On the Execution of ODP Computational Specifications

José Raúl Romero
Dpt. Informática y Análisis Numérico
University of Córdoba
jrromero@uco.es

Antonio Vallecillo
Dpt. Lenguajes y Ciencias de la Computación
University of Málaga
av@lcc.uma.es

Abstract

The ODP computational viewpoint allows the description of the functional decomposition of a system and its environment in terms of configurations of objects that interact at interfaces. RM-ODP does not prescribe any concrete notation for describing this viewpoint, which hinders the development and use of tools for writing, analyzing and executing ODP computational specifications. Thus, several authors have proposed either formal or visual languages to describe this ODP viewpoint. However, the former notations are complex and lack industrial support, while the latter do not count with proper tools for executing and analyzing the specification produced. In this paper, we explore the use of model transformation techniques to establish a connection between UML models and Maude formal specifications of the ODP computational viewpoint, in order to obtain the best of both worlds.

1 Introduction

As software technology becomes a core part of business enterprises in all market sectors, customers demand more flexible enterprise systems. This demand coincides with the increasing use of personal computers and devices, and today's easy access to local and global networks, which together provide an excellent infrastructure for building open distributed systems. The problem is that these heterogeneous and distributed systems are inherently much more complex to specify, develop and maintain than classical, homogeneous, centralized systems. As a response to these needs, ISO, IEC and ITU-T started working on a joint standardization effort under the heading of *Open Distributed Processing* (ODP). Their goal was to define a reference model to integrate a wide range of ODP standards for distributed systems and maintain consistency among them. The *Reference Model of Open Distributed Processing* (RM-ODP) provides the coordination framework for ODP standards, creating an infrastructure within which support of distribution, interworking and portability can be integrated.

However, there are some issues that may jeopardize the wide diffusion and use of ODP. For instance, RM-ODP does not prescribe any specific notation to represent its concepts and viewpoint languages, which hampers the development of tools for writing and analyzing ODP specifications.

So far, most notations and tools for capturing and modeling business requirements tend to be either graphical or formal. Initially, most of the notations proposed as ODP specification languages were formal notations (e.g., Z, Object-Z, LOTOS, etc.) [6]. These notations provide precise and unambiguous system specifications and, more importantly, they also allow the rigorous analysis of the systems, with tools for reasoning about the specifications produced (i.e., quick-prototyping, model checking, or theorem proving). However, the complexity inherent to formal description techniques and the lack of industrial support has traditionally hindered their wide adoption and use.

On the contrary, graphical notations are intuitive and easy to learn and to use, and do not require users to have a deep and specialized knowledge of complex concepts, formalisms, and mechanisms. The wide adoption of UML by industry, the number of available UML tools, and the increasing interest for model-driven development and the MDA initiative [18], motivated ISO/IEC and ITU-T to launch a joint project in 2004, which aims to define the use of UML for ODP system specifications [13]. Thus, ODP modelers could use the UML notation for expressing their ODP specifications in a standard graphical way, and UML modelers could use the RM-ODP concepts and mechanisms to structure their UML system specifications. However, graphical notations do not usually count with tools for analyzing the specification produced, nor to provide executable systems that fully conform to these specifications.

The worlds of graphical and formal notations have usually lived apart. This paper tries to provide a bridge between them in the context of the ODP computational viewpoint. In particular, we show how model transformations can be effectively used for connecting the graphical ODP viewpoint specification with their corresponding formal specifications

in Maude, aiming at getting all the benefits that they individually provide. More precisely, Maude specifications are fully executable and can also be analyzed using the Maude formal toolkit. In addition, they can be progressively refined to obtain implementation of the system using commercial technologies, such as CORBA or WebServices [1, 2].

This paper is structured as follows. Section 2 provides a brief description of the ODP computational viewpoint. Then, sections 3 and 4 discuss how it can be specified in UML 2.0 and Maude, respectively. Section 5 shows how model transformations can be defined between both approaches. Then, some issues for discussion are raised in Section 6. Finally, Section 7 draws some conclusions and outlines some future work.

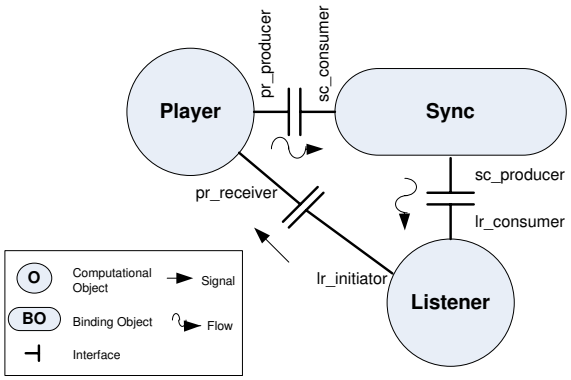

Figure 1. A simple example system

2 The ODP Computational Viewpoint

The computational viewpoint specification describes the basic functionality of the system, independently from distribution. A computational specification decomposes the system into a collection of objects performing their individual functions and interacting at well-defined interfaces.

