Model Mapping Using Formalism Extensions

Guy Caplat and Jean-Louis Sourrouille, *INSA of Lyon*

Transforming models from one formalism to another is key in OMG's Model Driven Architecture. Formalism extension plays an essential role in this process.

The Object Management Group's Model Driven Architecture[1] defines a system development approach that formally separates system specification from platform implementations—in *platform-independent models* and *platform-specific models*, respectively. According to MDA, software development involves a sequence of model mappings that transform an initial PIM to a final PSM that is precise enough for direct translation into an executable program.

A mapping is a set of rules and techniques for translating one model into another. Mappings range from minor model refinements to major transformations.

When the starting and final models are expressed in the same formalism, the mapping is said to be *intralanguage*; otherwise, it is *interlanguage*.[2] The MDA sequence of mappings from PIM to code—that is, from a semiformal to a formal language—includes an undefined number of intralanguage mappings and at least one interlanguage mapping. We focus here on interlanguage mapping, showing the central role of formalism extension mechanisms in managing the abstraction-level gap between languages as well as the platform-level details of specific implementations.

System modeling and formalisms

Figure 1 uses OMG's Unified Modeling Language (UML)[3] notation to illustrate the relationships among the main notions in system modeling. A *model* represents a system using a given *formalism*, or language—that is, a set of conventional signs or terms. A model always reflects a subjective viewpoint on a system: To model is to observe something according to a chosen theory and to express the observer viewpoint formally.

Primitives, semantics, and constraints

A formalism is based on a set of relevant *abstract primitives*, or notions. In system modeling, abstract primitives include notions such as class, inheritance, message, and constraint. *Concrete primitives* represent these notions in forms such as boxes and arrows; *semantics* specify their meanings: what it means to be "a class," for example. A notion's semantics are an abstraction that can be expressed in any language.

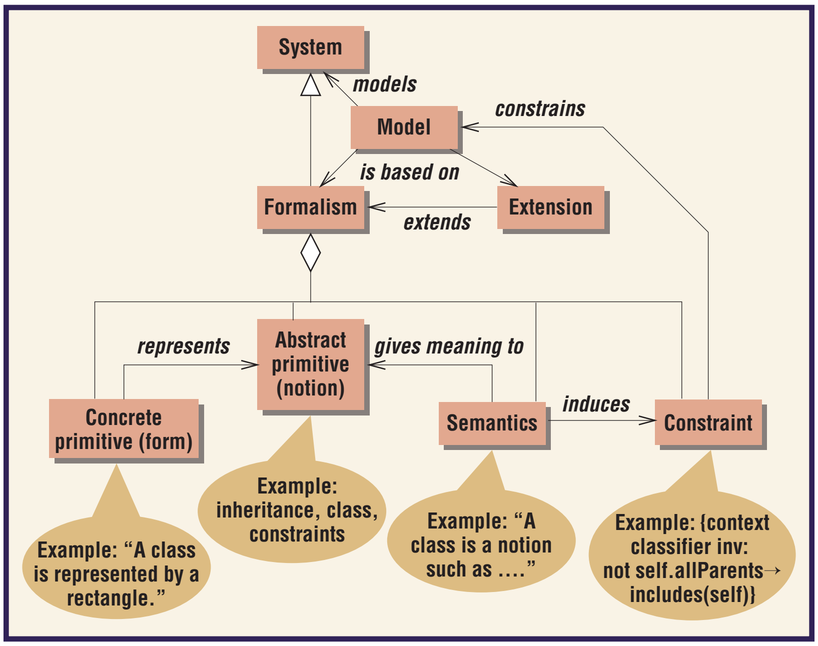

System
models
constrains
Model
is based on
Formalism
extends
Extension
represents
Abstract primitive (notion)
gives meaning to
Concrete primitive (form)
Semantics
induces
Constraint
Example: inheritance, class, constraints
Example: "A class is represented by a rectangle."
Example: "A class is a notion such as ...."
Example: {context classifier inv: not self.allParents→includes(self)}

Figure 1. Basic model of relationships among the main notions in system modeling. The relationships are expressed in UML notation; the example constraint is expressed in OCL.

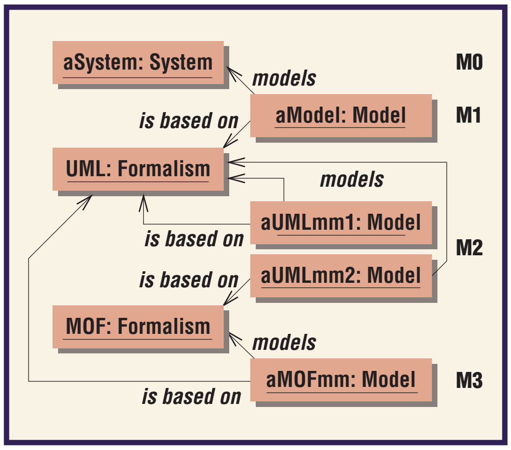


Figure 2. Four-layer UML formalism instantiating Figure 1's basic model. M0 is the modeled system layer; M1 describes the user models; M2 is the UML metamodel layer; and M3 is the meta-metamodeling layer, which defines metalanguages such as MOF for the M2 level.

