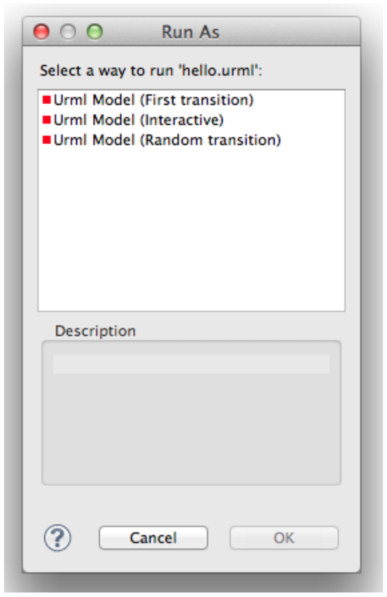
Execution Options: [✓ Choose First Transition] Choose Random Transition Choose Transition Interactively

Exit Conditions: [✓ Exit after: 0 milliseconds] Exit after: 0 transitions No conditions

Figure 6.2: A screenshot depicting the launch configuration dialog

3. **Exit conditions.** (Exit Conditions:) This option decides whether the interpreter should stop after a certain number of seconds or transitions or should run indefinitely.

After the three settings are specified, the **Run** button should be enabled. Press it to launch the example.



Select a way to run 'hello.urml':

- Urml Model (First transition)
- Urml Model (Interactive)
- Urml Model (Random transition)

Figure 6.3: A screenshot depicting the Run As window

Through a Launch Shortcut

From the package explorer, select the example model to launch and double-click on it to open the model. From the menu, select **Run** → **Run**. A **Run As** window (Figure 6.3) appears, which prompts for which execution options for nondeterminism to choose; refer to the above for brief description. Choose the execution option and press **Run** to launch the example.

6.3 Disclaimer: Urml as a Textual Language

Note that although Urml is a textual language, it is possible to express an Urml model graphically. In this chapter, *both* the textual language and the graphical syntax are shown together for the first two examples. However, for the rest of the chapter, only the graphical language is shown for brevity; the interested reader may refer to Appendix B for the source code for each of the examples.

6.4 A Simple Handshake

This is a very simple model (`handshake.urml`, source at Appendix B.1 on page 168), where one capsule sends a signal to another capsule, and then both capsules enter their final state. The protocol, `HandshakeProtocol`, consists of only one kind of outgoing messages:

```
protocol HandshakeProtocol {
  outgoing {
    shake()
  }
}
```

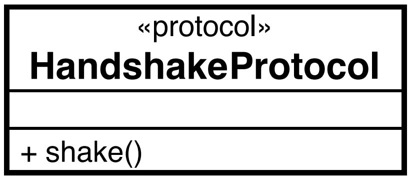


Figure 6.4: Class diagram for `HandshakeProtocol`

The message `shake()` simulates the action of the handshake signal that is being sent by the handshake sender to the handshake receiver.

The structure of the model is very simple. It consists of a handshake originator and a handshake receiver, with each of the capsule containing one `HandshakeProtocol` port called `hand`. The handshake originator contains a base port while the receiver contains a conjugated port. The two ports are connected together. The model is depicted below with logging ports omitted.

```
root capsule Handshake {
  capsuleInstance sender : Originator
  capsuleInstance receiver : Receiver
  connector sender.hand and receiver.hand
}
```

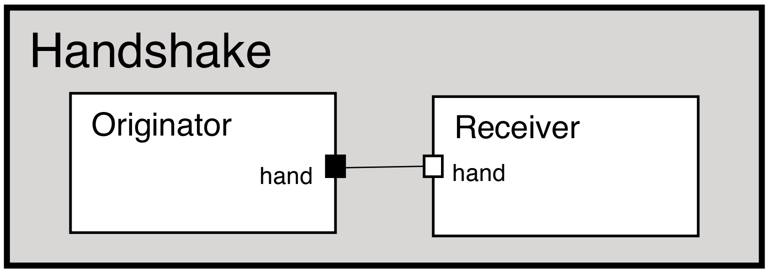


Figure 6.5: Structure diagram for the Handshake model

The behaviour of the model is also very simple. The originator simply sends a `shake()` signal to the `hand` port:

```
state start
final state end
transition init : initial -> start {}
transition doHandShake : start -> end {
  action {
    send hand.shake()
  }
}
```

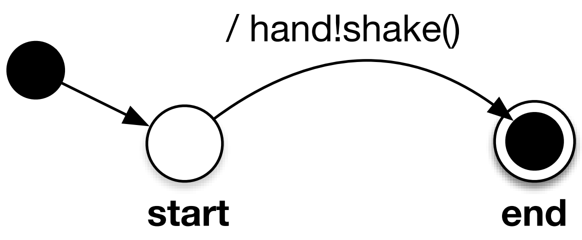


Figure 6.6: State diagram for the handshake originator

The receiver receives a `shake()` signal from its hand port:

```
state start
final state end
transition init : initial -> start {}
transition receiveHandshake : start -> end {
  triggeredBy hand.shake()
}
```

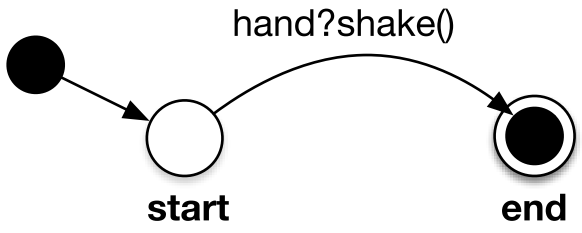


Figure 6.7: State diagram for the handshake receiver

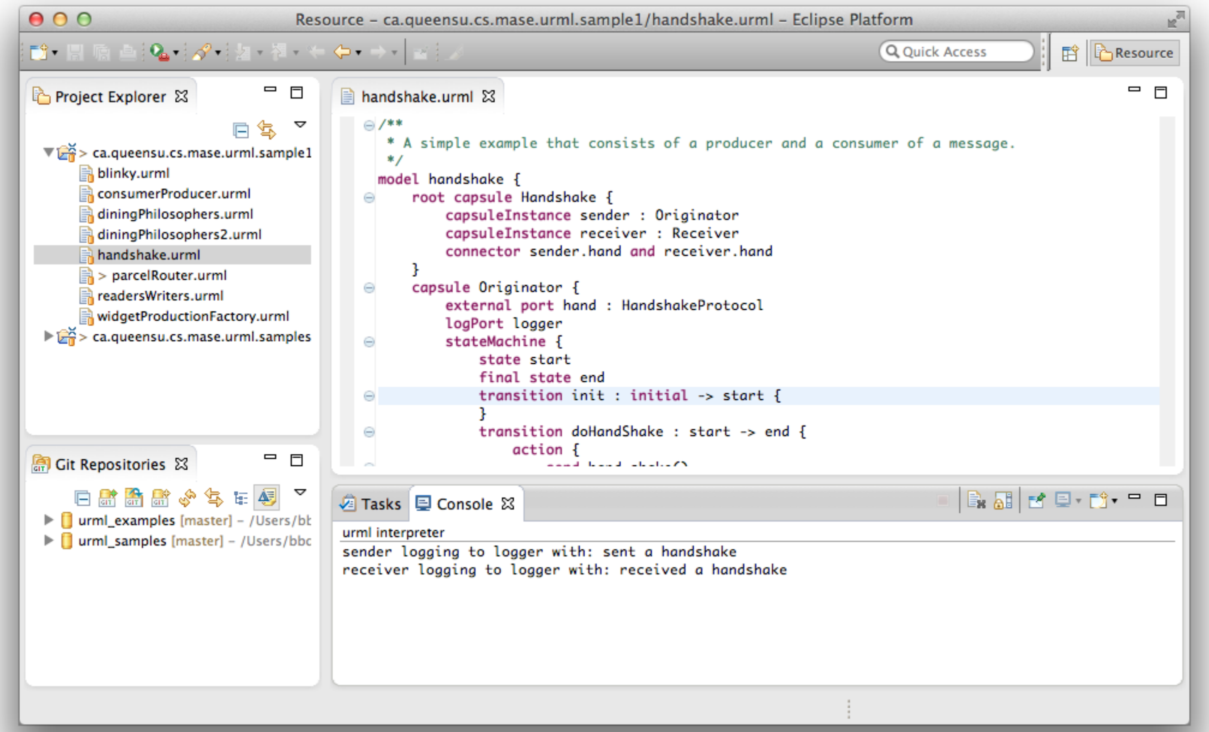
Note that both of the destination states from the two state diagrams are final; thus the model ends running after this handshake has occurred.

As for testing, the following output is displayed by the model program (screenshot shown in Figure 6.9):

```
sender logging to logger with: sent a handshake
receiver logging to logger with: received a handshake
```

Figure 6.8: Output of the handshake model

The sender sent a `shake()` signal and displayed the first line, while the receiver received a `shake()` signal and displayed the second line. This is expected behaviour. After the testing is done, this ends the discussion of the handshake model.


Resource – ca.queensu.cs.mase.urml.sample1/handshake.urml – Eclipse Platform
Quick Access
Resource
Project Explorer
ca.queensu.cs.mase.urml.sample1
blinky.urml
consumerProducer.urml
diningPhilosophers.urml
diningPhilosophers2.urml
handshake.urml
parcelRouter.urml
readersWriters.urml
widgetProductionFactory.urml
ca.queensu.cs.mase.urml.samples
Git Repositories
urml_examples [master] – /Users/bt
urml_samples [master] – /Users/bbc
handshake.urml
Tasks
Console
urml interpreter

```
/**
 * A simple example that consists of a producer and a consumer of a message.
 */
model handshake {
    root capsule Handshake {
        capsuleInstance sender : Originator
        capsuleInstance receiver : Receiver
        connector sender.hand and receiver.hand
    }
    capsule Originator {
        external port hand : HandshakeProtocol
        logPort logger
        stateMachine {
            state start
            final state end
            transition init : initial -> start {
            }
            transition doHandShake : start -> end {
                action {
```

```
sender logging to logger with: sent a handshake
receiver logging to logger with: received a handshake
```

Figure 6.9: A screenshot depicting the execution of the handshake model

6.5 Consumer–Producer Problem

The consumer–producer model (consumerProducer.urml, source at Appendix B.2 on page 169) handles data—producing and consuming them—without modifying the data. The main component of the consumer–producer model is the bounded buffer, which can hold at most 5 integers.

6.5.1 Structure

The textual code for the bounded buffer is as follows:

```
capsule BoundedBuffer {
  attribute a := 0
  attribute b := 0
  attribute c := 0
  attribute d := 0
  attribute e := 0
  external port ~consumer : BufferProtocol
  external port ~producer : BufferProtocol
}
```

Likewise the class diagram for the bounded buffer is as follows:

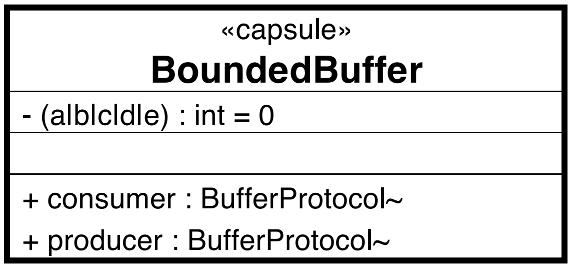


Figure 6.10: Class diagram for the bounded buffer

Its attributes—`a`, `b`, `c`, `d`, and `e`—store as the integer buffer, which are the five integers. The bounded buffer connects to the consumer and the producer through `BufferProtocol` ports. The `BufferProtocol` is shown as follows:

```
protocol BufferProtocol {
  incoming {
    get(data)
  }
  outgoing {
    put(data)
  }
}
```

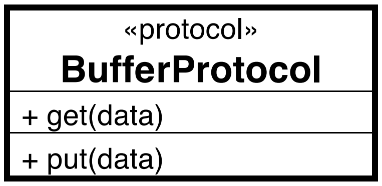


Figure 6.11: Class diagram for the buffer's protocol

The buffer sends `get(data)` signals to the consumer so that the consumer displays the data, and the producer sends `put(data)` signals to the buffer for the buffer to store the data. To put all the pieces together, the overall structure of the consumer-producer model is as follows:

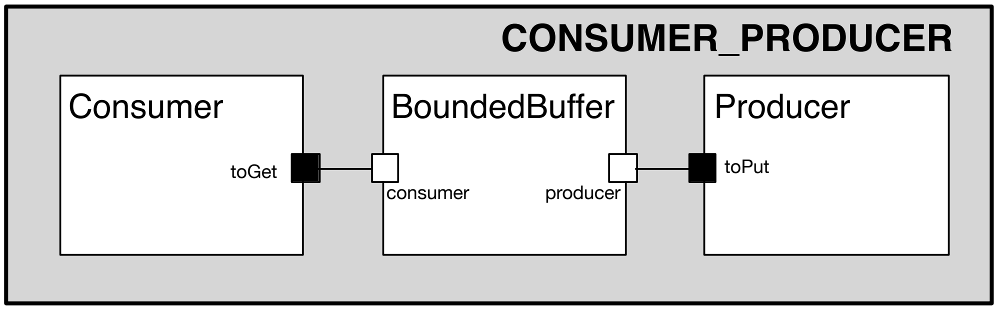


Figure 6.12: Structure diagram for the consumer–producer problem

The corresponding code is as follows:

```
root capsule ConsumerProducer {
  capsuleInstance c : Consumer
  capsuleInstance p : Producer
  capsuleInstance b : BoundedBuffer
  connector c.toGet and b.consumer
  connector p.toPut and b.producer
}
```

6.5.2 Behaviour

The behaviour of the model can be described with the state diagrams of the bounded buffer, consumer, and the producer. The state diagram of the bounded buffer is described first:

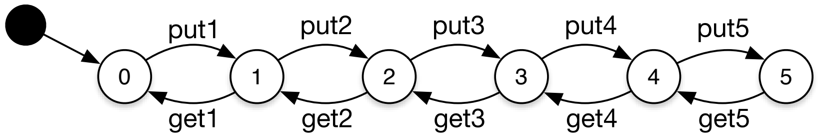


Figure 6.13: State diagram for the bounded buffer

The corresponding code for this state diagram is verbose. Interested reader can find the code in the appendix.

The `put`[12345] transitions are triggered by `producer?get(data)` signals and these transitions store the data value onto the attributes [`abcde`], respectively. The `get`[12345] transitions have no triggers and simply send the signals `consumer!put(`[`abcde`]`)` to the consumer.

The behaviour of the consumer is very simple:

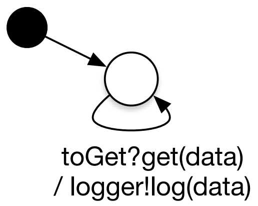


Figure 6.14: State diagram for the consumer

The corresponding source code is as follows:

```
state single
transition : initial -> single {}
transition : single -> single {
  triggeredBy toGet.get(d)
}
```

Whenever it receives a signal `toGet!get(d)`, it displays `d` onto the screen. As for the behaviour of the producer, it is also very simple:

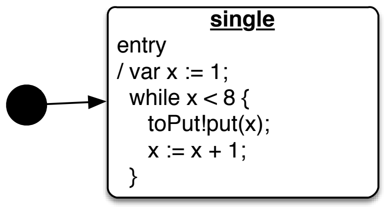


Figure 6.15: State diagram for the producer

The corresponding source code is as follows:

```
state single {
  entry {
    var x:= 1
    while (x < 8) {
      send toPut.put(x)
      x := x + 1
    }
  }
}
```

It starts by sending 7 `toPut!put(x)` signals to the buffer, where `x` is an integer ranging from 1 to 8.

6.5.3 Testing

The output of the model is shown below:

```
NON-DETERMINISM
NON-DETERMINISM
NON-DETERMINISM
NON-DETERMINISM
c logging to logger with: 5
NON-DETERMINISM
c logging to logger with: 6
NON-DETERMINISM
c logging to logger with: 7
c logging to logger with: 4
c logging to logger with: 3
c logging to logger with: 2
c logging to logger with: 1
```

Figure 6.16: Output of the consumer-producer model

The model is non-deterministic as both `put` and `get` signals are enabled at any state.

6.6 Widget Production Factory

The widget production factory model (`widgetProductionFactory.urml`, source at Appendix B.3 on page 172) simulates a factory that manufactures widgets and a robot that consumes the widget by delivering it to somewhere else [22].

6.6.1 Protocol — Actions of the Model

Like the other models, the overall behaviour can be described by the protocols of the model. There are two protocols in the model: the `WorkstationProtocol` and the `RobotProtocol`.

Workstation Protocol

The first protocol is `WorkstationProtocol`, which describes the actions of the widget-producing workstation. Its class diagram is shown below:

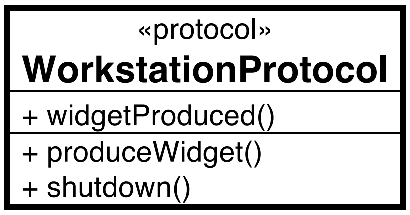


Figure 6.17: Class diagram of the Workstation protocol

The outgoing signal `produceWidget()` commands the workstation to produce widget on demand, and once the workstation has finished making the widget, it fires back the `widgetProduced()` incoming signal to indicate completion. The workstation can also be stopped completely through the `shutdown()` outgoing signal.

Robot Protocol

The second protocol is `RobotProtocol`, which describes the actions of the widget-delivering robot. Its class diagram is shown below:

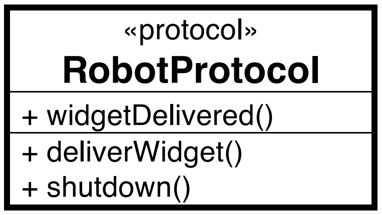


Figure 6.18: Class diagram of the Robot protocol

The protocol is very similar to the `WorkstationProtocol`. The outgoing signal `deliverWidget()` commands the robot to deliver the widget on demand, and once the robot has finished delivering the widget, it fires back the `widgetDelivered()` incoming signal to indicate completion. The robot can also be stopped completely through the `shutdown()` outgoing signal.