The computational viewpoint language, as defined in the Reference Model of ODP [12], uses a set of basic concepts and structuring rules. Some of the most relevant concepts are described here.

ODP systems are modeled in terms of *objects*, each of which contains information and offers services. Computational objects are abstractions of entities that occur in the real world, in the same ODP system, or in other viewpoints.

Computational objects have *state* and can interact with their environment at *interfaces*, which are considered as an abstraction of the object's behavior.

RM-ODP prescribes three different types of interactions: *signals*, *operations* and *flows*. A signal is regarded as an atomic and simple action between computational objects. Operations are used to model object interactions as represented by most message passing object models. Finally, a flow represents an abstraction of a sequence of interactions between a *producer* and a *consumer*, whose semantics depend on the application domain. Operations and flows may be modeled in terms of signals.

Computational objects and interfaces can be specified by *templates*. In ODP, an <X> *template* is "the specification of the common features of a collection of <X>s in sufficient detail that an <X> can be instantiated using it". <X> can be anything that has a type. Thus, an interface of a computational object is usually specified by a *computational interface template*, which is an interface template for either a signal interface, a stream interface or an operation interface. A computational interface template comprises a signal, stream or operation *interface signature* as appropriate; a *behavior* specification; and an *environment contract* specification.

An *interface signature* consists of a name, a *causality* role (*producer*, *consumer*, etc.), and set of *interaction signatures*, according to the interaction type.

Following ODP, a computational specification describes the functional decomposition of an ODP system, in distribution transparent terms, as: (a) a configuration of computational objects (including binding objects); (b) the internal actions of those objects; (c) the interactions that occur among those objects; (d) environment contracts for those objects and their interfaces.

To illustrate our proposal with an example, as shown in Figure 1, let us consider a very simple multimedia ODP system, which consists of three computational objects: an audio Player, that emits flows to a Listener through a binding object, named Sync, which manages the synchronization of audio frames and delivers them to the connected listener. All these objects, which are instantiated from their corresponding computational object templates, interact at computational interfaces, which are also instantiated from their corresponding interface templates. Thus, pr_producer, sc_consumer, sc_producer and pr_producer are instantiated from IAudioStream (with different causalities). This template is used to declare a stream interface, through which the audio frames will flow. lr_initiator and pr_receiver are instantiated from the signal interface template IControl. Both interfaces have opposite causalities. In this case, IControl comprises two different signals: play, which begins emitting flows, and stop, which stops the flow.

3 Computational mapping between ODP and UML 2.0 specifications

As previously mentioned, RM-ODP does not prescribe any concrete syntax to represent the computational concepts.

However, both ODP modelers and UML experts would find extremely profitable to be able to use some graphical notation for drawing their ODP specifications. In this case, UML 2.0 provides the appropriate constructs to model the software architecture of large distributed systems (e.g., components, connectors, etc.). The language extension mechanisms have been greatly enhanced too, with the more precise definition of UML Profiles that allow the customization of UML constructs and semantics for specific application domains.

These new concepts, improvements and mechanisms of UML 2.0 constitute the basis of the UML profile proposed in [21, 22, 13] for modeling ODP computational viewpoint specifications. The definition of this profile comprises three main parts: (*a*) the computational viewpoint metamodel, which defines the semantics, properties and related elements to each metaclass; (*b*) the mappings between ODP concepts and UML elements; and (*c*) the profile, whose elements corresponds to the mappings to the specific ODP domain. A short description of this modeling approach follows.

Every *computational object* is instantiated from its *object template*, which is expressed as a UML component. UML components represent autonomous system units, that encapsulate state and behavior and interact with their environment in terms of provided and required interfaces. Then, a *computational object* is expressed as a UML component instance. *Binding objects*, as a particular case of *computational objects*, are modeled as UML component instances, too.

Computational objects interact with their environment at *interfaces*, which are instantiated from their *computational interface templates*. It is not possible to find a direct mapping for these concepts in UML 2.0. However, there are some UML elements that seem to provide the semantics (slightly adapted) for its representation. For instance, a *computational interface* can be mapped to a UML interaction point, i.e., a UML port at the instance level, as explained in [21]. An *interface template* can be modeled in UML by a port at the class level.