Semantics induce *constraints*, such as the irreflexive inheritance relationship. Constraints apply to models, restricting the use of concrete primitives and ensuring the model's syntactical legitimacy. The constraints are given an external form in a language such as Object Constraint Language. Figure 1, for example, uses OCL to express the irreflexive constraint. A language's expressive power and computability determine whether constraints can be checked automatically or not.

An *extension* specifies new notions and defines their semantics using formalism primitives, just as the original notions are specified.

Metamodels

Any formalism reflects a viewpoint that determines a set of modeling primitives and their semantics, along with the orthodox way to use them. A formalism thus embodies principles for viewing and modeling any system.[4] Because a formalism has all the characteristics of a system, it can be modeled as such—in Figure 1's terms, Formalism inherits from System.

A formalism's model is usually called a *metamodel*. We use $m(s)/f$ to denote a model m of the system s in the formalism f, and we use $mm(f)/F$ to denote a metamodel of f expressed in F—that is, a model mm of the system f in the formalism F.

Notice that a model is a metamodel only when it models a formalism. Any metamodel mm of a formalism f describes the set of abstract notions in f, their syntax, semantics, and constraints. For example, the metamodel $mm_1(f_1)/F$ defines and expresses in F how to make and give sense to any model $m_k(s_k)/f_1$.

In turn, F should be modeled. Although the number of modeling layers is theoretically unlimited, UML is based on a four-layer architecture labeled M0 to M3. Figure 2 shows four layers instantiating the model in Figure 1:

- M0 defines the modeled system, *aSystem*.
- M1 defines *aSystem*'s UML model, *aModel*.
- M2 defines two metamodels of UML—specifically, *aUMLmm1(UML)/UML*, expressed in UML, and *aUMLmm2(UML)/MOF*, expressed in OMG's Meta-Object Facility.[5]
- M3 defines a metamodel *aMOFmm(MOF)/UML* of the MOF expressed in UML.

Because UML provides all the elements needed to describe a language, UML models exist for both the MOF and the UML itself.

Model mapping

A model mapping is a transformation $m_1(s)/f_1 \rightarrow m_2(s)/f_2$, shortened $m_1/f_1 \rightarrow m_2/f_2$.

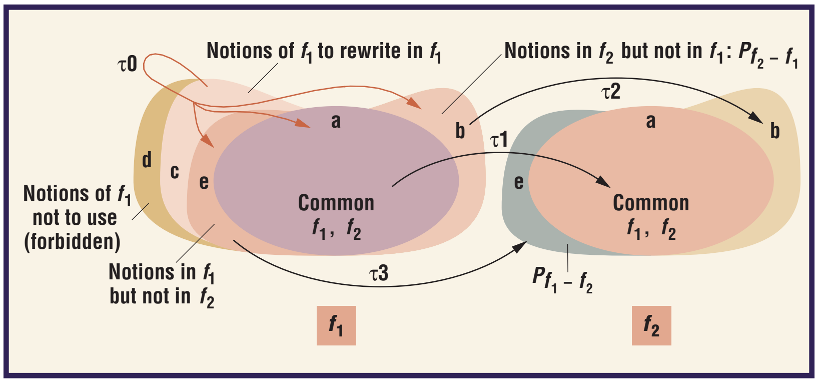

τ0
Notions of f_1 to rewrite in f_1
Notions in f_2 but not in f_1: $P_{f_2} - f_1$
τ2
a
b
d
c
e
τ1
a
b
e
Notions of f_1 not to use (forbidden)
Common f_1, f_2
Common f_1, f_2
Notions in f_1 but not in f_2
τ3
$P_{f_1} - f_2$
f_1
f_2

Figure 3. Translating m/f₁ into m/f₂. f₁ requires extension to add notions from f₂, and f₂ requires extension to accept some notions of f₁. In addition, f₁ includes extensions independent of the mapping to f₂, which are not translated.

In intralanguage mappings, where $f_1 = f_2$, an ordered suite of modeling actions describes the transformation $m_1/f \rightarrow m_2/f$. These modeling actions are of few types: `Create aModelElement`, `Delete aModelElement`, and `AssignValueTo aModelElement`, where `aModelElement` is an instance of any primitive notion of *f*.