6.6.2 Structure

Once the actions of the model is introduced, it is ready to show the overall structure of the model. The top capsule is the `SystemContainer`, which contains an instance of the `ControlSoftware` and an instance of `ProductionLine`. The `ControlSoftware` is the command centre of this whole model while the `ProductionLine` contains the actuators of the model, which are the workstation and the robot. The structure diagram of the `SystemContainer` is depicted below:

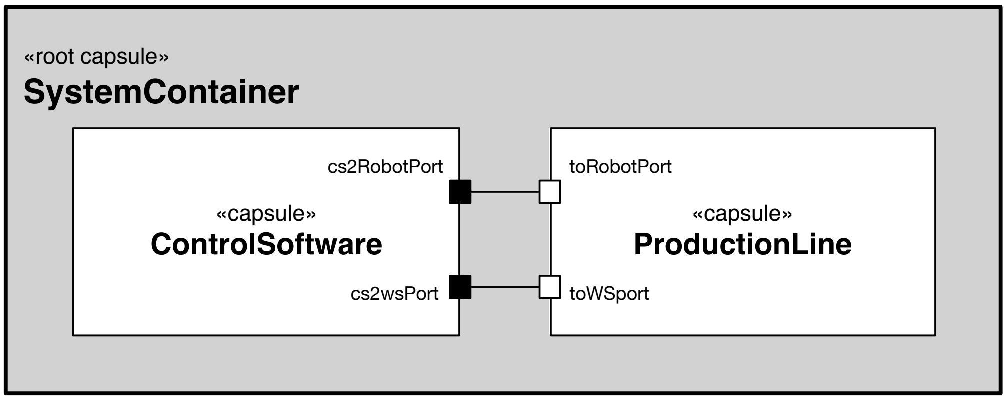


Figure 6.19: Structure diagram of the system container

The structure of the `ProductionLine` is hereby described. The `ProductionLine` contains the two actuating components (1) the workstation and (2) the robot. Its overall structure diagram is shown below:

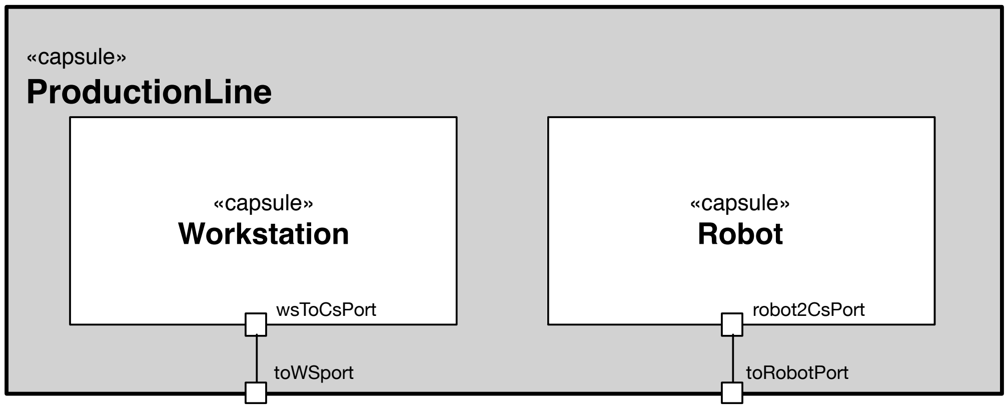


Figure 6.20: Structure diagram of the production line

As mentioned in the protocol section, the workstation is an actuator that manufactures the widget and the robot is an actuator that delivers the widget. Both their actions are controlled by the `ControlSoftware`, which is contained at the top capsule `SystemContainer`.

6.6.3 Attributes

Once the structure of the system is discussed, it is convenient to describe the attributes of each capsule in the model by showing each capsule's class diagram. As described before, there are five capsules in the model: (1) the `SystemContainer`, (2) the `ControlSoftware`, (3) the `ProductionLine`, (4) the `Workstation` and (5) the `Robot`. Disregarding the container capsules, which are the `SystemContainer` and the `ProductionLine`, the class diagrams of the `ControlSoftware`, the `Workstation` and the `Robot` will be shown.

Control System — Attributes

The class diagram of the `ControlSoftware` is shown below:

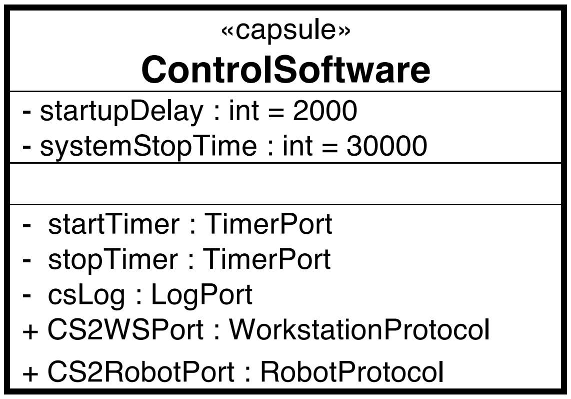

«capsule»
ControlSoftware
- startupDelay : int = 2000
- systemStopTime : int = 30000
- startTimer : TimerPort
- stopTimer : TimerPort
- csLog : LogPort
+ CS2WSPort : WorkstationProtocol
+ CS2RobotPort : RobotProtocol

Figure 6.21: Class diagram of the control software

When the ControlSoftware starts, it waits for two seconds; this can be shown by the startupDelay attribute, which is 2000 milliseconds. The ControlSoftware will use the startTimer to measure this delay. Also, the whole model will run for 30 seconds; this can be seen by the systemStopTime, which is 30000 milliseconds. The ControlSoftware will use the stopTimer to measure when it will stop the whole system.

As can be seen by the ports, the ControlSoftware provides messages to the workstation and the robot through the CS2WSPort and CS2RobotPort, respectively.

Workstation — Attributes

The class diagram of the Workstation is shown below:

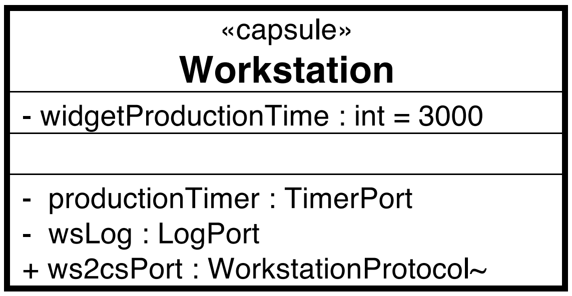

«capsule»
Workstation
- widgetProductionTime : int = 3000
- productionTimer : TimerPort
- wsLog : LogPort
+ ws2csPort : WorkstationProtocol~

Figure 6.22: Class diagram of the workstation

The production time will take 3 seconds; this can be seen in the `widgetProductionTime` attributes, which is 3000 milliseconds. This will be controlled by the `productionTimer`. The workstation receives commands from the `ControlSoftware` through the `ws2csPort`.

Robot — Attributes

The class diagram of the `Robot` is shown below:

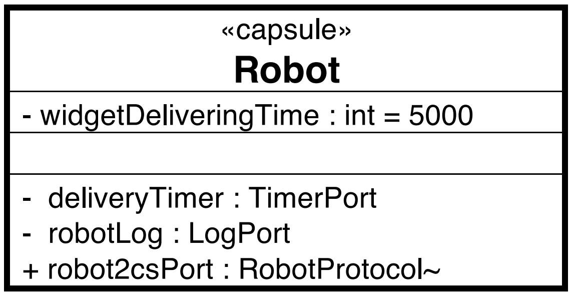

«capsule»
Robot
- widgetDeliveringTime : int = 5000
- deliveryTimer : TimerPort
- robotLog : LogPort
+ robot2csPort : RobotProtocol~

Figure 6.23: Class diagram of the robot

The delivery time will be 5 seconds; this can be seen in the `widgetDeliveringTime` attribute, which is 5000 milliseconds. The delivery time will be controlled by the `deliveryTimer`. The robot receives commands from the `ControlSoftware` through the `robot2csPort`.

6.6.4 Behaviour

Control Software

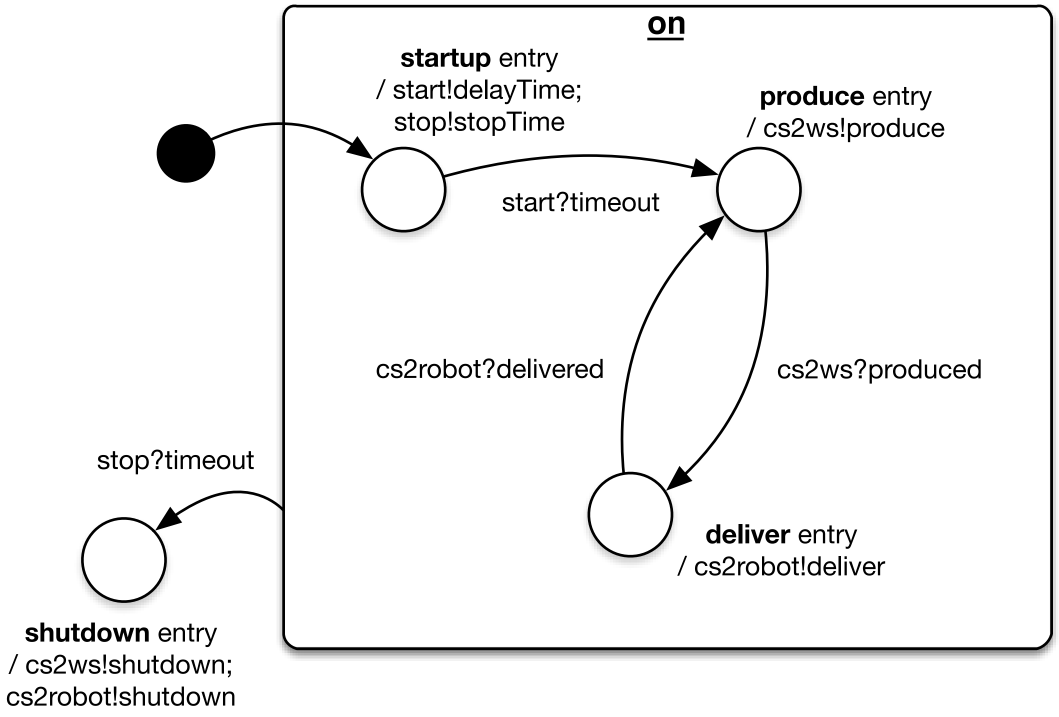

on
startup entry / start!delayTime; stop!stopTime
produce entry / cs2ws!produce
start?timeout
cs2robot?delivered
cs2ws?produced
stop?timeout
deliver entry / cs2robot!deliver
shutdown entry / cs2ws!shutdown; cs2robot!shutdown

Figure 6.24: State diagram of the ControlSoftware

The `ControlSoftware` starts at the `startUp` state, where it will start waiting for 3 seconds and start counting the stop time. Once it stops waiting, it proceeds to the `produce` state. The `ControlSoftware` enters the loop switching between `produce` and `deliver` states.

Once it enters the `produce` state, it immediately fires a `produce()` signal to the workstation to produce widgets. When the workstation signals back to the `Control-Software` that it has finished producing widget through the `produced()` signal, it enters the `deliver` state; entering the `deliver` state triggers the state machine to fire a `deliver()` signal to the robot. When the robot finished delivering as triggered through the `delivered()` signal, it goes back to the `produce` state. The `ControlSoftware` then continues looping between the `produce` and `deliver` states until it is signalled to stop.

Since `startup`, `produce`, and `deliver` states are all grouped into the `on` state, they will all go to the `shutdown` final state when the stop timer expires. If that happens, the `ControlSoftware` signals the workstation and the robot to shut down through the `shutdown()` signal as well.

Workstation

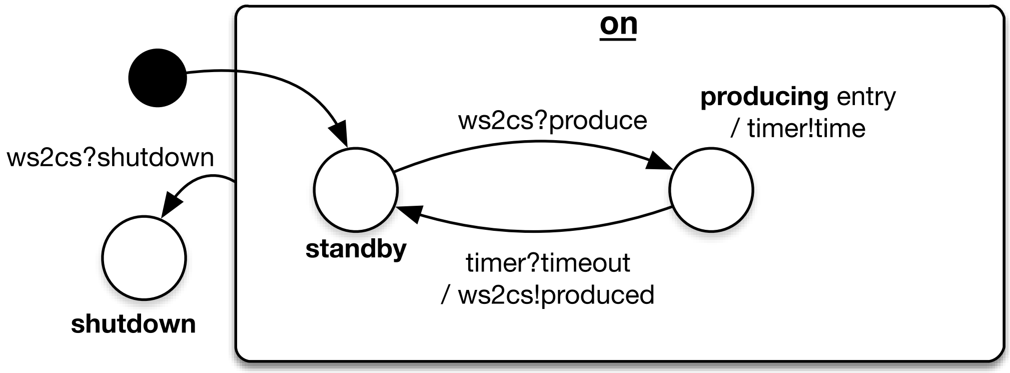


Figure 6.25: State diagram of the Workstation

The `Workstation` starts at the `standby` state.

When it is signalled by the `ControlSoftware` to produce widgets through the `produce()` signal, it transitions to the `producing` state. When it enters the `producing` state, producing widgets, the `Workstation` makes its timer start; once the timer expires, the workstation finishes producing widgets, as signalled by the expiration of the timer. The `Workstation` then transitions back to the `standby` state. The `Workstation` continues looping between the `standby` and `producing` states until it is asked by the `ControlSoftware` to shut down through the `shutdown()` signal.

If triggered by the `shutdown()` signal, the `Workstation` then moves to the `shutdown` final state.

Robot

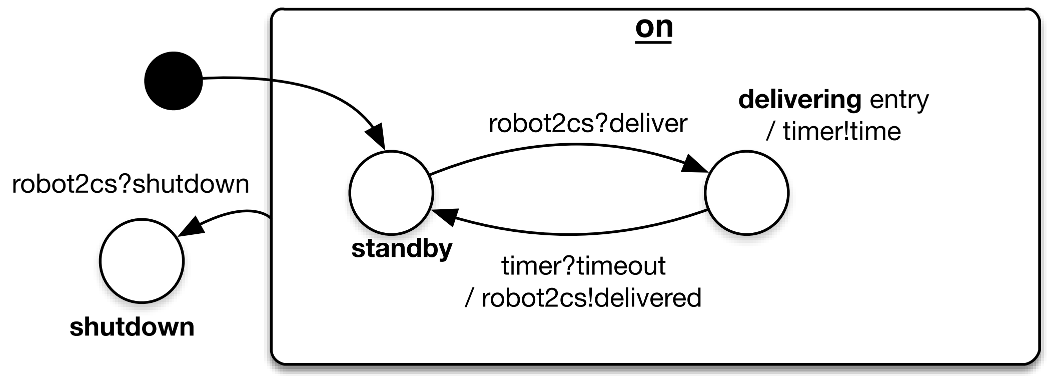

on
robot2cs?deliver
delivering entry / timer!time
robot2cs?shutdown
standby
timer?timeout / robot2cs!delivered
shutdown

Figure 6.26: State diagram of the Robot

The `Robot`'s state machine is very similar to the `Workstation`'s state machine, except that the `produce()` signal is substituted with `deliver()` signal, the `producing` state is replaced with `delivering` state, and the timer is named differently. Otherwise, the behaviour is exactly the same as that of workstation.

6.7 Blinking Yellow Light

This model (`blinky.urml`, source at Appendix B.4 on page 176) simulates a yellow light that blinks [17]. There are two units of control for this blinking light: (1) the blinking part, which turns on the light for 1 second then off for another 1 second, and (2) the cycle part, in which the light blinks for 5 seances and then turns off completely for 5 seconds.

There are three capsules in the model: (1) the blinking light, which turns on and off for 1 second; (2) the controller, which sets the light to blink for 5 seconds and off for 5 seconds, and (3) the top capsule, which contains instances of the blinking light and the controller.

6.7.1 Capsules

A description of the blinking light and the controller capsules are described first, to describe the structural components of the model.

Blinking Light

The blinking light contains the following:

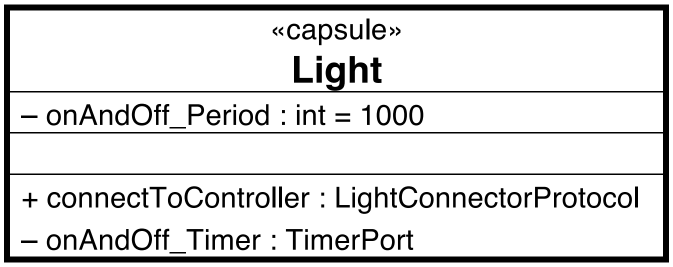

«capsule»
Light
– onAndOff_Period : int = 1000
+ connectToController : LightConnectorProtocol
– onAndOff_Timer : TimerPort

Figure 6.27: Class diagram of the light

The **onAndOff_Period** is an attribute that determines how many milliseconds to turn on or off the light, **onAndOff_Timer** is a timer port to count the time for **onAndOff_Period** milliseconds. The **connectToController** port communicates to the controller for the light.