Every *computational interface template* comprises a *signature* (according to the kind of interface), the behavioral specification and its environment contracts. UML interfaces are used to represent signatures. A UML interface is expressed as a classifier that represents a declaration of a set of coherent public features and obligations and, therefore, it can be considered as the specification of a contract that must be fulfilled by any instance of a classifier that realizes the interface. Note that ODP allows the instantiation of interfaces. However, this is not possible in UML.

We can distinguish between the different kinds of ODP *computational interfaces* and *signatures* by using the proper UML extension mechanisms, i.e., stereotypes and tagged

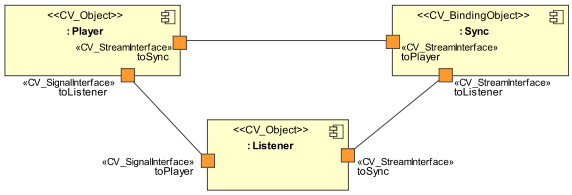

Figure 2. Computational configuration

definitions. These mechanisms are also useful for modeling *causalities* (at the object level) and *interactions*.

In ODP, *signals* are considered as the basic one-way communication mechanism from an *initiating object* to a *responding object*. As mentioned in Section 2, *operations* and *flows* can be handled in terms of *signals*. Both synchronous and asynchronous interactions are possible in UML and in ODP [15]. An ODP *signal* is expressed as a UML stereotyped message, which is the specification of the conveyance of information from one instance to another. In UML, a message can specify either the raising of a UML signal or the call of a UML operation.

In ODP, in order to specify an *interaction* we need to provide its *signature* and its *behavior*. An *interaction signature* will be represented by a UML reception. As mentioned in [20], "by declaring a reception associated to a given signal, a classifier specifies that its instances will be able to receive that signal, or a subtype thereof, and will respond to it with the designated behavior." In UML, these receptions will be defined inside the interface classifier. The behavior of *interactions* refers to the communication process between *computational objects*, which will be expressed in UML with behavioral diagrams [5]:

- Interaction models describe how messages are passed between objects and cause invocations of other behaviors.
- Activity models focus on the sequence, input/outputs and conditions for invoking other behaviors.
- Finally, state machine models show how events (e.g., signal events) cause changes to the object state and invoke other behaviors.

Using the UML Profile for the ODP Computational Viewpoint (UML4ODP-CV), Figure 2 depicts the computational configuration corresponding to the snapshot presented in Figure 1. At the computational object template level, the system could be modeled as shown in Figure 3.