In interlanguage mappings, the source and target formalisms are different—for instance, m_1/UML → m_2/Java or m_2/C++ → m_3/assembly code. The mapping effort depends on the semantic distance between the two formalisms' notions. Interlanguage mappings are more complex than intralanguage mappings: They must not only describe the modeling actions to go from m_1 to m_2 but also compare the two formalisms to determine the extent to which the semantics of f_1 transfer to f_2. For example, they must translate UML asynchronous messages to C++.

To compare the notions of f_1 and f_2 requires an M2 metalevel description of $mm_1(f_1)/F$ and $mm_2(f_2)/F$ in a common formalism *F*.6 MOF is a good candidate for this task because it provides operations on several abstraction levels. For example, it allows users to invoke an update operation on M1-level objects and to access properties such as an object's M2-level metaobject.

Comparing formalisms

Obviously, when f_1 includes primitive notions that f_2 primitives cannot express, some mappings $m(s_1)/f_1 \rightarrow m(s_2)/f_2$ are not possible. This problem doesn't occur when f_1 and f_2 are defined informally or semiformally. Nor does it come up when a programming language is translated to executable code insofar as programming languages are designed for such translation. On the other hand, the problem can arise when a semantic gap exists between two formalisms—for example, differences in their granularity or expressive power.

This semantic gap raises the question of a formalism's strength to build accurate models of any system. Should the semantics of the formalism's notions reproduce the semantics of the "categories of things" in the modeled domains? This position would require UML, for example, to include notions of Class and Inheritance because they correspond to notions that exist in "real life" modeled systems. Clearly, this position is unrealistic.

A language is both a tool for expressing a theory, or interpretation, of a given system and a filter that acts on the perception of the system. It is because natural languages provide the notions of *concept* (or its equivalent) and *specialization* that we have learned to perceive similarities between objects and organize them into taxonomies. Thanks to natural-language primitives, the world appears to be structured, and by a boomerang effect, modeling formalisms provide sets of primitive notions that are similar to each other, exhibiting only slight semantic variations.

Extensions play an essential role in dealing with the differences between formalisms.

Extending formalisms

The vocabulary needed to model a system depends on the domain: Real-time systems differ from information systems, and so on. A formalism might require new primitives or structures to increase its expressive power for different domains. There are two approaches to adding primitives:

- accepting the principle of modifying the set of primitives, and
- including the notion of extension as an abstract primitive.

The former approach is heavy. It requires a new release of the language to modify the metamodel as well as a consensus on the new version's abstract primitives, concrete forms, and semantics. Moreover, the formalism will continually grow.

UML follows the latter approach and provides extension mechanisms: notions (*stereo-*

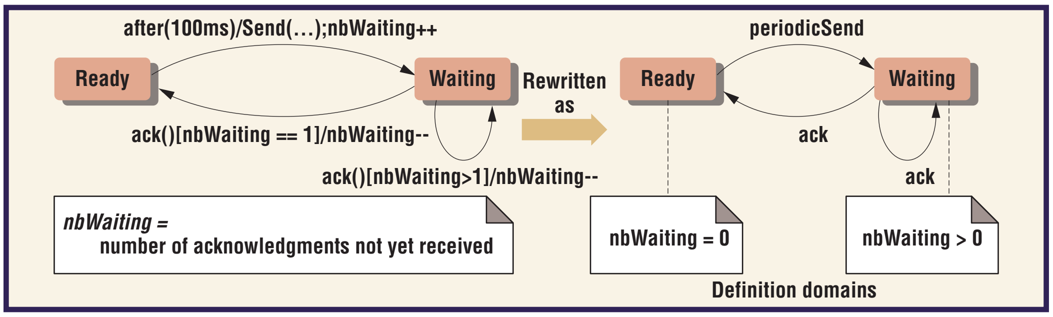

after(100ms)/Send(...);nbWaiting++
Ready
Waiting
ack()[nbWaiting == 1]/nbWaiting--
ack()[nbWaiting>1]/nbWaiting--
nbWaiting = number of acknowledgments not yet received
Rewritten as
periodicSend
Ready
Waiting
ack
ack
nbWaiting = 0
nbWaiting > 0
Definition domains

Figure 4. ARTO state diagram in UML rewritten and tagged for translation to C++.

type) based on existing primitives (*metaclass*), properties (*taggedValue*), and constraints (*constraint*). This way, UML deals with light, local, or not yet commonly accepted modifications. UML defines how to make extensions; the extensions, stored in *profiles*, define new primitives and their semantics.[7] For example, the *UML Profile for Schedulability, Performance, and Time* defines primitives in the real-time domain.