Controller

The controller has similar contents as the blinking light. Its class diagram is shown below:

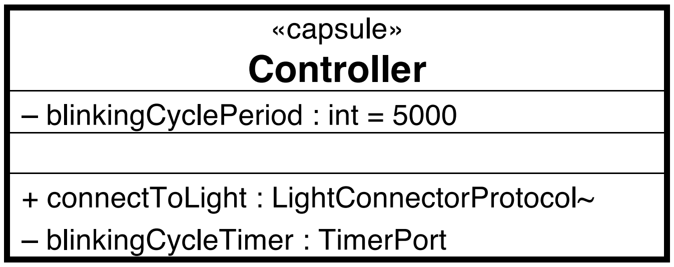

«capsule»
Controller
– blinkingCyclePeriod : int = 5000
+ connectToLight : LightConnectorProtocol~
– blinkingCycleTimer : TimerPort

Figure 6.28: Class diagram of the controller

The **blinkingCyclePeriod** is an attribute that determines how many milliseconds to let the light blink, **blinkingCycleTimer** is the timer that count the time for **blinkingCyclePeriod** milliseconds. The **connectToLight** port communicates to the light for the controller.

Overall structure

The relationship between the light and the controller is summarized by showing the following structure diagram for the top capsule:

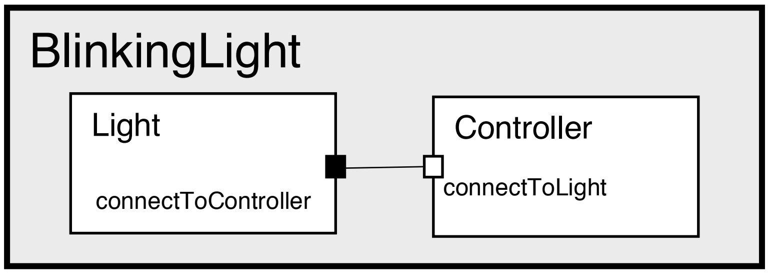


Figure 6.29: Structure diagram for the blinking light model

Note that the timer ports are omitted from the structural diagram for brevity. This completes the discussion of the structure of the model.

6.7.2 Protocol

The behaviour of the model is briefly discussed by describing the protocol between the light and the controller: the `LightController` protocol. The protocol has the following class diagram:

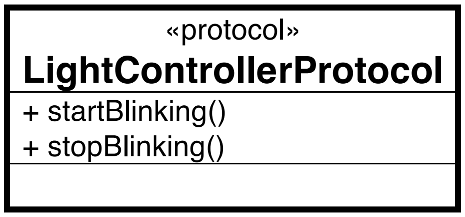


Figure 6.30: Class diagram of the LightController protocol

The `start()` incoming signal comes from the controller to the light so that the light starts blinking for 5 seconds, and the `stop()` incoming signal comes from the controller to the light so that the light stops blinking for 5 seconds.

6.7.3 Behaviour

The state machines of the light and the controller describes the behaviour of the model in more detail. Below is the state machine of the blinking light:

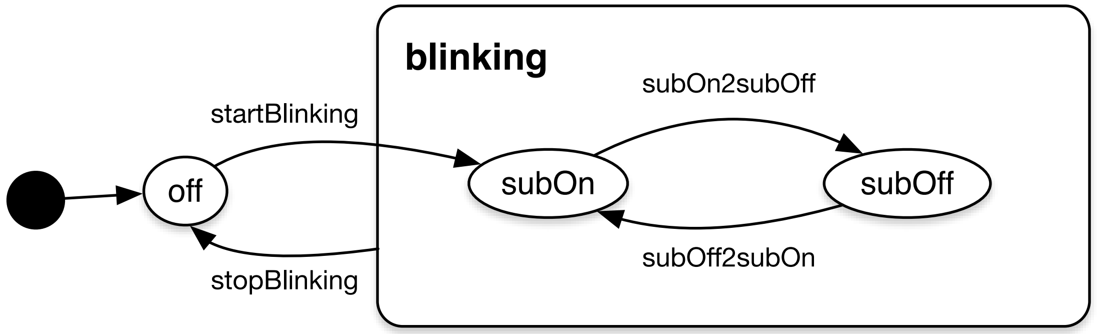


Figure 6.31: State diagram of the light

Except the initial transitions, all the other transitions are triggered by timeout events from the timers. The `startBlinking` and `stopBlinking` transitions are triggered by the timer from the controller, which dictates the light to blink for 5 seconds and to not blink for 5 seconds. These triggers are sent through the `toController` conjugated port with the signals `start()` and `stop()`, respectively.

The `subOn2subOff` and `subOff2subOn` transitions are triggered by the timer from the light, which dictates the light to turn on for 1 second and off for another second while the light is in blinking state. These triggers are *not* sent through communication port because the timer is already contained by the light.

On the other hand, the state machine for the controller is shown below:

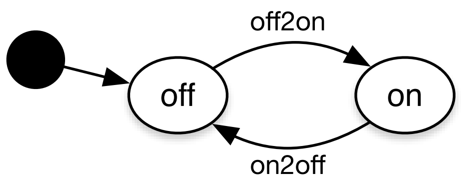


Figure 6.32: State diagram of the controller

This state machine is much simpler than the one from the light. Initially, the controller starts the timer for 5 seconds and then begins at `off` state; the light is not blinking now. After 5 seconds, the timer triggers the controller to start the `off2on` transition, which fires a `startBlinking()` signal to the light and restarts the timer again for 5 seconds; the controller is in the `on` state and the light is blinking when the transition ends. After 5 seconds, the timer times out and triggers the controller to start the `on2off` transition, which fires a `stopBlinking()` signal to the light and restarts the timer again for 5 seconds; the controller ends up in the `off` state and the light stops blinking when the transition ends. This cycle goes on forever until the user interrupts the simulation.

6.7.4 Testing

The result of the model is difficult to describe on paper because it involves the progress of time. Regardless, once the model is launched in any of the three settings (there is no non-determinism in the model, and thus any one configuration will give the same result), the following message is displayed,

```
controller logging to logger with: connect to light start
blinky logging to logger with: Lights turn to yellow for 1 seconds...
```

and the following message is displayed after 1 second,

```
blinky logging to logger with: Lights turn off for 1 seconds
```

Each line of the following messages is displayed for 1 second,

```
blinky logging to logger with: Lights turn to yellow for 1 seconds...
blinky logging to logger with: Lights turn off for 1 seconds
blinky logging to logger with: Lights turn to yellow for 1 seconds...
```

Then, the following message appears, indicating that a `stopBlinking()` signal is fired,

```
controller logging to logger with: Blinky stops blinking for 5 seconds
```

Then, no message appears, until after 5 seconds when the following message occur:

```
controller logging to logger with: Blinky blinks for 5 seconds
blinky logging to logger with: Lights turn to yellow for 1 seconds...
```

The cycle goes on until the user interrupts the simulation.

This is the expected result of the model, where the light turns on and off (i.e. blinking) for 1 second, and stops blinking at all for 5 seconds, and then continue turning on and off for 1 second, and so on.

6.8 Dining Philosophers

The dining philosophers problem is commonly used for describing different processes sharing limited resources, which can result to deadlocks in concurrent operating systems. Three philosophers sit with a bowl of food around a circular table and spend their lives eating and thinking without communicating with each other. Between each philosopher there is a fork. To be able to eat, a philosopher must use two forks that are shared with one philosopher on either side.

Urml provides an example of a scheme for this problem that demonstrates how non-determinism can occur, particularly when one philosopher has two possible forks to pick up (`diningPhilosophers.urml`, source at Appendix B.5 on page 179).

6.8.1 Protocol

Since this example only concerns of the interaction between philosophers and forks, the only protocol needed here is the PhilosopherForkProtocol. The base side of the protocol is intended for a philosopher; the conjugate side is for a fork. The class diagram for PhilosopherForkProtocol is described below:

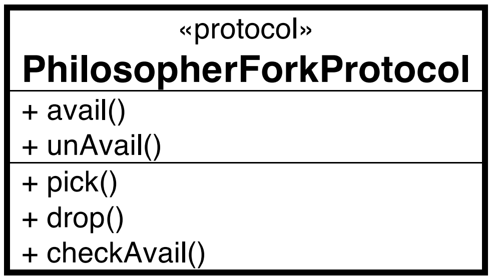


Figure 6.33: Class diagram of the philosopher–fork protocol

Incoming signals are triggered from a fork to a philosopher to indicate whether the fork is available to be picked up. avail() is triggered when the signalling fork is available; notAvail() is triggered when the fork is not unavailable.

Outgoing signals simulate how a philosopher interacts with a fork. The pick() signal is triggered by a philosopher when the signalling philosopher is picking up the fork in order to eat. The drop() signal is triggered when the signalling philosopher is putting down the fork in order to think. When a philosopher wishes to pick up a fork, the philosopher uses the signal checkAvail() to verify if the target fork is available to pick up.

6.8.2 Capsules

The whole model consists of three capsules: (1) the Philosopher capsule, (2) the Fork capsule, and (3) the Top capsule that puts all the Philosopher and Fork instances together.

The Philosopher capsule

The contents of a philosopher is shown in the following class diagram:

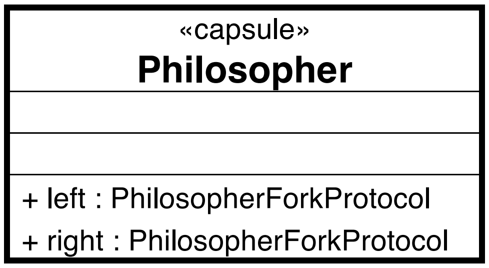

«capsule»
Philosopher
+ left : PhilosopherForkProtocol
+ right : PhilosopherForkProtocol

Figure 6.34: Class diagram of the philosopher capsule

A philosopher contains two external ports—left and right. These ports are used to communicate to forks, which allows the philosopher to verify if the forks are available, and to pick up or drop their forks. In summary, its structure is shown below:

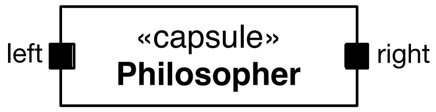


Figure 6.35: Structure diagram of the philosopher capsule

The Fork capsule

Structured similarly to the Philosopher capsule, the Fork capsule has the following class diagram:

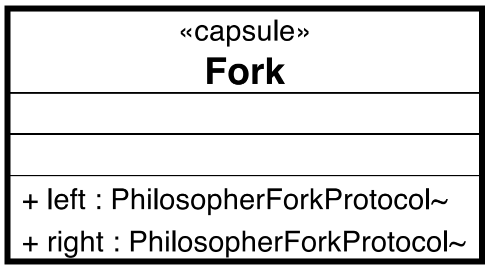

«capsule»
Fork
+ left : PhilosopherForkProtocol~
+ right : PhilosopherForkProtocol~

Figure 6.36: Class diagram of the fork capsule

The left and right ports allow the fork to message to a philosopher whether it is available to pick up. In summary, structure diagram of a Fork is shown below:

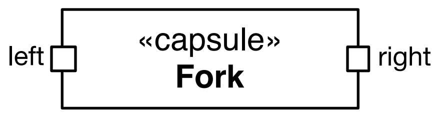


Figure 6.37: Structure diagram of the fork capsule

The Top capsule

The Top capsule puts all instances of philosophers and capsules together, with the connections among them realized. To review, its structure diagram is shown below:

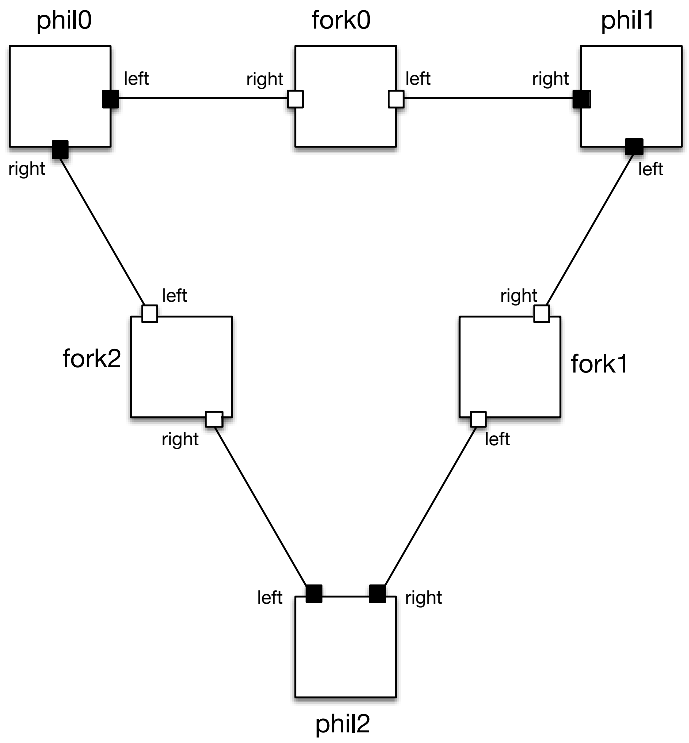


Figure 6.38: Structure diagram of the top capsule for the dining philosopher model

The top capsule contains 3 instances of philosophers and 3 instances of forks. Each philosopher's right port is connected to a fork's left port. Likewise, each fork's right port is connected to a philosopher's left port. This completes the discussion of the dining philosopher's structure.

6.8.3 Behaviour

The state machines of a philosopher and a fork describe the behaviour of the overall model. The state machine diagram for a philosopher is shown below:

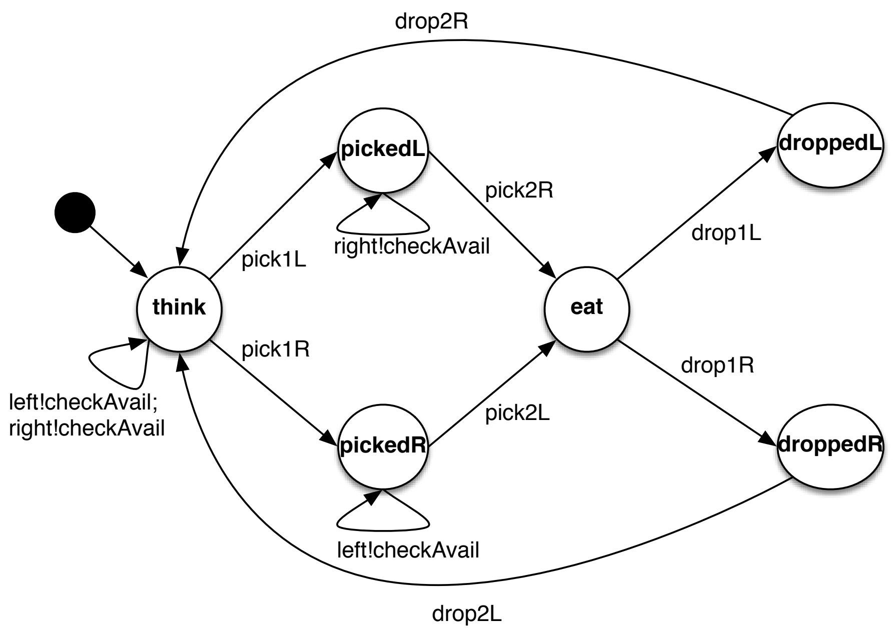


Figure 6.39: State diagram for a philosopher capsule

The philosopher first begins to think. While it is thinking, it sends checkAvail() signals to its left and right forks to determine their availability. If it receives an avail() signal from its left (i.e., the left fork is available), it processes the pickL1 transition (i.e., the philosopher picks up its left fork first) and proceeds to the pickedL state; likewise, if it receives an avail() signal from its right (i.e. the right fork is available), it processes the pickR1 transition (i.e. the philosopher picks up its right fork first) and process to the pickedR state. Note the non-determinism at the think state: the philosopher may processes either pickL1 transition or pickR1 transition if both forks are available. If the user is in interactive mode, the interpreter may ask which transition to execute.

Once the philosopher enters the pickedL state, it sends checkAvail() signal to its right to determine the availability of the right fork. If it receives an avail() signal from its right (i.e., the right fork is available), it processes the pickR2 transition (i.e., the philosopher picks up its right fork second) and proceeds to eat.

The pickedR state is similar to the pickedR state; once the philosopher enters the pickedR state, it sends a checkAvail() signal it its left to determine the availability of the left fork. If it receives an avail() signal from its left (i.e. the left fork is available), it processes the picikL2 transition (i.e., the philosopher picks up its left fork second) and proceeds to eat.

Once the philosopher stops eating, the philosopher approaches non-determinism again: both drop1L and drop1R transitions are outgoing from the eat state (i.e. the philosopher can drop either its left or right fork). If drop1L is chosen, it proceeds to droppedL state; likewise, if drop1R is chosen, it proceeds to droppedR state.

The droppedL and droppedR states have no triggers and thus the next transition will be processed next. Here, the philosopher will drop the remaining fork (either left, which processes the drop2L transition, or right, which processes the drop2R transition), and goes back to think. This completes the description of a philosopher's behaviour during its think-pick-eat-drop cycle.

The state diagram for a fork is shown as follows:

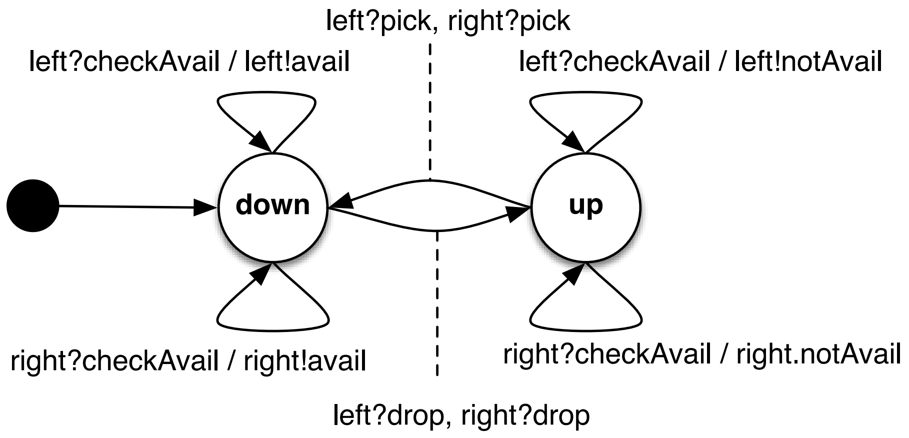

left?pick, right?pick
left?checkAvail / left!avail
left?checkAvail / left!notAvail
down
up
right?checkAvail / right!avail
right?checkAvail / right.notAvail
left?drop, right?drop

Figure 6.40: State diagram for a fork capsule

The fork is at either two state: down or up.