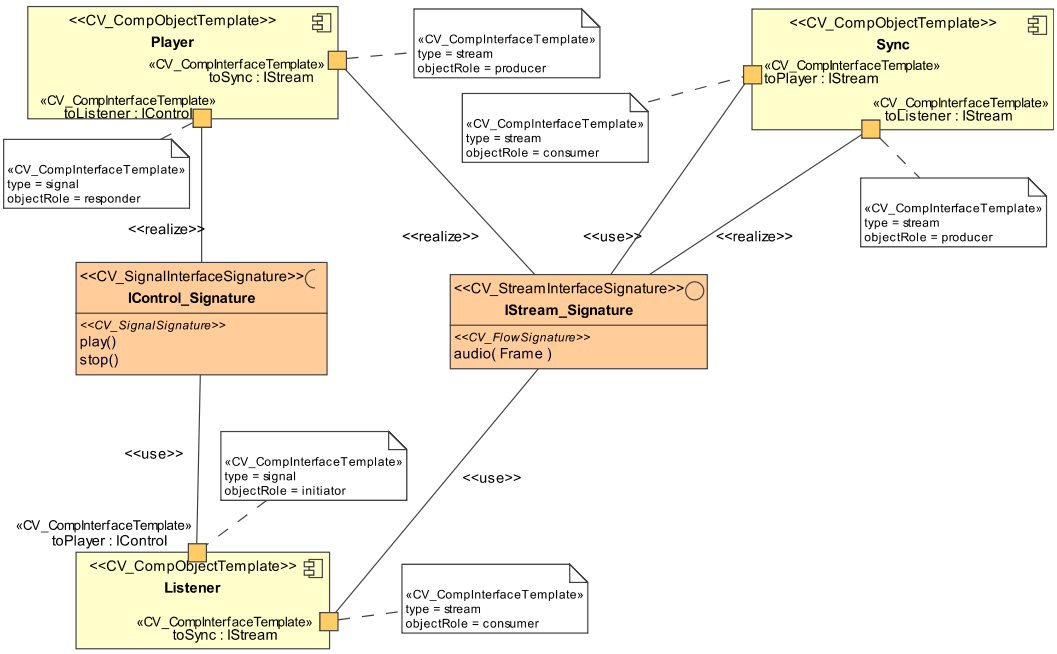

Figure 3. Computational templates diagram

4 Computational mapping to Maude specifications

Maude [8] is a high-level language and a high-performance interpreter and compiler in the OBJ [10] algebraic specification family that supports membership equational logic and rewriting logic specification and programming of systems. This kind of rewriting logic [16] can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification $(\sum, E)$, where $\sum$ is a signature of `sorts` (types) and `operations`, and E is a set of (conditional) `equational axioms`.

To specify the dynamics of a system in rewriting logic we make use of rewrite `rules` of the form $t \rightarrow t'$, where t and t' are $\sum$-terms. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

Maude also allows to specify object-oriented systems by using specific `modules`, in which classes and subclasses are declared. The objects instantiated from these classes can interact in a number of different ways, including message passing. `Messages` are declared in Maude by `msg` clauses, in which the syntax and arguments of the messages are defined.

In a concurrent object-oriented system, the concurrent state, which is called a `configuration`, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages.

A description of how the ODP computational specifications can be expressed in Maude can be found in [23]. In addition, rewriting logic allows us to write executable specifications, and we can also make use of a broad spectrum of formal methods and analysis tools—the Maude toolkit, which offers a model-checker, theorem prover, etc. A brief description of the mapping between Maude and the ODP computational specification follows.

Computational object templates will be represented by Maude `classes`, which are defined by a name and a set of attributes (of certain `sort`) that describe the state of the objects of the class. All *computational object templates* will inherit from class `CV-Object`, which describes the common features that any *computational object* exhibits. *Computational objects* will be then represented by Maude `objects`, instantiated from `CV-Object`.

ODP *signals* are represented by Maude `messages`. Every `message` has a name and a set of parameters of some type (`Sort`), as specified in its message declaration (`msg`). Thus, Maude `message` declaration will represent *signal signatures*, while `message instances` will represent concrete *signals*. *Operations* and *flows* can be ex-

pressed in terms of signals.

Interfaces will be modeled as Maude `objects`. The class to which any interface belongs will inherit from a general Maude `class` CV-Interface, which represents a generic *interface template*. These interface `objects` exist inside the local configuration of the Maude `objects` representing *computational objects*. Every *computational interface template* is declared in the scope of a Maude `module`, so `messages` and `operations` declared there are local to the *computational interface* it represents.

In the example described in Section 3, the computational templates Player and IControl shown in Figure 3 are specified in Maude as follows:

```
(omod COT-PLAYER is
   class Player | fps : Nat .
   subclass Player < CV-Object .
endom)

(omod CIT-ICONTROL is
   class IControl .
   subclass IControl < CV-Interface .
   msgs play stop : -> Msg .
endom)
```

The code above declares two Maude `object-oriented modules` (one per *computational template*) which define the `classes` that specify both the `computational object` (Player) and the `interface` (IControl). We have also included an `attribute` (fps) representing the *frames per second* rate at which the flow is emitted. *Object and interface templates* inherit from `CV-Object` and `CV-Interface`, respectively. According to the template diagram, the IControl interface comprises two *signals* (play and stop), which are specified in terms of their corresponding Maude `messages`.

Maude `configurations` are collections of `objects` and `messages` that are used represent the `state` of the system in a given moment in time. Thus, the UML diagram shown in Figure 2 can be naturally represented by a Maude `configuration`. The following Maude code shows one of the objects of such a configuration (namely, the Player with its two interfaces):

```
< O : Player | fps : 25,
  conf : < IC : IControl |
             uniqueId : toListener,
             objectRole : responder,
             interfaceType : signal >
         < IS : IStream |
             uniqueId : toSync,
             objectRole : producer,
             interfaceType : stream > >
```

Attribute `conf` represents the collection of interfaces and messages internally managed by the computational object.

5 Transforming computational specifications

Model transformation is the process of converting one model to another model of the same system. With this purpose, a transformation engine applies a series of transformation rules [7] on the source model to generate a target model.

According to [9], there are two different types of transformations. Firstly, model-to-code transformations are normally based on using templates, which comprise predefined parts of metacode text (e.g., FPL, XFramer, etc.). Secondly, model-to-model transformations translate from source to target models, which can be in accordance to the same or different meta-model. Most of the available MDA-based tools provides (restricted) functionalities for this second case, since model-to-code transformations can be seen as a particular case of the model-to-model ones but using a specific programming language as meta-model.