For model mapping, formalism extensibility is both a burden and an advantage. It's a burden insofar as developers who add extensions f_{ext} to f must precisely define their semantics to allow $m_1(s)/(f_1 + f_{ext}) \rightarrow m_2(s)/f_2$. On the other hand, extensibility offers a chance to enhance expressive power: Let profile $P_{f_1-f_2}$ be UML profile extensions that define the f_1 primitives not existing in f_2. The formalism $f_2 + P_{f_1-f_2}$ has the expressive power of f_1, and the mapping $m_1/f_1 \rightarrow m_2/(f_2 + P_{f_1-f_2})$ becomes straightforward.

Each formalism in the $m_1/f_1 \rightarrow m_2/f_2$ mapping can provide extension mechanisms, and both are useful in the MDA approach. PIMs are described in UML, then adorned and tagged with ad hoc elements that are defined in profiles—like marking in MDA.[8] For example, the CORBA profile defines stereotypes—CORBAInterface, CORBAValue, and so on—that mark classes to indicate what they represent.[7] The UML-to-Ptolemy translation follows the same path.[9]

Using extensions to map between formalisms

Figure 3 summarizes the general cases for translating models from one formalism to another, $m/f_1 \rightarrow m/f_2$.

Translation $\tau1$ addresses notions common to f_1 and f_2 (a, in Figure 3). Without language extension, direct translation would be limited to these notions.

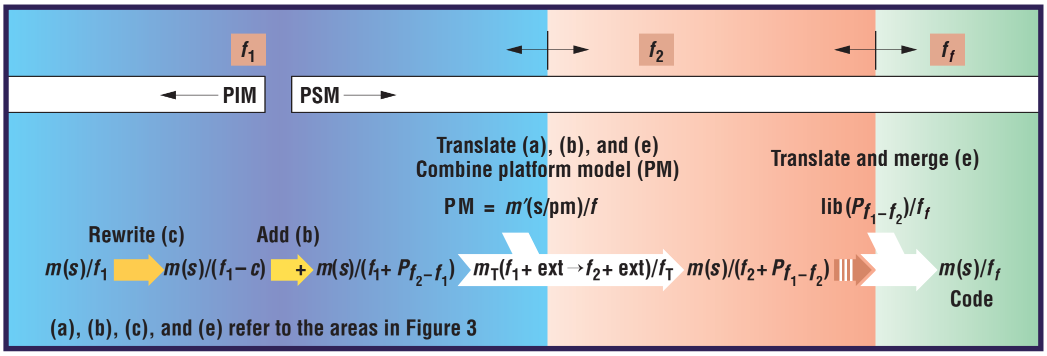
Translation $\tau2$ addresses notions in f_2 but not in f_1. The translation requires an extension of f_1 to capture the notions needed from f_2 (b, in Figure 3). Figure 4 illustrates the extension of a UML model (f_1) for a $\tau2$ translation into C++ (f_2) for the Adaptable Real-Time Object (ARTO) framework,[10] which we developed for optimizing the quality of service (QoS) of real-time software systems. C++ includes many notions and expressions not in UML; for instance, the *main* function to manage multithreaded programs, which is automatically created during translation. Figure 4 shows code in UML notes—f_1 extensions with notions from f_2, such as `nbWaiting = 0`—to compute the object's current state from the states' definition domains.

For notions in f_1 but not in f_2 (c, d, and e in Figure 3), we distinguish three cases.

One is rewriting f_1 ($\tau0$) notions in f_1, $m/f_1 \rightarrow m/f_1$ (c, in Figure 3). For example, an n-ary relationship is translated into n binary relations without changing the semantics. Figure 4 illustrates this case with regard to the general form of a UML transition: `event[condition]/action`. To hide actions in the state diagram, the ARTO framework calls the object method `m()` when the object receives the event `m`—as in the UML protocols. When the object receives `ack`, the ARTO framework calls the method `ack(){nbWaiting – ;}` to execute the code related to `action`. ARTO doesn't provide synchronous messaging: rewriting a synchronous message in an asynchronous message followed by a wait might be suitable.

When rewriting is impossible or when it requires model changes that go too deep to warrant it, notions of f_1 that have no translation into f_2 are forbidden (d, in Figure 3); for instance, multiple inheritance is forbidden when the target language is Java.