The fork starts at "down" state. When a checkAvail() signal is sent from the left (i.e., the philosopher from the left checks for the fork's availability), it sends an avail() signal to its left (i.e., the fork tells the philosopher that it is available); likewise, when a checkAvail() signal is sent from the right (i.e. the philosopher from the right checks for the fork's availability), it sends an avail() signal to its right (i.e. the fork tells the philosopher that it is available). If it receives a pick() signal from its left or right, it goes to the "up" state (i.e. the fork is picked up).

The fork then proceeds to its "up" state. The fork is unavailable at this time, so when a checkAvail() signal is sent from either side, it sends an notAvail() signal back to that side, indicating its unavailability. Once it receives a drop() signal from its philosopher from its left or right, it then goes back to the down state (i.e., the fork is put down).

This completes the description of the fork's behaviour during its up-down cycle.

6.8.4 Testing

The original purpose of the dining philosopher problem is to demonstrate how processes sharing limited resources can deadlock. Thus, without using an algorithm to distribute the forks to the philosophers, the model is likely to end up in a deadlock situation. The model provided in the example does not use any special algorithm, and so deadlock is expected. Referring back to the state diagram of the philosophers, it is very likely to end up having two possible transitions to execute at the same time, and so non-determinism is expected.

The interpreter is slightly edited in which the message "NON-DETERMINISM" is displayed when there is more than one enabled transition occurring at the same time. Using the configuration that chooses the first transition during non-determinism occurs, the output for the model ends up as shown in Figure 6.41.

The process deadlocks afterwards. This is indeed expected behaviour for the model.

The "NON-DETERMINISM" message is displayed while Philosopher 0 finishes eating and is deciding whether to drop its left or right fork. Non-determinism occurs again when Philosopher 1 encounters the same situation. This is expected behaviour as well.

Not non-deterministic enough?

One might expect the model to be non-deterministic when it is actually not so. Non-determinism might occur for instance when the philosophers are deciding which forks to put up during the first around. However, this did not occur. Instead, the philosophers "line up" amongst themselves to ask for the availability of forks.

```
phil0 logging to logger with: think
phil1 logging to logger with: think
phil2 logging to logger with: think
fork0 logging to logger with: down
fork1 logging to logger with: down
fork2 logging to logger with: down
fork0 logging to logger with: sendAvailR
fork1 logging to logger with: sendAvailR
fork2 logging to logger with: sendAvailL
phil0 logging to logger with: pick1L
phil0 logging to logger with: pickedL
phil1 logging to logger with: pick1L
phil1 logging to logger with: pickedL
fork0 logging to logger with: sendAvailL
fork1 logging to logger with: sendAvailL
fork2 logging to logger with: sendAvailR
phil0 logging to logger with: pick2R
phil0 logging to logger with: eat
phil1 logging to logger with: pick2R
phil1 logging to logger with: eat
phil2 logging to logger with: pick1R
phil2 logging to logger with: pickedR
fork0 logging to logger with: up
fork1 logging to logger with: up
fork2 logging to logger with: sendAvailL
NON-DETERMINISM
phil0 logging to logger with: drop1L
phil0 logging to logger with: droppedLeft
NON-DETERMINISM
phil1 logging to logger with: drop1L
phil1 logging to logger with: droppedLeft
phil2 logging to logger with: pick2L
phil2 logging to logger with: eat
fork0 logging to logger with: sendNotAvailL
fork2 logging to logger with: up
phil0 logging to logger with: drop2R
phil0 logging to logger with: think
phil1 logging to logger with: drop2R
phil1 logging to logger with: think
NON-DETERMINISM
phil2 logging to logger with: drop1L
phil2 logging to logger with: droppedLeft
fork2 logging to logger with: sendNotAvailR
phil0 logging to logger with: pick1R
phil0 logging to logger with: pickedR
phil1 logging to logger with: reask1r
phil2 logging to logger with: drop2R
phil2 logging to logger with: think
phil2 logging to logger with: reask1L
```

Figure 6.41: Output for the dining philosopher model

Consider the following trace. First, phil0 asks fork0 and fork2 for availability, and so the forks' message queues are:

fork0.msgQ: [phil0]

fork2.msgQ: [phil0]

Now, phil1 asks for fork0 and fork1 for availability; as the result, the message queues are:

fork0.msgQ: [phil0, phil1]

fork1.msgQ: [phil1]

fork2.msgQ: [phil0]

and this goes on until we finish a cycle. When we finish the cycle, we have the following message queues:

fork0.msgQ: [phil0, phil1]

fork1.msgQ: [phil1, phil2]

fork2.msgQ: [phil0, phil2]

When the forks are trying to send back their status to the philosophers, the ideal model should be non-deterministic. That is, fork0 does not know whether to return its status to phil0 or phil1, and fork1 does not know whether to return its status to phil1 or phil2, and so on. However, in the implemented model, we are more biased towards sending the status to the philosopher with a smaller index for they start lining up at the forks' message queues earlier.

In this case then, fork0 will send its avail() status to phil0, fork1 to phil1, and fork2 to phil0. After this cycle, the message queues for the philosophers are:

phil0.msgQ: [fork0, fork2]

phil1.msgQ: [fork1]

The ideal model expects non-determinism for phil0 between picking up fork0 or fork2, but it does not in the implemented model. It can only see fork0's message while fork2's message is behind the queue. So, the implemented model is biased towards picking up the fork with the smaller index because it starts lining up at the philosophers' message queues earlier.

This is due to the nature of Urml's computational model, which involves the use of message queues, which employs a first-in-first-out implementation. The queue, which is part of the computation model, add the restriction where the element that pops out is always the element that lines up at the queue earlier, and therefore eliminates certain non-deterministic choices. This is, therefore, expected behaviour.

6.9 Parcel Router

This example [13] deals with a parcel-sorting device, of which the structure is shown below (`parcelRouter.urml`, source at Appendix B.6 on page 186):

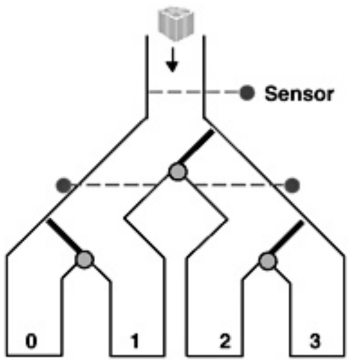


Figure 6.42: Schematic for a parcel router

A parcel is dropped from the top of the router and falls through the chutes. Each parcel has a destination identifier that can be read by the sensors. When a parcel passes a sensor, the sensor reads the destination code and the switch following the sensor is set to send the parcel to the correct destination bin.

As for the destination identifier, the code can be of any non-negative integer, but the router will route the parcel to the destination bin whose number is the parcel's identifier modulo 4; therefore, there are only four destinations numbered from zero to three.

In the following sections, a timed model for the parcel router is developed and its simulation will be discussed. Gravity and friction are ignored; it is assumed that the parcels are falling at constant speed through the chutes and switches.

6.9.1 Overall Structure

The overall structure of the parcel router is first discussed. The structure diagram of the top capsule is shown as follows, noting that timer ports and log ports are all omitted:

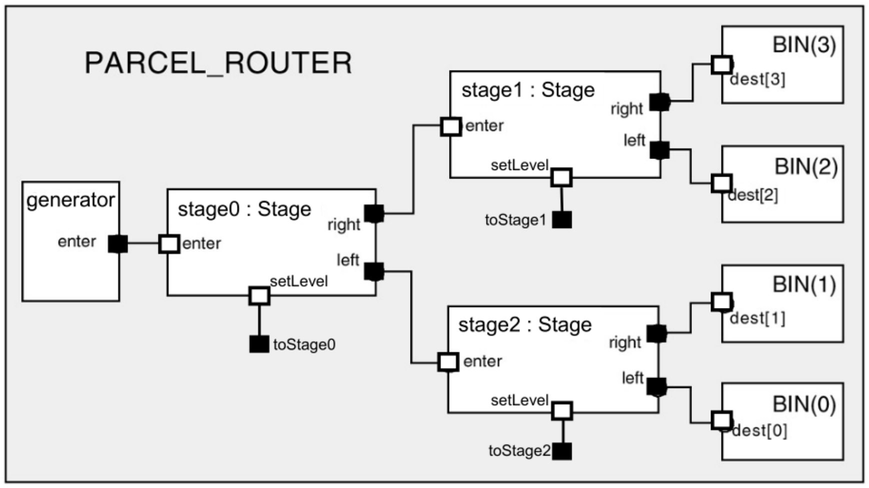


Figure 6.43: Structure diagram for the top capsule of the parcel router

The parcel is first made by the generator capsule, and then is fed into the routing network. The parcel eventually emerges from the network to one of the four destination bins.

Note that each stage has a setLevel port, which is connected to a setLevel port from the parcel router. This port is used for the top capsule to communicate to the stage which level the stage is at. The level number will be useful to calculate which destination bin the parcel is sent to. The following state diagram of the parcel router indicates that the level of the stages are set upon the parcel router starts.

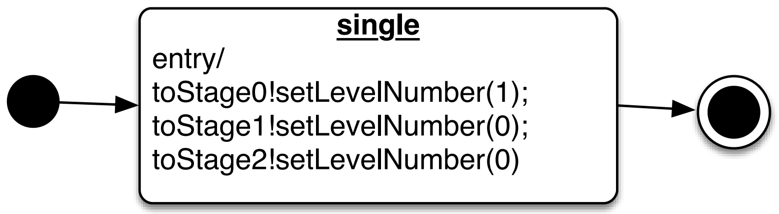

single
entry/
toStage0!setLevelNumber(1);
toStage1!setLevelNumber(0);
toStage2!setLevelNumber(0)

Figure 6.44: State diagram for the top capsule of the parcel router

6.9.2 Protocols

After one is familiar with the overall structure of the model, perhaps it is more convenient to discuss the three communication protocols. First, the passing of each parcel is simulated through the ParcelPassage protocol, whose class diagram is shown as follows:

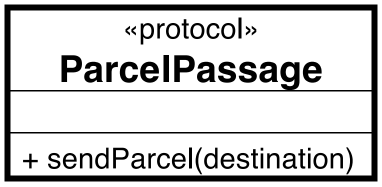


Figure 6.45: Class diagram for the ParcelPassage protocol

Second, the sensor of each stage is required to signal which side (left or right) of the passage the switch should send its parcel to. The protocol of such signals is as follows:

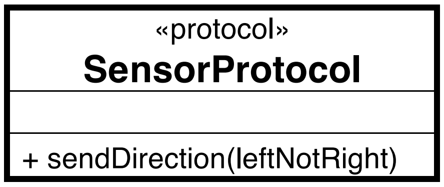


Figure 6.46: Class diagram for the SensorProtocol

Third, the parcel router must somehow send the level number to each of the stages. The protocol for doing this is shown as follows:

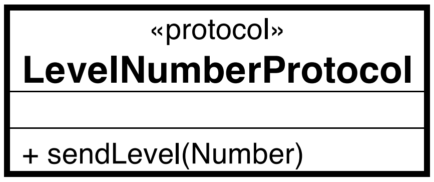


Figure 6.47: Class diagram for the LevelNumberProtocol

Thus, there are three forms of communication in the parcel router, each with a different purpose: (1) to simulate the parcel passing, (2) to provide the direction to send the parcel, and (3) to send the level number of each stage.

6.9.3 Generator — Structure and Behaviour

Now, we should discuss each capsule of the model separately. Refer back to the structure diagram of the parcel router, and we can see that the generator capsule has the following:

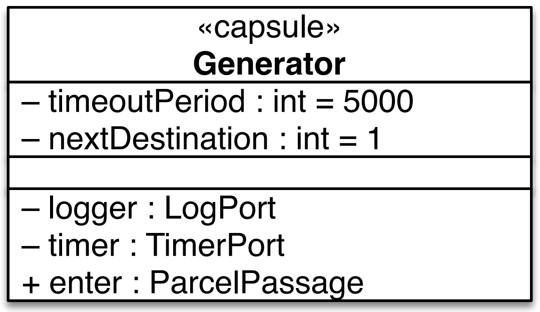


Figure 6.48: Class diagram for the generator

Its behaviour is simple, as shown in the following state machine diagram:

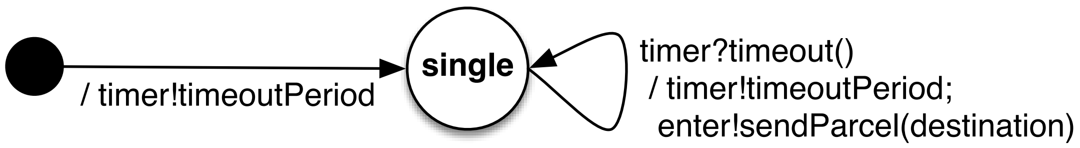


Figure 6.49: State diagram for the generator

Initially, set the timer on for timeoutPeriod amount of time. Then once the timer goes off, set the timer again for timeoutPeriod amount of time, then send the parcel with the nextDestination identifier, and then increment the nextDestination identifier. Continue in this single state until the timer goes off again, and so on.

6.9.4 Destination Bin — Structure and Behaviour

The bin also has a simple structure. Refer back to the structure diagram of the parcel router, and it has the following components:

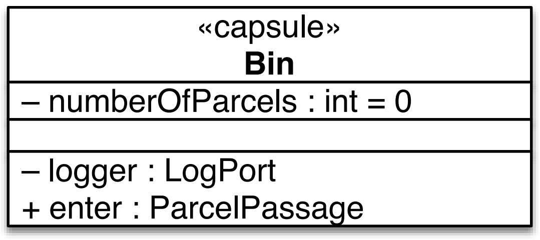


Figure 6.50: Class diagram for the destination bin

Its behaviour is simple; the bin has the following state machine diagram:

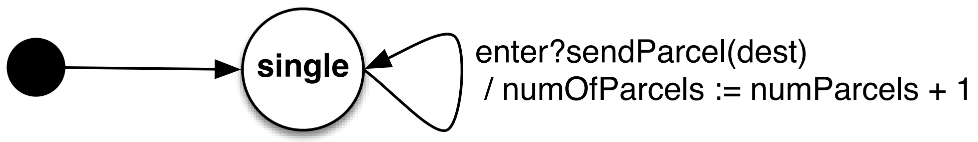


Figure 6.51: State diagram for the destination bin

Where sending is triggered by enter?sendParcel(destination). The bin is initially at the single state. When it is triggered by the sendParcel(destination) signal from the enter port, it increments its numOfParcels attribute.

6.9.5 Stage — Structure

The overall routing network is divided into three stage capsule instances, with each stage having the same structure and behaviour. The structure of each stage composite capsule is depicted as follows:

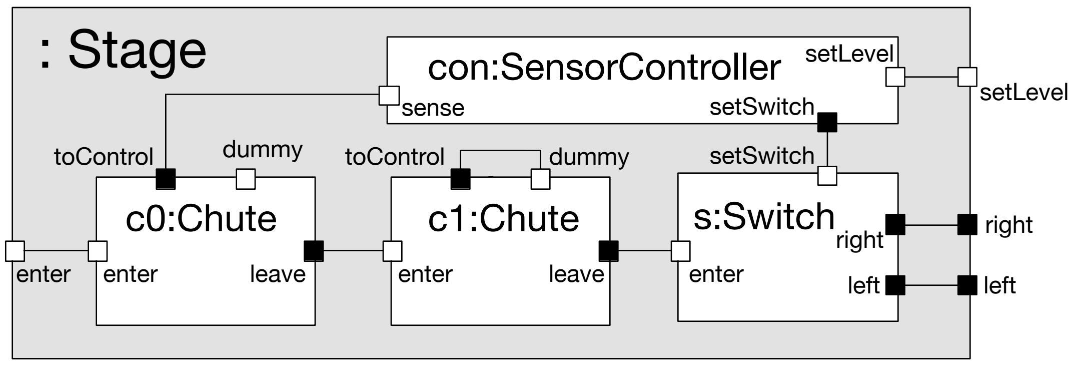


Figure 6.52: Structure diagram for the stage

Each stage has a setLevel port, which sets the level number of that stage (from the top capsule); this level number is then relayed to the sensor controller through the port setLevel.

6.9.6 Sample Output

A sample output using the first-transition setting is shown in Figure 6.53. Here the generator creates a parcel every 10 seconds and the parcel takes 1 second to go through a chute. The execution is stopped when the first parcel has entered the bin. Note that the parcel is eventually sent to bin 1, and this is expected behaviour.

6.10 Readers–Writers Problem

Access to a database is shared among two kinds of processes: (1) readers and (2) writers (`readersWriters.urml`, source at Appendix B.7 on page 192). The readers execute transactions that only read the database, while the writers execute transactions that can both read and modify the database [13].

There are certain restrictions to apply a protocol for the readers and writers. First, while a writer is modifying a database, the writer must have exclusive access to that

```
generator logging to logger with: generator sending out a parcel to bin 1
chute0 logging to logger with: Chute: received to destination 1
chute0 logging to logger with: Chute: informing timer for 1 seconds
chute0 logging to logger with: Chute: sending parcel with id to 1
NON-DETERMINISM
chute1 logging to logger with: Chute: received to destination 1
chute1 logging to logger with: Chute: informing timer for 1 seconds
chute1 logging to logger with: Chute: sending parcel with id to 1
NON-DETERMINISM
switch logging to logger with: Switch: parcel entered, sending to 1
chute0 logging to logger with: Chute: received to destination 1
chute0 logging to logger with: Chute: informing timer for 1 seconds
chute0 logging to logger with: Chute: sending parcel with id to 1
NON-DETERMINISM
chute1 logging to logger with: Chute: received to destination 1
chute1 logging to logger with: Chute: informing timer for 1 seconds
chute1 logging to logger with: Chute: sending parcel with id to 1
NON-DETERMINISM
switch logging to logger with: Switch: parcel entered, sending to 1
bin1 logging to logger with: Bin: a parcel with id 1 has entered bin
```

Figure 6.53: Output for the parcel router model

database. Second, if no writer is updating the database, any number of readers can access the database at the same time.