Our purpose is to obtain executable computational specifications from models represented in terms of the UML profile for ODP-CV. Thus, we will consider transformations between UML source models (written using the UML4ODP-CV profile) and target models conforming to the Maude-CV meta-model. As shown in Figure 4, transformation rules will allow us to specify the way both models are translated, conforming to the appropriate transformation meta-model. (In Section 6 we also discuss the possibility of using other alternatives.)

The next step is to translate the Maude model representing the computational specification to the specific program code that implements it (model-to-code transformation). This is a simple process, which can be performed with well-known techniques, such as XSLT, TCS or just another template-based language. As shown in Figure 5, these templates serve as a bridge between *technology spaces* [11]. In our case, the Maude code belongs to the "Syntax TS", where programming languages exist, which is characterized by its excellent degree of executability and formalization. UML is defined in the "MDA TS". In this paper we focus on the transformations performed in the scope of the MDA technology space.

There are many tools and languages proposed to perform the translation between models. However, most of them are still in a very early stage, so they are immature and difficult to use properly. In fact, OMG proposes QVT (*Query-View-Transformation*) [18] as the language to specify transformations between model conforming to MOF-based metamodels. QVT defines three different (but closely related) languages for specifying transformations using declarative and imperative styles. Black-box implementations of operations can also be used to allow reuse of existing algorithms or domain specific libraries in certain model transformations.

In particular, QVT Relations is a language to write

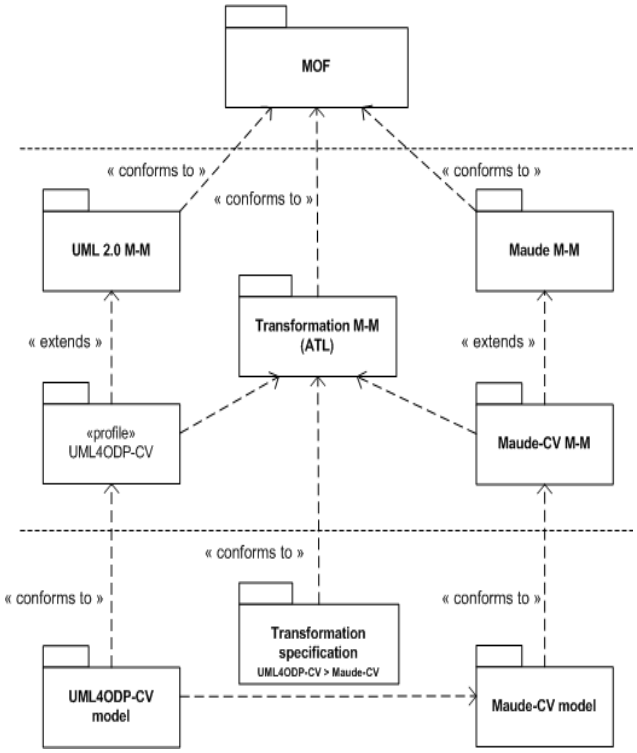

Figure 4. (Meta-)models involved in the transformation process

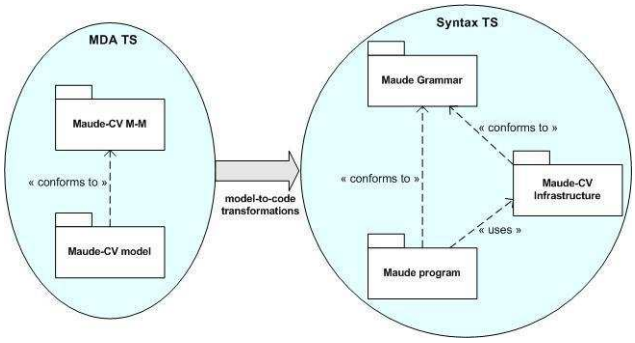

Figure 5. Bridging from MDA to Syntax TS

declarative specifications of the relationships between MOF models. The QVT Relations language supports object pattern matching, and implicitly creates trace classes and their instances to record what occurred during a transformation execution. Relations can assert that other relations also hold between particular model elements matched by their patterns.

QVT Relations allow for the following desirable execution scenarios [19]: (*a*) check-only transformations to verify that models are related in a specified way; (*b*) single direction and bi-directional transformations; (*c*) the ability to establish relationships between pre-existing models, whether developed manually, or through some other tool or mechanism; (*d*) incremental updates (in any direction) when one related model is changed after an initial execution; and (*e*) the ability to create as well as delete objects and values, while also being able to specify which objects and values must not be modified.