The third case is to define an f_2 extension that includes the needed f_1 notions (e, in Figure 3), then to translate f_1 into terms of the f_2 extension ($\tau3$). In ARTO, this case applies to



f_1
f_2
f_f
PIM
PSM
Translate (a), (b), and (e)
Combine platform model (PM)
PM = $m'(s/pm)/f$
Translate and merge (e)
lib$(P_{f_1-f_2})/f_f$
Rewrite (c)
Add (b)
$m(s)/f_1$
$m(s)/(f_1 - c)$
+
$m(s)/(f_1 + P_{f_2-f_1})$
$m_T(f_1 + \text{ext} \rightarrow f_2 + \text{ext})/f_T$
$m(s)/(f_2 + P_{f_1-f_2})$
$m(s)/f_f$
Code
(a), (b), (c), and (e) refer to the areas in Figure 3

Figure 5. Development process example: *m*(*s*) is the system model at various states of development.

the numerous UML notions that C++ does not provide, such as active objects and asynchronous messaging, but above all to the extensions related to QoS, such as message deadline, worst-case execution time (WCET), or message queue size.

The τ3 translation mode shifts translation difficulties to later in the development process—specifically, to code generation. For instance, generating code from UML models requires a C++ library to implement the asynchronous messages.

How difficult are these translations? *τ*1, *τ*2, and *τ*3, which deal with a, b, and e in Figure 3, are easy to automate because the translation preserves the notions. *τ*0 is a rewriting in the same language that can be partly automated; moreover, software developers can avoid it by not using notions of a. All these translations are quite simple; the Figure 3 framework therefore shows a general and practical approach for model translation.

From PIM to code

Figure 5 illustrates the general development process for transforming the specification of a given target application's PIM, $m(s)/f_1$, into a program $m(s)/f_f$ written in an executable language f_f. Typically, a developer first describes the system in UML (f_1 in Figure 5). Then the developer or an automatic process translates the model into a programming language such as C++ (f_2 in Figure 5, with $f_2 \neq f_1$), and the compiler translates the program into executable code (f_f). (Figure 5 doesn't show the earlier transformations that refined the PIM by adding elements related to the application domain.)

Ideally, once a model $m_T(f_1 \rightarrow f_2)/f_T$ describes the actions to translate f_1 notions into f_2 notions in a transformation language f_T, such as QVT,[11] a generic metatranslator can build a specific translator. Generally, translators such as compilers are not language-parameterized, and the developer must code actions directly. On the other hand, $m/(f_1 + P_{f_2-f_1}) \rightarrow m/(f_2 + P_{f_1-f_2})$ requires a minimal description of the target platform model $m'(s/pm)/f$. This description includes application-independent data, such as the target language, and various application-dependent parameters. In ARTO, for example, such parameters include methods' WCET and active objects' thread assignment.

A developer could set values directly in the PSM, but it is preferable to set parameters that can be replaced by values at translation time. For instance, instead of marking the operation op_i with a 10-millisecond WCET in the PSM, op_i is associated with a parameter d_op_i. The developer will define the value of d_op_i in $m'(s/pm)/f$ according to the platform, and the compiler will use the correct value during translation (*f* is translator specific).

Therefore, several models are combined at translation time to build the target platform model. Mixing models isn't easy even when they are expressed in the same formalism: Data from the two models must be linked (for example, d_op_i with its value); moreover, the links must be preserved throughout successive development iterations. The same problem arises in aspect-oriented software development where an aspect weaver gathers the viewpoints on a system, or aspects, to generate the target code.[12,13] Marking the PIM/PSM (Add (b) in Figure 5) using f_1 extensions and annotations[14] makes the translation into f_2 easier. The extension mechanisms have great practical interest: $P_{f_2-f_1}$ anticipates the mapping from f_1 to f_2, by using f_2 primitives as if they are f_1's, while $P_{f_1-f_2}$ extends f_2 primitives with f_1 primitives that do not exist in f_2.

By definition, as soon as the PIM integrates platform-specific information, it becomes a PSM. Elements such as operating system API, hardware compression, or operations' WCET are related to the target platform. Assuming that f_2 is the mapping's target language, the transformation of PIM to PSM begins when model elements depending on f_2 are added using the $P_{f_2-f_1}$ extension. Hence, the real mapping issue is model translation between different formalisms rather than model transformation from PIM to PSM.

Adding specific data to the PIM might seem paradoxical, but it works because the description is captured separately in extended f_1. Many CASE tools, such as I-Logix's Rhapsody and IBM Rational Rose RT, follow this path to generate code from extended UML, keeping rather good platform independency. Unfortunately, tool extensions aren't compatible, and models are tool dependent. This makes a strong point in favor of common profiles.

The development process ends by mapping $m(s)/f_2$ to $m(s)/f_f$. While programming languages cannot add new notions, they can simulate extensions through external libraries. The libraries implement specific f_2 notions absent from f_f. A thread management library is one example.

To sum up, the PIM-to-code process involves defining the profiles $P_{f_2-f_1}$ and $P_{f_1-f_2}$, then translating from $f_1 + P_{f_2-f_1}$ into $f_2 + P_{f_1-f_2}$ while combining $m/(f_1 + P_{f_2-f_1})$ and $m'(s/pm)/f$, and finally implementing in f_f the libraries that correspond to f_2 extensions.