In this section, a model that simulates the readers–writers problem is developed.

First, the actions of the model are discussed first; these actions are modelled through the protocols between the readers/writers and the controller.

6.10.1 Protocols — Actions

ControllerRW protocol

Between the controller and the reader/writer is the ControllerRW protocol, which has base gender at the controller side and conjugated gender at the reader/writer side.

The reader must request to read by sending the acquireRead() signal and when finished reading, release its reading access by sending the releaseRead() signal. These signals are synchronous, and so the reader is blocked until an acknowledge() signal is sent by the controller back to the reader.

Likewise, the writer must request to write by sending the acquireWrite() signal and when finished writing, release its exclusive access by sending the releaseWrite() signal. These signals are synchronous, and so the writer is blocked until an acknowledge() signal is sent by the controller back to the writer.

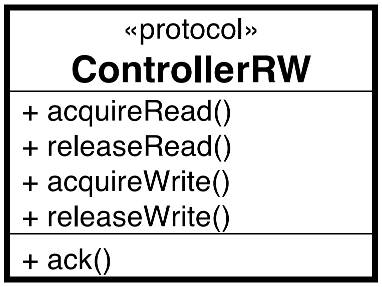


Figure 6.54: Class diagram for the ControllerRW protocol

ReadAndWrite protocol

The ReadAndWrite protocol provides communication between the reader/writer (base gender) and the shared database (conjugated gender). The reader may request to read() or write(data) with some data in the database; these signals only occur if the controller allows the reader/writer to interact with the database.

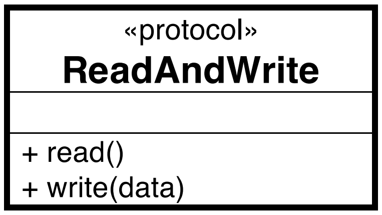


Figure 6.55: Class diagram for the ReadAndWrite protocol

6.10.2 Controller — Structure

The controller has the following attributes and ports:

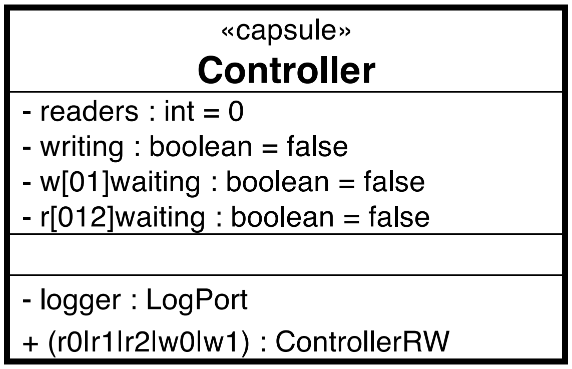

«capsule»
Controller
- readers : int = 0
- writing : boolean = false
- w[01]waiting : boolean = false
- r[012]waiting : boolean = false
- logger : LogPort
+ (r0|r1|r2|w0|w1) : ControllerRW

Figure 6.56: Class diagram for the Controller capsule

The controller contains several attributes that serve as read/write locks. These attributes are listed as follows:

readers is an integer that counts the number of readers that are currently in progress. The number of readers is required to be zero if the controller is in writing mode.

writing is a boolean variable that determines whether a writer is currently executing a transaction on the database. Being in writing mode must imply that the number of readers is zero.

w[01]waiting are boolean variables that determine whether w0 or w1 has sent a request to the controller and is waiting for the controller to return an acknowledge signal when it is free from reading or writing.

r[012]waiting are boolean variables that determine whether r0, r1, or r2 has sent a request to the controller and is waiting for the controller to return an acknowledge signal when it is free from writing.

The controller also connects to readers and writers for synchronization. The ConnectorRW ports to readers are r0, r1 and r2; the ConnectorRW ports to writers are w0 and w1.

6.10.3 Controller — Behaviour

The state diagram for the controller involves a fair number of transitions. Its state diagram is depicted in Figure 6.57 and each of the transitions will be discussed.

The state machine starts from the "none" state.

From the "none" state, it transitions to the "write" state if any of the following occurs:

- when w0 or w1 sends an acquireWrite signal to the controller, and when there is no activated readers and no activated writers. The controller sets the writing variable to true and return an acknowledgment signal back to w0 or w1.
- when there are no activated readers and no activated writers, and either w0 or w1 is waiting to be activated. The controller turns off the waiting flag, sets the writing variable to true, and return an acknowledgment signal back to w0 or w1.

From the "write" state, the controller transitions back to the "write" state if any of the following occurs:

- when w0 or w1 sends an acquireWrite signal to the controller. The controller turns on the waiting flag for w0 or w1.
- when r0, r1, or r2 sends an acquireRead signal to the controller. The controller turns on the waiting flag for r0, r1, or r2.

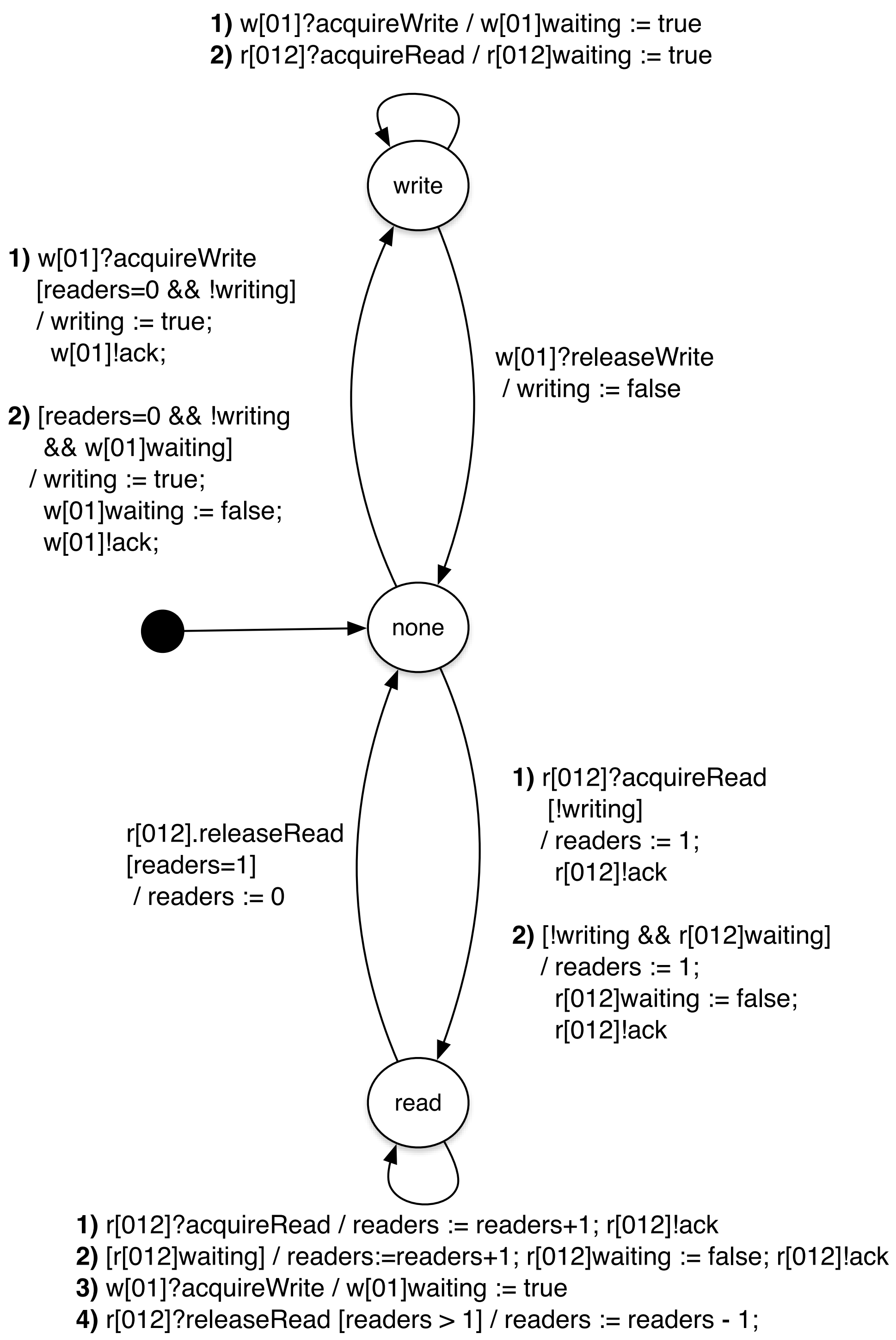

1) w[01]?acquireWrite / w[01]waiting := true
2) r[012]?acquireRead / r[012]waiting := true
write
1) w[01]?acquireWrite [readers=0 && !writing] / writing := true; w[01]!ack;
2) [readers=0 && !writing && w[01]waiting] / writing := true; w[01]waiting := false; w[01]!ack;
w[01]?releaseWrite / writing := false
none
r[012].releaseRead [readers=1] / readers := 0
1) r[012]?acquireRead [!writing] / readers := 1; r[012]!ack
2) [!writing && r[012]waiting] / readers := 1; r[012]waiting := false; r[012]!ack
read
1) r[012]?acquireRead / readers := readers+1; r[012]!ack
2) [r[012]waiting] / readers:=readers+1; r[012]waiting := false; r[012]!ack
3) w[01]?acquireWrite / w[01]waiting := true
4) r[012]?releaseRead [readers > 1] / readers := readers - 1;

Figure 6.57: State diagram for the Controller capsule in the readers–writers problem

The above two cases use the waiting flag to defer the acquisition of read/write lock until the controller finishes being in the “write” state.

The controller transitions from the “write” state to the “none” state when w0 or w1 sends a releaseWrite signal to the controller. The controller sets the writing variable to false.

Now the controller is back to the “none” state. It transitions to the “read” state if any of the following occurs:

- when r0, r1, or r2 sends an acquireRead signal to the controller and no writing is occurring. The controller sets readers to 1 and returns the acknowledgment signal to r0, r1, or r2.

- when no writing is occurring and when r0, r1, or r2 is waiting. The controller sets the readers to 1, turns the waiting flag back to off, and return the acknowledgement signal to r0, r1, or r2.

The controller transitions from the “read” state back to the “read” state if any of the following four conditions occurs:

- when r0, r1, or r2 sends an acquireRead signal. The controller increments the readers value and return an acknowledgement signal to r0, r1, or r2.

- when r0, r1, or r2 is waiting. The controller turns the waiting flag back off, increments the readers value, and returns an acknowledgement signal to r0, r1, or r2.

- when w0 or w1 send an acquireWrite signal. The controller defers the signal by setting the writer’s waiting flag to on.

- when r0, r1, or r2 sends an releaseRead signal and there are more than one readers reading. The controller decrements the reader value.

The controller transitions from the “read” state to the “none” state if r0, r1, or r2 sends a releaseRead signal and there is only one activated reader. The controller then sets the readers value to 0.

6.11 Educational Benefits and Possible Extensions

There are several educational benefits provided by the examples. First of all, the students are able to appreciate the complexity of concurrent systems. They are able to identify situations when non-determinism occurs, and to notice deadlocks and livelocks and how to avoid them. Secondly, the students are able to be familiar with various notations to model structure and behaviour of reactive systems. Thirdly, the students are able to practise using timers. Furthermore, additional concepts and techniques that could be using examples; these include exhaustive state space exploration, testing and validation, and test case generation.

The examples provided in this section can also be extended. For instance, one can avoid deadlocks in the dining philosopher model by adding a “butler” that centralizes the synchronization between the forks and philosophers. For various ways to extend these classical examples, one may refer to book *The Little Book of Semaphores* [7].

Chapter 7

Future Work and Conclusion

7.1 Debugger

Eclipse comes with several modes to launch a program—whether it is used for an external DSL or a regular programming language: (1) the **Run mode**, which launches the program in the standard way, (2) the **Debug mode**, which launches the program with additional options for debugging purposes, (3) the **Profile mode**, which launches the program to test the performance of the program. Additionally, modes for other purposes can be made by plugin developers.

So far, Urml supports only the Run mode by executing the interpreter as a thread directly on top of Eclipse's JVM. The launching framework requires a few abstractions to work for this custom interpreter: the target interpreter itself and user configuration to guide how the interpreter is to be launched. Other abstractions, such as (1) the console view to interact with the interpreter's I/O, (2) launch modes, (3) UI elements such as actions that delegate the launch, and (4) an asynchronous event dispatcher are already provided by the Eclipse platform.

The Debug mode, on the other hand, which requires the implementation of a debugger, is not yet developed. Building such a mode is not as trivial, as it at least provides a few more features such as to suspend/resume and to terminate the interpreter. As well, unlike the launching framework, the debug mode requires far more abstractions to be implemented by the plugin developer. These abstractions are collectively called the **Debug Model**, which is already defined as interfaces in Eclipse. The Eclipse platform itself provides a debugger UI that interacts directly with the Debug Model.

To implement a debugger in Eclipse, a debug model is to be implemented to wrap the various elements from the language-specific interpreter to be used by the Eclipse platform. A debugger may also provide additional features such as breakpoints and source lookup.

7.1.1 Eclipse Debug Model

Eclipse debug model provides a model of generic interfaces that allows the developer to provide concrete language-specific implementations for debuggers. The model defines generic **elements** that are common among typical imperative execution environments; these elements include threads, expressions, stack frames, variables, etc. Each of these elements is called `IDebugElement` in the model.

The model also defines generic **actions** that are common among debuggers. These actions include suspend-and-resume, disconnect, terminate, etc. A debug action may be implemented by various elements themselves; for example, a thread interface might implement a terminate action such that the thread has the ability to be terminated. Once an action is made, the implementation must fire a **debug event**. For example, by implementing the terminate action, the subclass that implements the thread interface must have a `terminate()` method. In this case, the `terminate()` method causes the threads to terminate and also causes a `DebugEvent.TERMINATE` event to be generated.

7.1.2 Breakpoints

A debugger may suspend execution at a specific location or when a specified condition is satisfied; these locations or conditions are known as **breakpoints**. The Eclipse platform provides facilities to ease the development and usage of breakpoints. One can already add, remove, and change breakpoint notifications; Eclipse even provides an API for manipulating breakpoints and for storing this information in a marker. Breakpoints are persistent, in that they remain between different invocations of Eclipse's workbench.

7.1.3 Source Lookup

Once the debugger suspends the program—possibly directly by the user, or when a breakpoint notifies a suspension—the debugger may provide information about the location where the program is currently at. The developer of the debug UI can use this information to implement source lookup such that the Eclipse platform can find the source code and display and highlight that code in the editor.

7.1.4 Challenges

In order to build a debugger around the interpreter, the interpreter needs to be remodelled to accommodate debugging facilities. There are challenges to perform this remodelling task.

First of all, it is not easy to make the interpreter terminate. In the main interpreter loop, typically one should store a volatile boolean variable to determine whether to terminate, and then check this variable at specific points in the loop. Where are these specific points? In Urml, there are 2 places: (1) the context switch when one capsule instance is changed to another, and (2) when an entry/exit code is run for states and an action code is run for transitions. The first place is easy as the context switch is implemented early in the loop; however, the second place is located deep within multiple methods in `ExpressionEvaluator` and `StatementExecuter` and it is burdensome for many methods to pass that one boolean variable deep in there. Currently, the interpreter throws an exception to interrupt itself in the 2 aforementioned places; however, throwing an exception in normal cases is not recommended programming practice and there must be better coding techniques to solve this problem.

Secondly, the plugin programmer must add a suspension point at each of the terminate points as well. Suspension points are places where the interpreter will suspend and wait (using the `Object.wait()` method) when the user requests to fire a suspend event. Again, adding such suspension points can be burdensome because various debug methods must be placed at *all* these suspension points.

Thirdly, the abstraction layer provided by Eclipse's Debug Model might assume too much about the user's implementation—it assumes that the user will build an imperative programming language. Since an imperative programming language is not being implemented here, the Eclipse's Debug Model cannot be directly mapped into the model Urml is building upon. There are artefacts from Urml that are significantly different from a typical imperative setting. Consider the case for capsules. The closest interface that matches a capsule is a thread, because having multiple capsules run at the same time simulates a multi-threading application. Nevertheless, there are things that a capsule has that a thread does not. For instance, a capsule may have various kinds of variable that can be used in code, such as local variables, incoming trigger variables, capsule attributes. However a thread only provides a stack frame, which allows the implementation of local variables. Further, the *structure* of capsules cannot be realized through Eclipse's Debug Model; for instance, some artefacts such as ports and connectors do not exist in an imperative language—let alone in a thread. The *behaviour* of capsules cannot be realized either. It is unknown how to glue in the concept of protocol into Eclipse's Debug Model, and thus, various commands—such as sending triggers, receiving triggers, and triggering a timeout event—are not trivial to map into imperative language settings.

Fourthly, breakpoints are not usable in most cases, because such breakpoints are useful only in action code. Line breakpoints can be made in action code only, and watch breakpoints involve variables that only exist in action code. No exception traps are possible to be made because there are no exceptions in the language. Thus, the potential for using breakpoints is severely limited in Urml.