However, despite providing most of the desirable capabilities to specify the mapping from UML4ODP-CV to Maude-CV models, there is no actual tool-support for QVT, which hinders its application for real cases, since any QVT-based program should be translated first to another transformation language in order to be executed.

ATL (*ATLAS Transformation Language*) [4] is a model transformation language aligned with QVT [14] that provides numerous advantages to perform the mapping between metamodels, and in particular between the UML and Maude metamodels. Despite being still under development/research, ATL already begins to bear fruit and is probably the most widely used model transformation language.

ATL considers source and target models as strictly separated models. Moreover, ATL transformation rules can be specified in a declarative, imperative or hybrid manner. When specifying declarative rules, the left side of each rule defines a set of typed and constrained variables that characterize the appropriate elements of the source model(s). The right side contains a set of variables and some logic to bind values to the proper attributes between source and target model elements.

Apart from its benefits, ATL also presents some difficulties that could limit its applicability: (*a*) ATL is unidirectional, i.e., two individual mappings are required to implement transformations in both directions; (*b*) ATL programs do not consider change propagation, as QVT does; (*c*) dealing with models that use profiles is somehow cumbersome; (*d*) tool-compability (e.g., importing (meta-)models from external modeling tools) need to be improved; and (*e*) ATL is not as efficient as desirable for large transformation programs in its current stage.

In any case, within the scope of this paper we think that ATL provides a good approach to explore the possibility of executing Maude computational specifications using a transformation from their UML representation because of the advantages it provides. Some open issues will be later discussed in Section 6.

5.1 From UML4ODP-CV to Maude models

The next step is to determine which UML meta-model elements, used to represent the computational concepts,

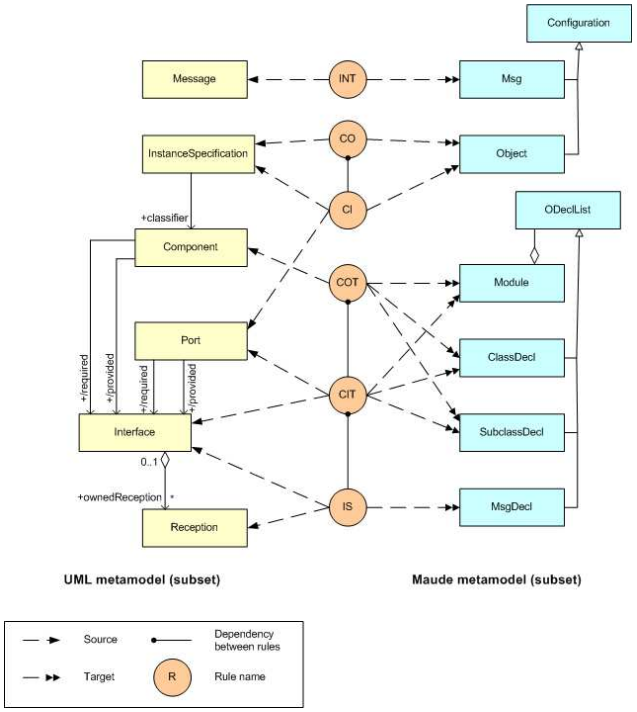

Figure 6. Structural mappings

should be mapped to Maude-CV meta-model elements. These correspondences are made by applying a set of rules that properly specify compatible and non-contradictory equivalences between UML and Maude elements.

Transformations should be applied both to the computational model structure, and to the behavioral part. The former is simpler to do. However, it is always a challenge to design and, even more, to translate behavior to some other representation respecting the meaning and the semantics, due to the dynamic nature of behavior. Moreover, behavioral specifications often depends on the designer's way of modeling things. In this paper, we assume that a behavioral specification is modeled in terms of state machines for the computational objects, and interaction diagrams that represent how the dialog between the objects and their environment through their computational interfaces is carried out.

Figure 6 shows the most important structural correspondences between both meta-models using a graphical notation, as indicated. Dependencies define relationships between transformation rules, so if rule *R2* depends on rule *R1*, then rule *R1* implies one or more executions of rule *R2*. As shown in the diagram, six transformation rules have been identified: *INT* (INTeration), *CO* (Computational Object), *CI* (Computational Interface), *COT* (Computational Object Template), *CIT* (Computational Interface Template) and *IS* (Interface Signature). Note that this nomenclature for rule names has been used for simplicity, since transformation rules do not necessarily have to match ODP concepts. In fact, an ATL program is usually composed of many other elements, such as *helpers*, *imperative blocks*, etc. [3]. These rules, as presented in this paper, mainly represent the declarative part of the transformation program.