Application to ARTO tool development

We used this process to develop a companion tool for the ARTO framework. The modeling language consisted of UML plus extensions to manage the QoS ($\mathrm{UML}_{\mathrm{ext}}$) and C++ for expressions that UML cannot describe ($P_{\mathrm{C{+}{+}-UML}_{\mathrm{ext}}}$). An iteration of the development process includes two mappings:

$$m(s)/(\mathrm{UML}_{\mathrm{ext}} + P_{\mathrm{C{+}{+}-UML}_{\mathrm{ext}}}) \xrightarrow{m(s\,/\,pm)}$$
$$m(s)/(\mathrm{C{+}{+}} + P_{\mathrm{UML}_{\mathrm{ext}}-\mathrm{C{+}{+}}}) \xrightarrow{lib(P_{\mathrm{UML}_{\mathrm{ext}}-\mathrm{C{+}{+}}})}$$
$$m(s)/code$$

We used the Rational Rose tool to handle UML and added functions to manage extensions and generate code. The ARTO specification wasn't complete at the beginning of the tool development project, so we first implemented and validated a library to supply all the necessary QoS notions and behavior. Then we defined the UML extensions, using stereotypes to introduce new notions such as the ActiveObject and MetaObject classes. System description requires many tagged values—method duration, message choice policy, object thread number, exceptions, and so on—too numerous to use the UML form `{name = value}`. We therefore developed an add-in with a dialog for inputting values in a more user-friendly way. The add-in verifies constraints related to values. Some model modifications imply changes in related parts; the add-in performs these automatically, thus relieving the developer of an unnecessary burden.

We based code generation on a general mechanism that proved useful. The ARTO framework includes numerous "holes" that developers can fill in many ways. Default behavior is inherited while specific behavior for each kind of object is handled by a metaobject. To effectively separate QoS-related code from system function code, almost all the generated code is in the metaobject class.

The Rational Rose tool generates only the code of object classes. The ARTO tool generates the code of metaobject classes, translates state diagrams to process events and ensure that transitions are licit. It also translates extensions into C++, replacing marks within template files and building data structures; and it generates additional expressions such as object structure (creation and links) from a collaboration diagram. Platform-dependent values such as method duration are given unique names that the compiler replaces by their value, including the corresponding macrodefinitions. To complete the program, the developer refines the PSM by adding C++ code through a standard development tool.

The ARTO tool doesn't provide the reverse model mapping from C++ to UML, except for regular classes and their relationships. Hence, all other modifications take place in the model. Developers should refrain from modifying generated code; modifications require knowing the implementation specifications while the ARTO tool gives only the extension specifications.

What role do extension mechanisms play in this process? They eliminate the need to change the original and target languages. As a result, developers can use ordinary tools to model systems and translate common notions. Only extensions require unique mappings.

Developers can use ordinary tools to model systems and translate common notions. Only extensions require unique mappings.

$$m_D(s_k\_\text{DomainConstraint})/\text{Sherlock} \xrightarrow{mm(\text{UML/Sherlock})} \text{CC/Sherlock}$$
$$m_D/(\text{UML} + P_{\text{Sherlock-UML}}) \xrightarrow{m_T(\text{UML} \to \text{Sherlock})} m_D/\text{Sherlock}$$
$$m_S(s_k\_\text{Structure})/\text{UML} \xrightarrow{m_T(\text{UML} \to \text{Sherlock})} m_S(s_k\_\text{Structure})/\text{Sherlock}$$

Figure 6. Mapping of generic Constraint Checker metamodel in UML to domain-specific constraints in the PSM rule base.

Translating new notions into the target language is the greater part of the work, but it follows a clear and organized implementation approach (see Figure 3). In particular, programming language extensions (libraries) are designed based on notions and properties that provide component boundaries and content automatically.

Application to Constraint Checker

We also developed an expert system, Constraint Checker,[2,15] to check $m_k(s_k)$/UML models. CC takes m_k contents as input and returns warnings about model inconsistencies and incompleteness.

We have written a UML metamodel subset (Class diagram) into Sherlock, which acts as the f_2 formalism in Figure 5. Following the model in Figure 1, the metamodel *mm*(UML)/Sherlock is composed of *mm*(UMLAbstractPrimitive)/Sherlock, which defines the UML abstract primitives, and *mm*(UMLConstraint)/Sherlock, which defines their semantics—for example, irreflexive(Inheritance). *mm*(UML)/Sherlock constitutes a generic frame for checking any UML class diagram.