7.2 Graphical Editor

While there can be only one model for the program, there can be multiple notations of the model: textual, graphical, mathematical, etc. The different views are merely different representations of the same model. Having such different views can be useful for different purposes. A programmer might prefer textual notation because s/he wants the code to be exact and specific. The detailed view of the model provided by textual notation might help. On the other hand, an architect might prefer a graphical

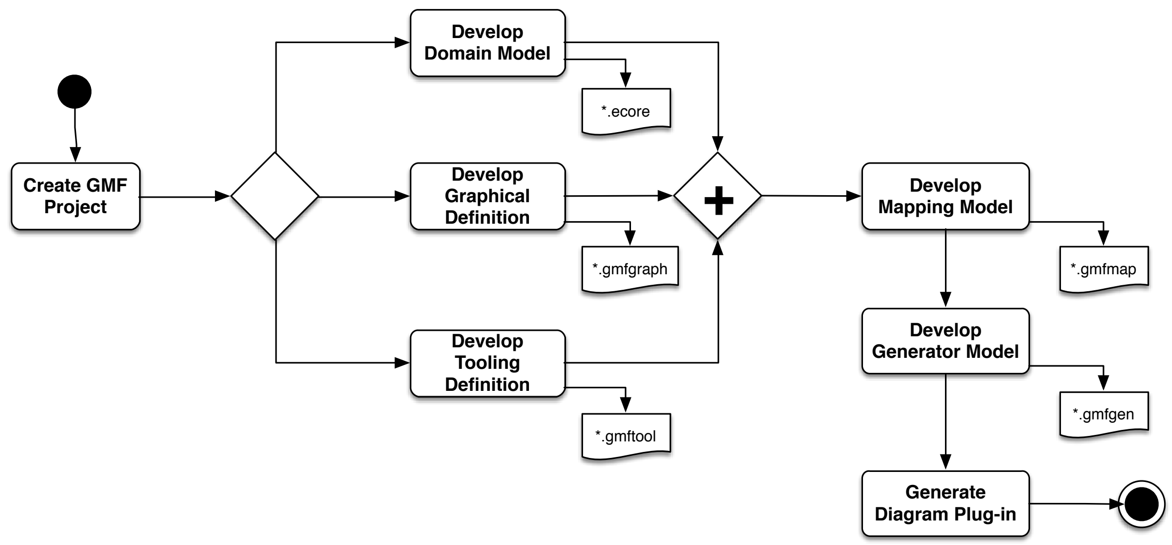


Figure 7.1: A typical workflow when working on GMF [9]

notation because s/he wants to view the big picture about the flow and structure of the model. This can be supported by the high-level view provided by a graphical notation. It is thus useful to build alternative notations, such as a graphical syntax, to accommodate these different needs.

7.2.1 Using the Graphical Modelling Framework (GMF)

The workflow for using GMF is shown in Figure 7.1. Once a GMF project is newly created, one needs (1) the Ecore domain model, which is already generated by Xtext through the grammar; (2) the graphical definition, which defines various graphical components such as figures, nodes, compartments, and connections for the editor; and (3) the tooling definition, which specifies functional tooling such as palette, creation tools, actions, in order for the graphical components defined in the graphical definition to be useful.

One can use these three models above to generate a mapping model, which consists of mappings of the Ecore domain elements to their corresponding figures and creation tools that are specified by the graphical definition and the tooling definition, respectively. Converted from the mapping model is the generator model, which loads the mapping model and the Ecore model to generate code for a graphical editor plugin.

7.2.2 Challenges

Since the graphical notation for Urml involves the use of three different kinds of diagrams—the class diagram, the structure diagram, and the state diagram—the textual file must be shared among multiple editors. This can create a few problems. First of all, with multiple instances of editors opened, multiple copies of the whole model will be loaded; with three duplicate models being loaded, unnecessarily large memory overhead is created. Secondly, whenever a file changes (i.e., when an editor becomes dirty), an editor must synchronize the changes. This produces an ambiguity: the user must either ask to update the changes to maintain the changed version, or the user must revert to the original saved file. Both of these options are problematic, as the former can potentially trap into an infinite loop of synchronization, and the latter might result in information loss.

7.3 Conclusion

In an educational environment, we need simple tools that help students develop a firm grasp of the core concepts of MDD for reactive systems, and Urml is developed to satisfy this need. Urml's primary contribution is to provide a scaffold so that the main pillars of modelling—(1) meta-modelling through language design, (2) model building, (3) model analysis and validation, and (4) model execution—can be shown in educational settings. Urml is based ROOM and RSA-RTE, in which the target domain is real-time and reactive systems. Its prominent concepts are the capsules, ports and state machines. Urml is developed on top of the Xtext framework, which generates the EMF model and corresponding IDE for its DSL. An interpreter is written in Java such that Urml models can be simulated. Future work for this project include a debugger and a graphical editor for Urml models.

Bibliography

[1] Heiko Behrens. Generation Gap Pattern, 2009. `http://heikobehrens.net/2009/04/23/generation-gap-pattern/`.

[2] Lorenzo Bettini. Xsemantics — a DSL for writing rules for Xtext languages, 2012. `http://xsemantics.sourceforge.net/`.

[3] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt, 2013.

[4] Lorenzo Bettini, Dietmar Stoll, Markus Voelter, and Serano Colameo. Approaches and tools for implementing type systems in Xtext. *Lecture Notes in Computer Science*, 7745:392–412, 2013.

[5] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Synthesis lecture on software engineering. Morgan & Claypool, 2012.

[6] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[7] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2009.

[8] Andrew Forward. *The Convergence of Modelling and Programming: Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language*. PhD thesis, University of Ottawa, 2010.

[9] Eclipse Foundation. Graphical Modeling Framework — Tutorial — Part 1, 2006. `http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1`.

[10] Eclipse Foundation. Eclipse Help — OCL Documentation — OCL Integration, 2013. `http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FIntegration.html`.

[11] Martin Fowler. *Domain-specific languages*. Addison-Wesley, 2011.

[12] itemis. Xtext documentation, 2012. `http://www.eclipse.org/Xtext/documentation.html`.

[13] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006.

[14] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[15] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *OOPSLA 08: Proceedings of the 23rd ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 563–582, 2008.

[16] Terrance Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[17] Henrik Rentz-Reichert. eTrice, 2014. `https://www.eclipse.org/etrice/`.

[18] M. Roth. CS 331 notes: LL grammar, 2008. `http://www.cs.uaf.edu/~cs331/notes/LL.pdf`.

[19] Bran Selic. Using UML for modelling complex real-time systems. *Lecture Notes in Computer Science*, 1474:250–260, 1998.

[20] Bran Selic. What will it take? A view on adoption of model-based methods in practice. *Software Systems Modelling*, 11(4):513–526, 2012.

[21] Bran Selic, G. Gullekson, and P. Ward. *Real-time object-oriented modelling.* Wiley, 1994.

[22] Ron Smith. EE499 and EE585 — Class Notes, 1999.

[23] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF — Eclipse Modeling Framework.* Addison-Wesley, 2009.

[24] John Vlissides. *Pattern Hatching: Design Patterns Applied.* Software Patterns Series. Addison-Wesley Professional, 1998.

[25] Markus Voelter. Xtext/TS — a type system framework for Xtext, 2011. `https://code.google.com/a/eclipselabs.org/p/xtext-typesystem/`.

[26] Markus Voelter. *DSL engineering: designing, implementing, and using domain-specific languages.* CreateSpace, 2013.

Appendix A

Urml Grammar

This chapter provides the Xtext source code for Urml's grammar.

```
grammar ca.queensu.cs.mase.Urml with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate urml "http://www.queensu.ca/cs/mase/Urml"

Model
  :   'model' name=ID "{"
      (
      capsules+=Capsule
      protocols+=Protocol
      )* '}'
  ;


/****************************************************************
 * Data and Variables
 ****************************************************************/


LocalVar
  :    name=ID
  ;


Attribute
  : 'attribute' name=ID
      (':=' defaultValue=Expression)?
  ;

```

```
/*************************************************************
 * Protocol
 *************************************************************/

Protocol
  : 'protocol' name=ID '{'
    (('incoming' '{' incomingMessages+=Signal* '}')
    ('outgoing' '{' outgoingMessages+=Signal* '}'))*
    '}'
  ;

Signal
  : name=ID
    '(' (LocalVars+=LocalVar (',' LocalVars+=LocalVar)*)? ')'
  ;

/*************************************************************
 * Capsules, Ports, and Connectors
 *************************************************************/

/**
 * The capsule
 */
Capsule
  : (root?='root')? 'capsule' name=ID '{'
    (
      'external' interfacePorts+=Port
      internalPorts+=Port
      timerPorts+=TimerPort
      logPorts+=LogPort
      attributes+=Attribute
      capsuleInsts+=CapsuleInst
      connectors+=Connector
      operations+=Operation
      statemachines+=StateMachine
    )*
    '}'
  ;

Operation
  : 'operation' name=ID
    '(' (LocalVars+=LocalVar (',' LocalVars+=LocalVar)*)? ')'
    '{'
      operationCode=OperationCode
    '}'
  ;
TimerPort:
  'timerPort' name=ID;
```

```
LogPort:
  'logPort' name=ID;
Port:
    'port' (conjugated?='~')? name=ID ':' protocol=[Protocol];
Connector:
  'connector' (capsuleInst1=[CapsuleInst] '.')? port1=[Port]
  'and' (capsuleInst2=[CapsuleInst] '.')? port2=[Port];
CapsuleInst:
  'capsuleInstance' name=ID ':' type=[Capsule];

/*************************************************************
 * State Machines
 *************************************************************/

StateMachine:
  {StateMachine}
  'stateMachine' '{'
    (
      states+=State_
        transitions+=Transition
    )*
  '}';

State_:
  (final?='final')? 'state' name=ID ('{'
    ('entry' '{' entryCode=ActionCode '}')?
    ('exit' '{' exitCode=ActionCode '}')?
    ('sub' substatemachine=StateMachine)?
  '}')?;


/*
 * transitions:
 */

Transition
  : 'transition' (name=ID)? ':'
    (init?='initial' from=[State_]) '->' to=[State_]
    '{'
      ('guard' '{' guard=Expression '}')?
      ('triggeredBy' ((triggers+=Trigger_in ('or' triggers+=Trigger_in)*)
          'timeout' timerPort=[TimerPort] universal?='*'
      ))?
      ('action' '{' action=ActionCode '}')?
    '}'
  ;


```

```
/*
 * other constructs for state machines:
 */

Trigger_in:
  from=[Port] '.'
  signal=[Signal]
  '(' (parameters+=IncomingVariable (','
    parameters+=IncomingVariable
  )*)? ')';

IncomingVariable
  : name=ID
  ;

Trigger_out:
  to=[Port] '.'
  signal=[SignalID]
  '('
    (parameters+=Expression (','
      parameters+=Expression
    )*)?
  ')'
  ;

/*****************************************************************
 * Operation and Action Code / Statements
 *****************************************************************/


// operation code

OperationCode
  :   statements+=StatementOperation+
  ;
StatementOperation
  :   NoOp
    Variable
    Invoke
    Assignment
     SendTrigger
     InformTimer
    WhileLoopOperation
    IfStatementOperation
    LogStatement
    ReturnStatement
  ;
WhileLoopOperation
```

```
  :   'while' condition=Expression '{' statements+=StatementOperation+ '}'
  ;
IfStatementOperation
  :   'if' condition=Expression '{' thenStatements+=StatementOperation+ '}'
    ('else ' '{' elseStatements+=StatementOperation+ '}')?
  ;
ReturnStatement
  :   {ReturnStatement} 'return' returnValue=Expression?
  ;


// action code

ActionCode
  :   statements+=Statement+
  ;
Statement
  :   SendTrigger
    Variable
    InformTimer
    NoOp
    Invoke
    Assignment
    WhileLoop
    IfStatement
    LogStatement
  ;
Variable
  : 'var' var=LocalVar (assign?=':=' exp=Expression)?
  ;
SendTrigger
  : 'send' triggers+=Trigger_out ('and' triggers+=Trigger_out)*
  ;
InformTimer
  : 'inform' timerPort=[TimerPortID] 'in' time=AdditiveExpression
  ;
NoOp
  : {NoOp} 'noop'
  ;
Invoke
  : 'call' operation=[OperationID]
    '('
      (parameters+=Expression (',' parameters+=Expression)*)?
    ')'
  ;
Assignment
  : lvalue=[AssignableID] ':=' exp=Expression
  ;
```

```

Assignable
  : LocalVar
    Attribute
  ;

WhileLoop
  : 'while' condition=Expression '{' statements+=Statement+ '}'
  ;
IfStatement
  : 'if' condition=Expression '{' thenStatements+=Statement+ '}'
    ('else ' '{' elseStatements+=Statement+ '}')?
  ;
LogStatement
  : 'log' logPort=[LogPort] 'with' left=StringExpression
  ;

StringExpression
  :   IndividualExpression => ({ConcatenateExpression.left=current} '^'
    rest=IndividualExpression)*
;

IndividualExpression returns StringExpression
  : expr=Expression str=STRING;

/*****************************************************************
 * Expressions
 *****************************************************************/

Expression
  : ConditionalOrExpression
  ;

ConditionalOrExpression returns Expression
    :    ConditionalAndExpression =>
         ({ConditionalOrExpression.left=current} '||'
         rest=ConditionalAndExpression)*
    ;

ConditionalAndExpression returns Expression
    :    RelationalOpExpression =>
         ({ConditionalAndExpression.left=current} '&&'
         rest=RelationalOpExpression)*
    ;

RelationalOpExpression returns Expression
  : AdditiveExpression  (=>
      (  {LessThanOrEqual.left=current}   "<="
            {LessThan.left=current}              "<"
              {GreaterThanOrEqual.left=current} ">="
            {GreaterThan.left=current}           ">"
            {Equal.left=current}                 "=="
            {NotEqual.left=current}              "!="
        )
      rest=AdditiveExpression)*
    ;

AdditiveExpression returns Expression
      :   MultiplicativeExpression (=>({Plus.left=current} '+'
                                        {Minus.left=current} '-'
                                       )
        rest=MultiplicativeExpression )*
      ;

MultiplicativeExpression returns Expression
      :   UnaryExpression (=>({Multiply.left=current} '*'
                            {Divide.left=current}     '/'
                            {Modulo.left=current}     '%'
                             )
          rest=UnaryExpression )*
      ;

UnaryExpression returns Expression
      :   UnaryExpressionNotPlusMinus
        ({UnaryExpression} '-' exp=UnaryExpression)
      ;

UnaryExpressionNotPlusMinus returns Expression
      :   NotBooleanExpression
        PrimaryExpression
      ;

NotBooleanExpression
    : '!' exp=UnaryExpression
    ;

PrimaryExpression returns Expression
    : LiteralOrIdentifier
      '(' Expression ')'
    ;

LiteralOrIdentifier returns Expression
    : Literal
      Identifier
    ;

```

```
Literal
  : IntLiteral
    BoolLiteral
    FunctionCall
  ;

IntLiteral
  :   {IntLiteral} int=INT
  ;

Identifier
  : id=[Identifiable]
  ;

Identifiable
  : Assignable
    IncomingVariable
  ;

FunctionCall
  :   {FunctionCall} call=[OperationID]
    '('
      (params+=Expression ( ',' params+=Expression)*)?
    ')'
  ;

BoolLiteral
  : {BoolLiteral} true=BOOLEAN
  ;

terminal BOOLEAN returns ecore::EBoolean
  : 'true'
    'false'
  ;
```

Appendix B

Sources for Urml Examples

This chapter provides the source for the examples provided in the Urml package.