For instance, the next piece of code specifies the transformation rule *COT*, which is just comprised of a declarative block.

```
rule COT {
  from
    comp : UML!Component
    (comp.isStereotyped('CV_ObjectTemplate'))
  to
   cls : Maude!ClassDecl (
      name <- comp.name,
      attrDeclList <-
        comp.ownedAttribute.toAttrString(),
      compInterfaces <-
        comp.ownedPort->iterate(
            p; res : Sequence() |
              res->union(thisModule.CIT(p)))),
   subcls : Maude!SubclassDecl (
      subsortRel.parents <- Set{comp.name}
      subsortRel.subclasses <- Set{CV-Object}),
   mod : Maude!Module (
      name <- 'COT' + comp.name,
      oDeclList <- Set{cls, subcls})
}

lazy rule CIT {
  from
    p : UML!Port
     (comp.appliedStereotypes->select(
        e | e.name = 'CV_InterfaceTemplate'
     )->notEmpty())
  to     ...
}
```

This rule takes a UML component (stereotyped as `CV_ObjectTemplate`) as the source element and creates a Maude module instance for the *computational object template* represented by the component. This `module` contains the declaration of the Maude `class`, whose instances correspond to *computational objects*, and a `subclass` declaration (because every *computational object* is inherited from `CV_Object`). The *helper* `toAttrString` is responsible of mapping a set of UML features (attributes) to a string declaration of attributes in the Maude-CV format. `compInterfaces` contains the set of *computational interface templates* associated to this *computational object template*. As mentioned in Section 3, an ODP *computational interface template* is modeled as a UML port, which is a feature of the component classifier that represents the *computational object template*. All these ports are trans-

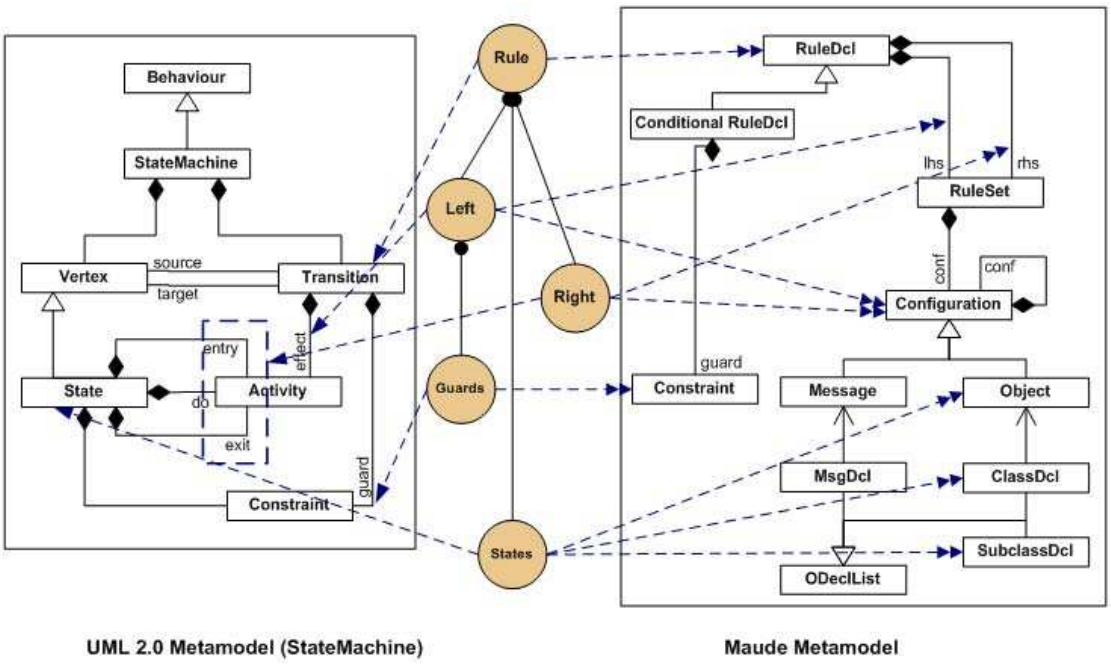

Figure 7. Behavioral mappings

lated using the lazy rule *CIT*. In ATL, *lazy rules* are not directly invoked by the transformation engine as the result of the matching process, but they are called instead from other rules by passing as parameters the source elements for the lazy rule.

The transformation rule *CIT* creates the Maude target elements for an ODP *computational interface template* in a similar way that *COT* does for *object templates*. Thus, for brevity, the target block in the rule *CIT* has been omitted.