At the metamodel level, $m_k(s_k)$ models both the structure of s_k and the specific domain constraints that apply. The former is represented by $m_S(s_k\_\text{Structure})$/UML; the latter are represented in UML + Extensions (f_1) by $m_D(s_k\_\text{Domain-Constraint})/(\text{UML} + P_{\text{Sherlock–UML}})$. Concretely, $P_{\text{Sherlock–UML}}$ defines stereotypes and tags to express constraints in a Sherlock-like form.

CC includes an inference engine and two rule bases: the PSM *mm*(UMLConstraint)/Sherlock and the domain-specific model $m_D(s_k\_\text{Domain-Constraint})$/Sherlock. Before checking a model, CC performs the mapping $m_D(s_k\_\text{DomainConstraint})/(\text{UML} + P_{\text{Sherlock–UML}}) \to m_D(s_k\_\text{DomainConstraint})$/Sherlock. These bases are run on $m_S(s_k\_\text{Structure})$/Sherlock mapped from $m_S(s_k\_\text{Structure})$/UML, as shown in Figure 6.

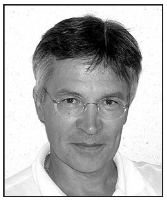
In brief, this approach customizes the generic CC made of *mm*(UML)/Sherlock with the domain-specific constraints contained in $m_D(s_k\_\text{DomainConstraint})$/Sherlock (see “combine platform model” in Figure 5). Then, the resulting CC runs the model $m_S(s_k\_\text{Structure})$/Sherlock mapped from $m_k(s_k)$/UML and gives back specific warnings or advice about $m_k(s_k)$/UML.

This example shows the use of extension mechanisms to add a UML profile specialized in the modeling of declarative programming. Actually, $P_{\text{Sherlock–UML}}$ defines a stereotype of production rules with several tags that allow defining *let*, *if*, and *then* parts of any standard production rule. Such a profile is a means to complete UML descriptions with inferential reasoning and to develop knowledge-based applications.

Gathered into UML profiles, extensions simplify translation by formalizing target-specific notions. Moreover, they provide a clear and organized implementation approach for translator development. Because the PIM becomes a PSM before translation into the target language, the real MDA issue is mapping from one language to another rather than from a PIM to a PSM.

The importance of UML profiles in MDA drives our current work to define an approach to building profiles in UML 2.0. Our goal is to describe a profile P as input to a knowledge base for automatically verifying that models expressed in UML + *P* conform to *P*. The main

About the Authors



Guy Caplat is an associate professor in the Department of Information Technology and Computer Engineering at the Institut National des Sciences Appliquées (INSA) of Lyon, France. His research interests focus on metamodeling and knowledge modeling. He received his PhD in computer sciences from the University of Lyon-I. Contact him at IF Bat. B. Pascal—F69621 Villeurbanne Cedex, France; guy.caplat@insa-lyon.fr.

Jean-Louis Sourrouille is a professor in the Department of Information Technology and Computer Engineering at INSA of Lyon. His research interests focus on quality of service and behavior adaptation in distributed systems, real-time objects, and software development using UML, including code generation and model consistency. He received his PhD in computer science from the University of Lyon-I. Contact him at INSA, PRISMa Bat. B. Pascal, F69621 Villeurbanne Cedex, France; Jean-Louis.Sourrouille@insa-lyon.fr.

issues are to formalize a design method and transform the profile description semantics into a set of CC constraints. When *P* specifies the rules of well-formedness, CC can use the rules reflexively to enable self-checking.