B.1 A Simple Handshake

The file name is `handshake.urml`.

```
/**
 * A simple example that consists of a producer and a consumer of a message.
 */
model handshake {
  root capsule Handshake {
    capsuleInstance sender : Originator
    capsuleInstance receiver : Receiver
    connector sender.hand and receiver.hand
  }
  capsule Originator {
    external port hand : HandshakeProtocol
    logPort logger
    stateMachine {
      state start
      final state end
      transition init : initial -> start {
      }
      transition doHandShake : start -> end {
        action {
          send hand.shake()
          log logger with "sent a handshake"
        }
      }
    }
  }
  capsule Receiver {
    external port ~hand : HandshakeProtocol
    logPort logger
    stateMachine {
      state start
      final state end
      transition init : initial -> start {
      }
      transition receiveHandshake : start -> end {
        triggeredBy hand.shake()
        action {
          log logger with "received a handshake"
        }
      }
    }
  }
  protocol HandshakeProtocol {
    outgoing {
      shake()
    }
  }
}
```

B.2 Consumer–Producer Problem

The file name is `consumerProducer.urml`

```

model ConsumerProducer {
  root capsule ConsumerProducer {
    capsuleInstance c : Consumer
    capsuleInstance p : Producer
    capsuleInstance b : BoundedBuffer
    connector c.toGet and b.consumer
    connector p.toPut and b.producer
  }
  capsule Consumer {
    external port toGet : BufferProtocol
    logPort logger
    stateMachine {
      state single
      transition : initial -> single {
      }
      transition : single -> single {
        triggeredBy toGet.get(toDisplay)
        action {
          log logger with toDisplay
        }
      }
    }
}
capsule Producer {
  external port toPut : BufferProtocol
  stateMachine {
    state single {
      entry {
        var x := 1
        while (x < 8) {
          send toPut.put(x)
          x := x + 1
        }
      }
    }
    transition : initial -> single {
    }
  }
}
capsule BoundedBuffer {
  attribute a := 0
  attribute b := 0
  attribute c := 0
  attribute d := 0
  attribute e := 0
  external port ~consumer : BufferProtocol
  external port ~producer : BufferProtocol
  stateMachine {
    state zero
    state one
    state two
    state three
    state four
    state five
    transition init : initial -> zero {
    }
    transition : zero -> one {
      triggeredBy producer.put(a_)
      action {
        a := a_
      }
    }
    transition : one -> two {
      triggeredBy producer.put(b_)
      action {
        b := b_
      }
    }
    transition : two -> three {
      triggeredBy producer.put(c_)
      action {
        c := c_
      }
    }
    transition : three -> four {
      triggeredBy producer.put(d_)
      action {
        d := d_
      }
    }
    transition : four -> five {
      triggeredBy producer.put(e_)
      action {
        e := e_
      }
    }
    transition : five -> four {
      action {
        send consumer.get(e)
        e := 0
      }
    }
    transition : four -> three {
      action {
        send consumer.get(d)
        d := 0
      }
    }
    transition : three -> two {
      action {
        send consumer.get(c)
        c := 0
      }
    }
    transition : two -> one {
      action {
        send consumer.get(b)
        b := 0
      }
    }
    transition : one -> zero {
      action {
            send consumer.get(a)
            a := 0
          }
        }
      }
    }
    protocol BufferProtocol {
      incoming {
        get(data)
      }
      outgoing {
        put(data)
      }
    }
}
```

B.3 Widget Production Factory

The file name is `widgetProductionFactory.urml`

```
/*
 * A widget production factory
 * October 2012
 *
 * Adapted from EE585 Notes, Ron Smith, 2001
 */
model widgetManufacturing {

/*
   * The top capsule, containing instances of other
   * capsules and their connectors
   */
  root capsule SystemContainer {
    capsuleInstance controlSoftware : ControlSoftware
    capsuleInstance productionLine : ProductionLine
    connector controlSoftware.CS2RobotPort and productionLine.toobotPort
    connector controlSoftware.CS2WSPort and productionLine.toWSPort
  }

  /*
   * Controller
   */
  capsule ControlSoftware {
    external port CS2WSPort : WorkstationProtocol
    external port CS2RobotPort : RobotProtocol
    timerPort StartTimer
    timerPort StopTimer
    logPort CSLog
    attribute startUpDelay := 2000
    attribute systemStopTime := 30000
    stateMachine {
      transition init : initial -> startup {
      }
      state on {
        sub stateMachine {
          state startup {
            entry {
              inform StartTimer in startUpDelay
              inform StopTimer in systemStopTime
              log CSLog with "CS: startup"
            }
          }
          state produce {
            entry {
              send CS2WSPort.produceWidget()
              log CSLog with "CS: produce"
            }
          }
          state deliver {
            entry {
              send CS2RobotPort.deliverWidget()
              log CSLog with "CS: deliver"
            }
          }
        }
      }
      final state shutDown {
        entry {
          send CS2WSPort.shutDown() and CS2RobotPort.shutDown()
          log CSLog with "CS: shutDown"
        }
      }
      transition start : startup -> produce {
        triggeredBy timeout StartTimer
      }
      transition deliverMe : produce -> deliver {
        triggeredBy CS2WSPort.widgetProduced()
      }
      transition goAgain : deliver -> produce {
        triggeredBy CS2RobotPort.widgetDelivered()
      }
      transition stop : on -> shutDown {
        triggeredBy timeout StopTimer
      }
      //        transition stop2 : produce -> shutDown {
      //          triggeredBy timeout StopTimer
      //        }
      //        transition stop3 : deliver -> shutDown {
      //          triggeredBy timeout StopTimer
      //        }

    }
  }

  /*
   * The capsule that produces things
   */
  capsule ProductionLine {
    external port ~toWSPort : WorkstationProtocol
    external port ~toobotPort : RobotProtocol
    capsuleInstance workstationRef : Workstation
    capsuleInstance robotRef : Robot
    connector toWSPort and workstationRef.WS2CSPort
    connector toobotPort and robotRef.Robot2CSPort
  }
  capsule Workstation {
    external port ~WS2CSPort : WorkstationProtocol
    timerPort ProductionTimer
    logPort WSLog
    attribute widgetProductionTime := 3000
    stateMachine {
      state on {
        sub stateMachine {
          state standby {
            entry {
              log WSLog with "WS: standby"
            }
          }
          state producing {
            entry {
              inform ProductionTimer in widgetProductionTime
              log WSLog with "WS: producing"
            }
          }
        }
      }
      final state shutDown {
        entry {
          log WSLog with "WS: shutDown"
        }
      }
      transition t1 : initial -> standby {
      }
      transition begin : standby -> producing {
        triggeredBy WS2CSPort.produceWidget()
      }
      transition finished : producing -> standby {
        triggeredBy timeout ProductionTimer
        action {
          send WS2CSPort.widgetProduced()
        }
      }
      transition stop : on -> shutDown {
        triggeredBy WS2CSPort.shutDown()
      }
      //      transition stop2 : standby -> shutDown {
      //        triggeredBy WS2CSPort.shutDown()
      //      }

    }
  }
  capsule Robot {
    external port ~Robot2CSPort : RobotProtocol
    timerPort DeliveryTimer
    logPort RobotLog
    attribute widgetDeliveringTime := 5000
    stateMachine {
      state on {
        sub stateMachine {
          state standby {
            entry {
              log RobotLog with "Robot: standby"
            }
          }
          state delivering {
            entry {
              inform DeliveryTimer in widgetDeliveringTime
              log RobotLog with "Robot: delivering"
            }
          }
        }
      }
      final state shutDown {
        entry {
          log RobotLog with "Robot: shutDown"
        }
      }
      transition init : initial -> standby {
      }
      transition start : standby -> delivering {
        triggeredBy Robot2CSPort.deliverWidget()
      }
      transition finished : delivering -> standby {
        triggeredBy timeout DeliveryTimer
        action {
          send Robot2CSPort.widgetDelivered()
        }
      }
      transition stop1 : standby -> shutDown {
        triggeredBy Robot2CSPort.shutDown()
      }
      transition stop2 : delivering -> shutDown {
        triggeredBy Robot2CSPort.shutDown()
      }
    }
  }
  protocol WorkstationProtocol {
    incoming {
      widgetProduced()
    }
    outgoing {
      produceWidget()
      shutDown()
    }
  }
  protocol RobotProtocol {
    incoming {
      widgetDelivered()
    }
    outgoing {
      deliverWidget()
      shutDown()
    }
  }
}
```

B.4 Blinking Yellow Light

The file name is `blinky.urml`.

```
/*
 * A blinking yellow light
 * October 2012
 *
 * This example involves a yellow light that blinks. The yellow light blinks by turning on
 * for 1 second and then off for 1 second. Additionally, the yellow light proceeds a cycle
 * of alternately blinking for 5 seconds and stop blinking for 5 seconds.
 */
model Blinky {

/*
   * The top of the model that consists of Blinky, the blinking yellow light,
   * and its controller, which coordinates the blink-and-stop cycle.
   */
  root capsule Top {
    capsuleInstance blinky : BlinkingLight
    capsuleInstance controller : Controller
    connector blinky.connectToController and controller.connectToLight
  }

  /*
   * The blinking light blinks --- it turns on for 1 second and then off
   * for 1 second.
   */
  capsule BlinkingLight {
    external port connectToController : ControllerProtocol
    timerPort onAndOff_Timer
    logPort logger
    attribute onAndOff_Period := 1000

    /*
     * This state machine describes the cycle alternating between
     * the light blinking and not blinking, which is coordinated by
     * the controller.
     */
    stateMachine {
      transition init : initial -> off {
      }
      transition startBlinking : off -> subOn {
        triggeredBy connectToController.start()
      }
      transition stopBlinking : blinking -> off {
        triggeredBy connectToController.stop()
      }
      state off
      /*
      * The yellow light is in blinking mode
      */
      state blinking {
        sub stateMachine {
          state subOn {
            entry {
              inform onAndOff_Timer in onAndOff_Period
              log logger with "Lights " ^ "turn to yellow for " ^ onAndOff_Period / 1000 ^
                   " seconds..."
            }
          }
          transition on2off : subOn -> subOff {
            triggeredBy timeout onAndOff_Timer
          }
          state subOff {
            entry {
              inform onAndOff_Timer in onAndOff_Period
              log logger with "Lights turn off for " ^ onAndOff_Period / 1000 ^ " seconds"
            }
          }
          transition subOff2on : subOff -> subOn {
            triggeredBy timeout onAndOff_Timer
          }
          transition subInit : initial -> subOn {
          }
        }
      }
    }
}

/*
 * The controller makes sure that the yellow light blinks for 5 seconds and
 * then stops blinking for 5 seconds.
 */
capsule Controller {
  external port ~connectToLight : ControllerProtocol
  timerPort blinkingCycleTimer
  attribute blinkingCyclePeriod := 5000
  logPort logger
  stateMachine {
    transition init : initial -> on {
      action {
        inform blinkingCycleTimer in blinkingCyclePeriod
        send connectToLight.start()
        log logger with "connect to light start"
      }
    }
    transition on2off : on -> off {
      triggeredBy timeout blinkingCycleTimer
      action {
        inform blinkingCycleTimer in blinkingCyclePeriod
        send connectToLight.stop()
        log logger with "Blinky stops blinking for " ^ blinkingCyclePeriod / 1000 ^ "
             seconds"
      }
      }
      transition off2On : off -> on {
        triggeredBy timeout blinkingCycleTimer
        action {
          inform blinkingCycleTimer in blinkingCyclePeriod
          send connectToLight.start()
          log logger with "Blinky blinks for " ^ blinkingCyclePeriod / 1000 ^ " seconds"
        }
      }
      state on {
      }
      state off {
      }
    }
  }
  protocol ControllerProtocol {
    incoming {
      start()
      stop()
    }
    outgoing {
    }
  }
}
```

B.5 Dining Philosophers

The file name is diningPhilosophers.urml

```
/*
 * Dining Philosophers Example
 * October 2012
 *
 * This example involves n philosophers sharing n forks,
 * where for each time a philosopher performs
 * its tasks, it needs 2 forks that is at its rights
 * and its left.
 */
model DiningPhilosophers {
/*
   * A container capsule that consists of a ring of philosophers and forks.
   */
  root capsule PhilosophersRing {
    capsuleInstance phil0 : Philosopher
    capsuleInstance phil1 : Philosopher
    capsuleInstance phil2 : Philosopher
    capsuleInstance phil3 : Philosopher
    capsuleInstance phil4 : Philosopher
    port phil0signer : RingPhilosopherProtocol
    port phil1signer : RingPhilosopherProtocol
    port phil2signer : RingPhilosopherProtocol
    port phil3signer : RingPhilosopherProtocol
    port phil4signer : RingPhilosopherProtocol
    connector phil0signer and phil0.signer
    connector phil1signer and phil1.signer
    connector phil2signer and phil2.signer
    connector phil3signer and phil3.signer
    connector phil4signer and phil4.signer
    capsuleInstance fork0 : Fork
    capsuleInstance fork1 : Fork
    capsuleInstance fork2 : Fork
    capsuleInstance fork3 : Fork
    capsuleInstance fork4 : Fork
    connector phil0.left and fork4.right
    connector fork4.left and phil4.right
    connector phil4.left and fork3.right
    connector fork3.left and phil3.right
    connector phil3.left and fork2.right
    connector fork2.left and phil2.right
    connector phil2.left and fork1.right
    connector fork1.left and phil1.right
    connector phil1.left and fork0.right
    connector fork0.left and phil0.right
    stateMachine {
      state one {
      }
      transition init : initial -> one {
        action {
          send phil0signer.sign(0)
          send phil1signer.sign(1)
          send phil2signer.sign(2)
          send phil3signer.sign(3)
          send phil4signer.sign(4)
        }
      }
    }
  }

  /*
   * A fork that is asked to be picked up or put down
   */
  capsule Fork {
    external port ~left : PhilosopherForkProtocol
    external port ~right : PhilosopherForkProtocol
    logPort logger
    /* Boolean value that specifies that the philosopher on the left side of the
     * fork is waiting.
     */
    attribute leftWaiting := false
    /* Boolean value that specifies that the philosopher on the right side of the
     * fork is waiting
     */
    attribute rightWaiting := false
    stateMachine {
      state down {
        entry {
          log logger with "down"
        }
      }
      state up {
        entry {
          log logger with "up"
        }
      }
      transition init : initial -> down {
      }
      transition pickUpFromLeft : down -> up {
        triggeredBy left.up()
        action {
          send left.ack()
        }
      }
      transition pickUpFromRight : down -> up {
        triggeredBy right.up()
        action {
          send right.ack()
        }
      }
      /*
       * While already up, but asked to be picked up again,
       * set the waiting flag to true.
       */
      transition pickUpFromLeftButIsAlreadyUp : up -> up {
        triggeredBy left.up()
        action {
          leftWaiting := true
        }
      }
      transition pickUpFromRightButIsAlreadyUp : up -> up {
        triggeredBy right.up()
        action {
          rightWaiting := true
        }
      }
      /*
       * While already up, but the left/side is already waiting,
       * exchange the fork from left/right to right/left.
       */
      transition switchFromLeftToRight : up -> up {
        guard {
          rightWaiting == true
        }
        triggeredBy left.down()
        action {
          send right.ack()
          rightWaiting := false
          log logger with "fork exchanged from left to right"
        }
      }
      transition switchFromRightToLeft : up -> up {
        guard {
          leftWaiting == true
        }
        triggeredBy right.down()
        action {
          send left.ack()
          leftWaiting := false
          log logger with "fork exchanged from right to left"
        }
      }
      /*
       * Put down the fork if asked and when left/right is not waiting.
       */
      transition putDownForLeft : up -> down {
        guard {
          rightWaiting == false
        }
        triggeredBy left.down()
      }
      transition putDownForRight : up -> down {
        guard {
          leftWaiting == false
        }
        triggeredBy right.down()
      }
    }
  }

  /*
   * A philosopher alternates its states in a cycle of (1) thinking,
   * (2) picking up its left fork, (3) picking up its right fork, (4)
   * eating, (5) putting down its left fork, and (6) putting down its
   * right fork.
   */
  capsule Philosopher {
    attribute id := −1
    attribute delayTimeout := 300
    attribute eatTimeout := 600
    attribute thinkTimeout := 600
    external port left : PhilosopherForkProtocol
    external port right : PhilosopherForkProtocol
    external port ~signer : RingPhilosopherProtocol
    logPort logger
    timerPort eatTimer
    timerPort thinkTimer
    timerPort delayTimer
    operation isEvenPhil() {
      return id % 2 == 0
    }
    operation setID(id_) {
      id := id_
    }
    stateMachine {
      state start
      state delay {
        entry {
          inform delayTimer in (delayTimeout * id)
        }
      }
      state think {
        entry {
          inform thinkTimer in thinkTimeout
          log logger with "thinking..."
        }
      }
      state pickingUpFirstFork {
        entry {
          log logger with "picking up the first fork"
        }
      }
      state gotFirstFork {
        entry {
          log logger with "got the first fork"
        }
      }
      state pickingUpSecondFork {
        entry {
          log logger with "picking up the second fork"
        }
      }
      state gotSecondFork {
        entry {
          log logger with "got the second fork"
        }
      }
      state eating {
        entry {
          inform eatTimer in eatTimeout
          log logger with "eating... om nom nom"
        }
      }
      state puttingDownFirstFork {
        entry {
          log logger with "putting down the first fork"
        }
      }
      state puttingDownSecondFork {
        entry {
          log logger with "putting down the second fork"
        }
      }
      transition init : initial -> start {
      }
      transition setPhilID : start -> delay {
        triggeredBy signer.sign(num)
        action {
          call setID(num)
        }
      }
      transition startToThink : delay -> think {
        triggeredBy timeout delayTimer
      }
      transition pickUpFirstFork : think -> pickingUpFirstFork {
        triggeredBy timeout thinkTimer
        action {
          if isEvenPhil() {
            send right.up()
          } else {
            send left.up()
          }
        }
      }
      transition waitAckFromFirstFork : pickingUpFirstFork -> gotFirstFork {
        triggeredBy left.ack() or right.ack()
      }
      transition pickUpSecondFork : gotFirstFork -> pickingUpSecondFork {
        action {
          if isEvenPhil() {
            send left.up()
          } else {
            send right.up()
          }
        }
      }
      transition waitAckFromSecondFork : pickingUpSecondFork -> gotSecondFork {
        triggeredBy left.ack() or right.ack()
      }
      transition goEat : gotSecondFork -> eating {
      }
      transition putDownFirstFork : eating -> puttingDownFirstFork {
        triggeredBy timeout eatTimer
        action {
          send left.down()
        }
      }
      transition putDownSecondFork : puttingDownFirstFork -> puttingDownSecondFork {
        action {
          send right.down()
        }
      }
      transition goThink : puttingDownSecondFork -> think {
      }
    }
}

/*
 * Communication portal between a philosopher and its forks
 */
protocol PhilosopherForkProtocol {
  incoming {
    ack()
  }
  outgoing {
    up()
    down()
  }
}
protocol RingPhilosopherProtocol {
  incoming {
  }
  outgoing {
    sign(id)
  }
}
```

```
}
```

B.6 Parcel Router

The file name is `parcelRouter.urml`

```
/*
 * Parcel Router
 * October 2012
 *
 * This example illustrates a parcel router. The parcel router first generates
 * packages for every 4 second; then, it delivers the packages into 4 bins.
 */
model ParcelRouter {
/*
   * Protocol to communicate from the stage to the sensor, so that the stage knows
   * whether to send parcels to its left or its right; the direction to which the stage
   * sends depends on the level nuber of that stage
   */
  protocol SensorProtocol {
    incoming {
    }
    outgoing {
    // leftNotRight is a boolean: if true, then the
      sendDirection(leftNotRight)
    }
  }
  /*
   * Protocol to simulate the passing of the parcels.
   */
  protocol ParcelPassage {
    incoming {
    }
    outgoing {
      sendParcel(destinationCode)
    }
  }
  protocol LevelNumberProtocol {
    incoming {
    }
    outgoing {
      sendLevelNumber(level)
    }
  }
  /*
   * The top capsule of the parcel router.
   */
  root capsule ParcelRouter {
  // -- PARCEL ROUTING --
  // generates a parcel
    capsuleInstance generator : Generator
    // first-level stage that distributes parcels to one of the two second-level stages
    capsuleInstance stage0 : Stage
    // second-level stage that distributes parcels to one of bin 0 and bin 1
    capsuleInstance stage1 : Stage
    // second-level stage that distributes parcels to one of bin 2 and bin 3
    capsuleInstance stage2 : Stage
    // the bins are final destinations for the parcels
    capsuleInstance bin0 : Bin
    capsuleInstance bin1 : Bin
    capsuleInstance bin2 : Bin
    capsuleInstance bin3 : Bin
    // connectors provide routes where the parcels will go
    connector generator.gettingOut and stage0.goingIn
    connector stage0.gettingOutRight and stage1.goingIn
    connector stage0.gettingOutLeft and stage2.goingIn
    connector stage1.gettingOutRight and bin0.goingIn
    connector stage1.gettingOutLeft and bin1.goingIn
    connector stage2.gettingOutRight and bin2.goingIn
    connector stage2.gettingOutLeft and bin3.goingIn
    // -- ASSIGNING LEVEL NUMBERS: We want to let the stages be aware of their own
        levels.
    // ports to put the level numbers in from this top capsule
    port toStage0 : LevelNumberProtocol
    port toStage1 : LevelNumberProtocol
    port toStage2 : LevelNumberProtocol
    // connectors provide routes to assign level numbers to the stages from this top capsule
    connector toStage0 and stage0.fromTop
    connector toStage1 and stage1.fromTop
    connector toStage2 and stage2.fromTop
    // when the model starts, this top capsule assigns a level number to each of the stages
    stateMachine {
      final state single
      transition init : initial -> single {
        action {
          send toStage0.sendLevelNumber(1)
          send toStage1.sendLevelNumber(0)
          send toStage2.sendLevelNumber(0)
        }
      }
    }
  }
  /*
   * A capsule that generates parcels
   */
  capsule Generator {
    external port gettingOut : ParcelPassage
    timerPort timer
    logPort logger
    attribute timeoutPeriod := 10000
    attribute destinationId := 1
    //     attribute sent := false
    stateMachine {
      state single
      transition init : initial -> single {
        action {
          inform timer in timeoutPeriod
        }
      }
      transition sending : single -> single {
      //         guard {
      //           !sent
      //         }

        triggeredBy timeout timer
        action {
          inform timer in timeoutPeriod
          send gettingOut.sendParcel(destinationId)
          log logger with "generator sending out a parcel to bin " ^ destinationId
          destinationId := destinationId + 1
          //           sent := true

        }
      }
    }
  }
  capsule Stage {
    external port ~goingIn : ParcelPassage
    external port gettingOutRight : ParcelPassage
    external port gettingOutLeft : ParcelPassage
    external port ~fromTop : LevelNumberProtocol
    port toSensorController : LevelNumberProtocol
    logPort logger
    attribute level
    capsuleInstance chute0 : Chute
    capsuleInstance chute1 : Chute
    capsuleInstance sensorCntl : SensorController
    capsuleInstance switch : Switch
    connector goingIn and chute0.enter
    connector chute0.leave and chute1.enter
    connector chute0.toControl and sensorCntl.sense
    connector chute1.leave and switch.enter
```

```
    connector chute1.toControl and chute1.dummy
    connector sensorCntl.setSwitch and switch.setSwitch
    connector switch.leaveLeft and gettingOutLeft
    connector switch.leaveRight and gettingOutRight
    connector toSensorController and sensorCntl.fromStage
    stateMachine {
      state single
      transition init : initial -> single {
      }
      transition setLevelNumber : single -> single {
        triggeredBy fromTop.sendLevelNumber(levelNumber)
        action {
          level := levelNumber
          send toSensorController.sendLevelNumber(levelNumber)
        }
      }
    }
  }
  capsule Bin {
    external port ~goingIn : ParcelPassage
    attribute numOfParcels := 0
    logPort logger
    stateMachine {
      state single
      transition init : initial -> single {
      }
      transition sending : single -> single {
        triggeredBy goingIn.sendParcel(designatedBin)
        action {
          numOfParcels := numOfParcels + 1
          log logger with "Bin: a parcel with id " ^ designatedBin ^ " has entered bin"
        }
      }
    }
  }
  capsule Chute {
    external port ~enter : ParcelPassage
    external port leave : ParcelPassage
    external port ~dummy : ParcelPassage
    external port toControl : ParcelPassage
    logPort logger
    timerPort timer
    attribute storedParcelDestination := −1
    attribute timeoutPeriod := 1000
    stateMachine {
      state empty
      state entered
      transition init : initial -> empty {
      }
      transition pass1 : empty -> empty {
        triggeredBy dummy.sendParcel(destination)
      }
      transition pass2 : entered -> entered {
        triggeredBy dummy.sendParcel(destination)
      }
      transition parcelEnters : empty -> entered {
        triggeredBy enter.sendParcel(destination)
        action {
          storedParcelDestination := destination
          log logger with "Chute: received to destination " ^ destination
          inform timer in timeoutPeriod
          log logger with "Chute: informing timer for " ^ timeoutPeriod / 1000 ^ "
              seconds"
        }
      }
      transition parcelLeaves : entered -> empty {
        triggeredBy timeout timer
        action {
          send leave.sendParcel(storedParcelDestination)
          send toControl.sendParcel(storedParcelDestination)
          log logger with "Chute: sending parcel with id to " ^ storedParcelDestination
          storedParcelDestination := -1
        }
      }
    }
  }
  capsule Switch {
    external port ~setSwitch : SensorProtocol
    external port ~enter : ParcelPassage
    external port leaveLeft : ParcelPassage
    external port leaveRight : ParcelPassage
    timerPort timer
    logPort logger
    attribute leftNotRight := true
    attribute timeoutPeriod := 2000
    attribute storedParcelDestination := -1
    stateMachine {
      state empty
      state entered
      transition init : initial -> empty {
      }
      transition setDirectionWhileEmpty : empty -> empty {
        triggeredBy setSwitch.sendDirection(direction)
        action {
          leftNotRight := direction
        }
      }
      transition setDirectionWhileEntered : entered -> entered {
        triggeredBy setSwitch.sendDirection(direction)
        action {
          leftNotRight := direction
        }
      }
      transition parcelEnters : empty -> entered {
        triggeredBy enter.sendParcel(destination)
        action {
          storedParcelDestination := destination
          log logger with "Switch: parcel entered, sending to " ^ destination
          inform timer in timeoutPeriod
        }
      }
      transition parcelLeaves : entered -> empty {
        triggeredBy timeout timer
        action {
          if leftNotRight {
            send leaveLeft.sendParcel(storedParcelDestination)
          } else {
            send leaveRight.sendParcel(storedParcelDestination)
          }
        }
      }
    }
  }
  capsule SensorController {
    external port ~sense : ParcelPassage
    external port setSwitch : SensorProtocol
    external port ~fromStage : LevelNumberProtocol
    attribute level
    // key = destination << level & 1
    operation leftShiftAndBitmask(destination, level) {
      var counter := 0
      while (counter < level) {
        destination := destination / 2
        counter := counter + 1
      }
      if (destination % 2 == 1) {
        return 1
      } else {
        return 0
      }
    }
    stateMachine {
      state single
      transition init : initial -> single {
      }
      transition setLevelNumber : single -> single {
        triggeredBy fromStage.sendLevelNumber(levelNumber)
        action {
          level := levelNumber
        }
      }
      transition senseParcel : single -> single {
        triggeredBy sense.sendParcel(destination)
        action {
          send setSwitch.sendDirection(leftShiftAndBitmask(destination, level) != 0)
        }
      }
    }
  }
}
```

B.7 Readers–Writers Problem

The file name is `readersWriters.urml`

```
/*
 * Readers– Writers Scenario
 * November 2012
 *
 * In the Readers– Writers problem, a database is shared among two kinds
 * of processes.
 *
 * (1) Readers. A reader process may execute a transaction that
 *     examines the data in a database.
 *
 * (2) Writers. A writer process may execute a transaction that
 *     examines and updates the data in a database.
 *
 * There are a few constraints for this sharing scheme. First, a
 * writer process must have exclusive access to the database
 * while updating it. Secondly, any number of readers is allowed
 * to access concurrently the database when no writer process is
 * accessing the database.
 */
model ReadersWriters {
  root capsule Top {
    capsuleInstance r0 : Reader
    capsuleInstance r1 : Reader
    capsuleInstance r2 : Reader
    capsuleInstance w0 : Writer
    capsuleInstance w1 : Writer
    capsuleInstance c  :  Controller
    capsuleInstance db : Database
    connector r0.con and c.r0
    connector r1.con and c.r1
    connector r2.con and c.r2
    connector w0.con and c.w0
    connector w1.con and c.w1
    connector r0.read and db.read
    connector r1.read and db.read
    connector r2.read and db.read
    connector w0.write and db.write
    connector w1.write and db.write
  }
  capsule Controller {
    attribute readers := 0
    attribute writing := false
    attribute w0waiting := false
    attribute w1waiting := false
    attribute r0waiting := false
    attribute r1waiting := false
    attribute r2waiting := false
    logPort logger
    external port r0  : ControllerRW
    external port r1  : ControllerRW
    external port r2  : ControllerRW
    external port w0 : ControllerRW
    external port w1 : ControllerRW
    stateMachine {
      state none
      state writing {
        entry {
          log logger with "CONTROLLER WRITING"
        }
      }
      state reading {
        entry {
          log logger with "CONTROLLER READING"
        }
      }
      transition  init  :  initial  -> none {
      }
      transition  askedWriteW0 : none -> writing {
        guard {
          readers == 0 && !writing
        }
        triggeredBy w0.acquireWrite()
        action {
```

```
        writing := true
        log logger with "acquire write W0"
        send w0.acknowledge()
      }
    }
    transition askedWriteW0A : none -> writing {
      guard {
        readers == 0 && !writing && w0waiting
      }
      action {
        writing := true
        send w0.acknowledge()
        w0waiting := false
      }
    }
    transition askedWriteW1 : none -> writing {
      guard {
        readers == 0 && !writing
      }
      triggeredBy w1.acquireWrite()
      action {
        writing := true
        log logger with "acquire write W1"
        send w1.acknowledge()
      }
    }
    transition askedWriteW1A : none -> writing {
      guard {
        readers == 0 && !writing && w1waiting
      }
      action {
        writing := true
        send w1.acknowledge()
        w1waiting := false
      }
    }
    transition askedReadR0 : none -> reading {
      guard {
        writing == false
      }
      triggeredBy r0.acquireRead()
      action {
        readers := 1
        log logger with "acquire read R0"
        send r0.acknowledge()
      }
    }
    transition askedReadR1 : none -> reading {
      guard {
        writing == false
      }
      triggeredBy r1.acquireRead()
      action {
        readers := 1
        log logger with "acquire read R1"
        send r1.acknowledge()
      }
    }
    transition askedReadR2 : none -> reading {
      guard {
        writing == false
      }
      triggeredBy r2.acquireRead()
      action {
        readers := 1
        log logger with "acquire read R2 with one reader"
        send r2.acknowledge()
      }
    }
    transition askedReadR0A : none -> reading {
      guard {
        !writing && r0waiting
      }
      action {
        readers := 1
        r0waiting := false
        log logger with "acquire read R0 with one reader"
        send r0.acknowledge()
      }
    }
    transition askedReadR1A : none -> reading {
      guard {
        !writing && r1waiting
      }
      action {
        readers := 1
        r1waiting := false
        log logger with "acquire read R1 with one reader"
        send r1.acknowledge()
      }
    }
    transition askedReadR2A : none -> reading {
      guard {
        !writing && r2waiting
      }
      action {
        readers := 1
        r2waiting := false
        log logger with "acquire read R2 with one reader"
        send r2.acknowledge()
      }
    }
    transition multipleAskedReaderR0 : reading -> reading {
      triggeredBy r0.acquireRead()
      action {
        readers := readers + 1
        log logger with "acquire read R0 with " ^ readers ^ " readers"
        send r0.acknowledge()
      }
    }
    transition multipleAskedReaderR1 : reading -> reading {
      triggeredBy r1.acquireRead()
      action {
        readers := readers + 1
        log logger with "acquire read R1 with " ^ readers ^ " readers"
        send r1.acknowledge()
      }
    }
    transition multipleAskedReaderR2 : reading -> reading {
      triggeredBy r2.acquireRead()
      action {
        readers := readers + 1
        log logger with "acquire read R2 with " ^ readers ^ " readers"
        send r2.acknowledge()
      }
    }
    transition multipleAskedReaderR0A : reading -> reading {
      guard {
        r0waiting
      }
      action {
        readers := readers + 1
        r0waiting := false
        log logger with "acquire read R0 with " ^ readers ^ " readers"
        send r0.acknowledge()
      }
    }
    transition multipleAskedReaderR1A : reading -> reading {
      guard {
        r1waiting
      }
      action {
        readers := readers + 1
        r1waiting := false
        log logger with "acquire read R1 with " ^ readers ^ " readers"
        send r1.acknowledge()
      }
    }
    transition multipleAskedReaderR2A : reading −> reading {
      guard {
        r2waiting
      }
      action {
        readers := readers + 1
        r2waiting := false
        log logger with "acquire read R2 with " ^ readers ^ " readers"
        send r2.acknowledge()
      }
    }
    transition w0requestR : reading −> reading {
      triggeredBy w0.acquireWrite()
      action {
        w0waiting := true
      }
    }
    transition w0requestW : writing −> writing {
      triggeredBy w0.acquireWrite()
      action {
        w0waiting := true
      }
    }
    transition w1requestR : reading −> reading {
      triggeredBy w1.acquireWrite()
      action {
        w1waiting := true
      }
    }
    transition w1requestW : writing −> writing {
      triggeredBy w1.acquireWrite()
      action {
        w1waiting := true
      }
    }
    transition r0request : writing −> writing {
      triggeredBy r0.acquireRead()
      action {
        r0waiting := true
      }
    }
    transition r1request : writing −> writing {
      triggeredBy r1.acquireRead()
      action {
        r1waiting := true
      }
    }
    transition r2request : writing -> writing {
      triggeredBy r2.acquireRead()
      action {
        r2waiting := true
      }
    }
    transition releaseWrite : writing -> none {
      triggeredBy w0.releaseWrite() or w1.releaseWrite()
      action {
        writing := false
        log logger with "release write"
      }
    }
    transition multipleReleaseRead : reading -> reading {
      guard {
        readers > 1
      }
      triggeredBy r0.releaseRead() or r1.releaseRead() or r2.releaseRead()
      action {
        readers := readers - 1
        log logger with "release read with " ^ readers ^ " readers remaining"
      }
    }
    transition releaseRead : reading -> none {
      guard {
        readers == 1
      }
      triggeredBy r0.releaseRead() or r1.releaseRead() or r2.releaseRead()
      action {
        readers := 0
        log logger with "release read with no readers remaining"
      }
    }
  }
}
capsule Reader {
  external port ~con : ControllerRW
  external port read : ReadAndWriteThings
  attribute readRequestSent := false
  timerPort timer
  logPort logger
  attribute timeoutPeriod := 200
  stateMachine {
    state notReading
    state reading {
      entry {
        send read.read()
        log logger with '' //cell

      }
    }
    transition init : initial -> notReading {
    }
    transition getReading : notReading -> notReading {
      guard {
        readRequestSent == false
      }
      action {
        send con.acquireRead()
        readRequestSent := true
      }
    }
    transition goReading : notReading -> reading {
      triggeredBy con.acknowledge()
      action {
        log logger with "goReading"
        readRequestSent := false
        inform timer in timeoutPeriod
      }
    }
    transition backToNotReading : reading -> notReading {
      triggeredBy timeout timer
      action {
        send con.releaseRead()
      }
    }
  }
}
capsule Writer {
  external port ~con : ControllerRW
  external port write : ReadAndWriteThings
  attribute writeRequestSent := false
  timerPort timer
  logPort logger
  attribute timeoutPeriod := 200
  stateMachine {
    state notWriting
    state writing {
      entry {
        log logger with "writing 1234"
        send write.write(1234)
      }
    }
    transition init : initial -> notWriting {
    }
    transition getWriting : notWriting -> notWriting {
      guard {
        writeRequestSent == false
      }
      action {
        send con.acquireWrite()
        writeRequestSent := true
      }
    }
    transition goWriting : notWriting -> writing {
      triggeredBy con.acknowledge()
      action {
        writeRequestSent := false
        log logger with "goWriting"
        inform timer in timeoutPeriod
      }
    }
    transition backToNotWriting : writing -> notWriting {
      triggeredBy timeout timer
      action {
        send con.releaseWrite()
      }
    }
  }
}
capsule Database {
  external port ~read : ReadAndWriteThings
  external port ~write : ReadAndWriteThings
  logPort logger
  attribute thing := 1
  stateMachine {
    state single
    transition init : initial -> single {
    }
    transition readData : single -> single {
      triggeredBy read.read()
      action {
        log logger with "read: " ^ thing
      }
    }
    transition writeData : single -> single {
      triggeredBy write.write(data)
      action {
        log logger with "original: " ^ thing
        thing := data
        log logger with "modified: " ^ thing
```

```
            }
          }
        }
      }
      protocol ControllerRW {
        incoming {
          acquireRead()
          releaseRead()
          acquireWrite()
          releaseWrite()
        }
        outgoing {
          acknowledge()
        }
      }
      protocol ReadAndWriteThings {
        incoming {
        }
        outgoing {
          read()
          write(data)
        }
      }
    }
```