References

1. *Model Driven Architecture*, doc. no. ormsc/2001-07-01, draft adopted specification by OMG, 9 July 2001.
2. G. Caplat and J.L. Sourrouille, "Model Mapping in MDA," paper presented at Workshop in Software Model Eng. (WISME), 2002; www.metamodel.com/wisme-2002.
3. Available specification, *Unified Modeling Language*, v. 1.5, formal/2003-03-01, OMG, 3 Mar. 2003.
4. G. Caplat, *Modélisation Cognitive* [Cognitive Modeling], PPUR Editions, 2002.
5. Available specification, *Meta-Object Facility*, v. 1.4, formal/0501-02, OMG, Jan. 2001.
6. T. Clark, A. Evans, and R. France, "OO Theories for Model Driven Architecture," *Proc. Object-Oriented Information Systems Workshop Model-Driven Approaches to Software Development*, LNCS 2426, Springer-Verlag, 2002, pp. 235–244.
7. Available specification, *UML Profile for CORBA*, v. 1.0, formal/02-04-02, OMG, Apr. 2002.
8. *MDA Guide*, v. 1.0.1, omg/2003-06-01, OMG, 12 June 2003.
9. V. Arnould, A. Kramer, and F. Madiot, "From UML to Ptolemy II Simulation: A Model Transformation," paper presented at European Conf. Object-Oriented Programming (ECOOP) 2002 Workshop Integration and Transformation of UML Models (WITUML); http://ctp.di.fct.unl.pt/~ja/witumlagenda.htm.
10. J.L. Contreras and J.L. Sourrouille, "A Framework for QoS Management," *Proc. 39th Int'l Conf. and Exhibition Technology of Object-Oriented Languages and Systems* (TOOLS 01), IEEE Press, 2001, pp. 183–193.
11. Request for proposal, *MOF 2.0 Query/Views/Transformations RFP*, ad/2002-04-10, OMG, 28 Oct. 2002.
12. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming* (ECOOP 97), LNCS 1241, Springer-Verlag, 1997, pp. 220–242.
13. F. Mekerke, G. Georg, and R. France, "Tool Support for Aspect-Oriented Design," *Proc. Object-Oriented Information Systems Workshop Model-Driven Approaches to Software Development*, LNCS 2426, Springer-Verlag, 2002, pp. 280–289.
14. S.J. Mellor et al., "Model-Driven Architecture," *Proc. OOIS Workshop Model-Driven Approaches to Software Development*, LNCS 2426, Springer-Verlag, 2002, pp. 290–297.
15. J.L. Sourrouille and G. Caplat, "Constraint Checking in UML Modeling," *Proc. 14th Int'l Conf. Software Engineering and Knowledge Engineering* (SEKE 02), ACM Press, 2002, pp. 217–224.

ADVERTISER / PRODUCT INDEX

MARCH/APRIL 2005

Advertiser / Product	Page Number
Addison-Wesley Professional	89
Classified Advertising	**10**
IEEE Computer Society CSDP	**1**
Prentice Hall	88, 90
Software Development Conference 2005	**Cover 4**
Software Engineering Conference 2005	**Cover 3**
Systems & Software Technology Conference 2005	**Cover 2**

Boldface denotes advertisements in this issue.

Advertising Personnel

Marion Delaney
IEEE Media, Advertising Director
Phone: +1 212 419 7766
Fax: +1 212 419 7589
Email: md.ieeemedia@ieee.org

Marian Anderson
Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: manderson@computer.org

Sandy Brown
IEEE Computer Society,
Business Development Manager
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sb.ieeemedia@ieee.org

Advertising Sales Representatives

Mid Atlantic (product/recruitment)
Dawn Becker
Phone: +1 732 772 0160
Fax: +1 732 772 0161
Email: db.ieeemedia@ieee.org

New England (product)
Jody Estabrook
Phone: +1 978 244 0192
Fax: +1 978 244 0103
Email: je.ieeemedia@ieee.org

New England (recruitment)
Robert Zwick
Phone: +1 212 419 7765
Fax: +1 212 419 7570
Email: r.zwick@ieee.org

Southeast (product)
Bob Doran
Phone: +1 770 587 9421
Fax: +1 770 587 9501
Email: bd.ieeemedia@ieee.org

Southern CA (product)
Marshall Rubin
Phone: +1 818 888 2407
Fax: +1 818 888 4907
Email: mr.ieeemedia@ieee.org

Southeast (recruitment)
Thomas M. Flynn
Phone: +1 770 645 2944
Fax: +1 770 993 4423
Email: flynntom@mindspring.com

Midwest/Southwest (recruitment)
Darcy Giovingo
Phone: +1 847 498-4520
Fax: +1 847 498-5911
Email: dg.ieeemedia@ieee.org

Midwest (product)
Dave Jones
Phone: +1 708 442 5633
Fax: +1 708 442 7620
Email: dj.ieeemedia@ieee.org

Will Hamilton
Phone: +1 269 381 2156
Fax: +1 269 381 2556
Email: wh.ieeemedia@ieee.org

Joe DiNardo
Phone: +1 440 248 2456
Fax: +1 440 248 2594
Email: jd.ieeemedia@ieee.org

Southwest (product)
Josh Mayer
Phone: +1 972 423 5507
Fax: +1 972 423 6858
Email: josh.mayer@wageneckassociates.com

Northwest (product)
Peter D. Scott
Phone: +1 415 421 7950
Fax: +1 415 398 4156
Email:
peterd@pscottassoc.com

Northwest/Southern CA (recruitment)
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Connecticut (product)
Stan Greenfield
Phone: +1 203 938 2418
Fax: +1 203 938 3211
Email: greenco@optonline.net

Japan (product/recruitment)
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Europe (product/recruitment)
Hilary Turnbull
Phone: +44 1875 825700
Fax: +44 1875 825701
Email:
impress@impressmedia.com

IEEE Software
IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314
USA
Phone: +1 714 821 8380
Fax: +1 714 821 4010
www.computer.org
advertising@computer.org