Analogously, Figure 7 shows a schema of the transformation rules required to perform behavioral mappings from UML state machines to the Maude specification model. In this case, each state is translated to one `subclass` of the Maude `class` representing the *object template*, from which the object is instantiated. A transition between two states is represented by the trigger of a Maude `rewriting rule` by properly indicating the left-hand side (previous state and invariants) and the right-hand side (do, entry, exit activities and effects) of the rule. In case the transition includes some guard to be triggered, then it is specified with the `constraint` of a `conditional rewriting rule`. Both the left and the right sides are composed of a `configuration` of `messages` and `objects` that represent which events are received or sent and how objects are affected by any state change. One important feature of Maude is that it allows to *reclassify* objects, i.e., the class of an object can be changed at run-time. So, if the state changes after a transition, the Maude object's class also can be changed to represent the new state.

For example, going back to the multimedia application, the transformation process for the *COT* rule produces the effect shown in the right-hand side of Figure 8.

5.2 Executing the specifications

Once we have briefly shown the model transformations required to transform UML4ODP computational specifications into Maude, we are now ready to execute and analyze the specifications.

Figure 9 shows a diagram with a possible sequence of model transformations. After representing his specifications in UML, the ODP designer makes the translation to Maude-CV as described above, and then to Maude code by using simple template-based transformations. The results of the Maude execution or analysis is then modeled back in UML by applying the reverse transformations (see also Section 6).

Implementations to other technology platforms (Java, CORBA) could be described too, as depicted in Figure 9. In this way, starting from a UML computational specification of the system we could analyze it using the Maude formal toolkit and quick-prototype it with Maude. Once the results are convincing, the system could be implemented in different technologies. (Notice that to ensure that these latter implementations are also correct we need to prove that the

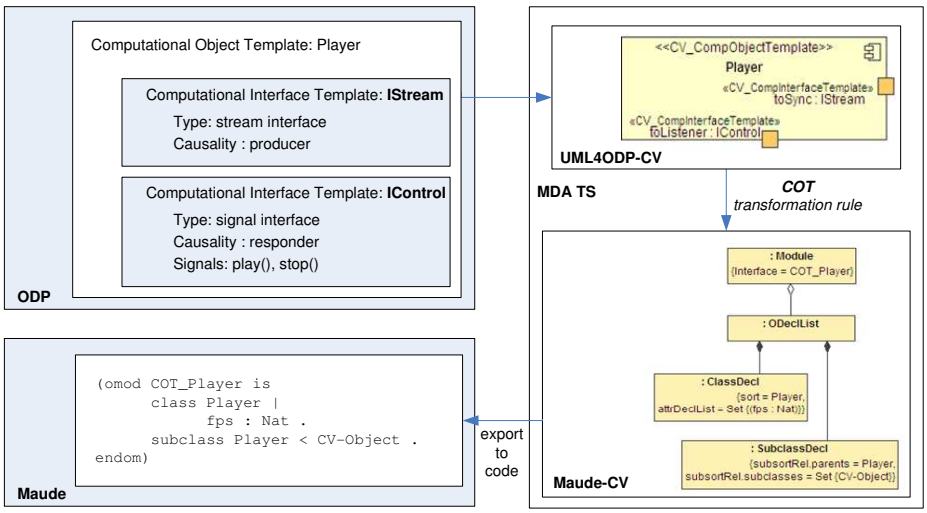

Figure 8. Application of the transformation rule COT

UML-to-Java and UML-to-CORBA transformations preserve the proved properties—see Section 6.)

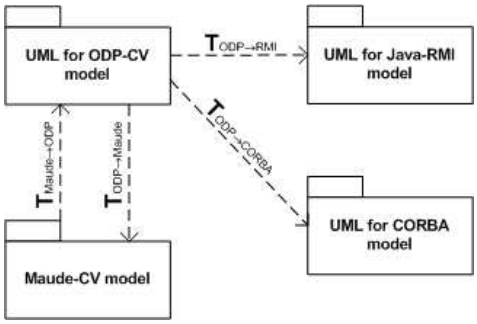

Figure 9. Mappings to different models.

6 Issues for discussion

6.1 The ODP-CV metamodel as intermediate

The UML Profile for the ODP Computational Viewpoint is not the only approach for representing a computational specification; some other approaches are also available, e.g., EDOC-CCA [17]. In this paper we have defined transformations using UML4ODP-CV models as the source models, although the same could have been done for the EDOC CCA models (see Figure 10(*b*)).

An alternative approach could use a metamodel of the computational viewing as pivotal element for the transformations, as shown in Figure 10(*a*).

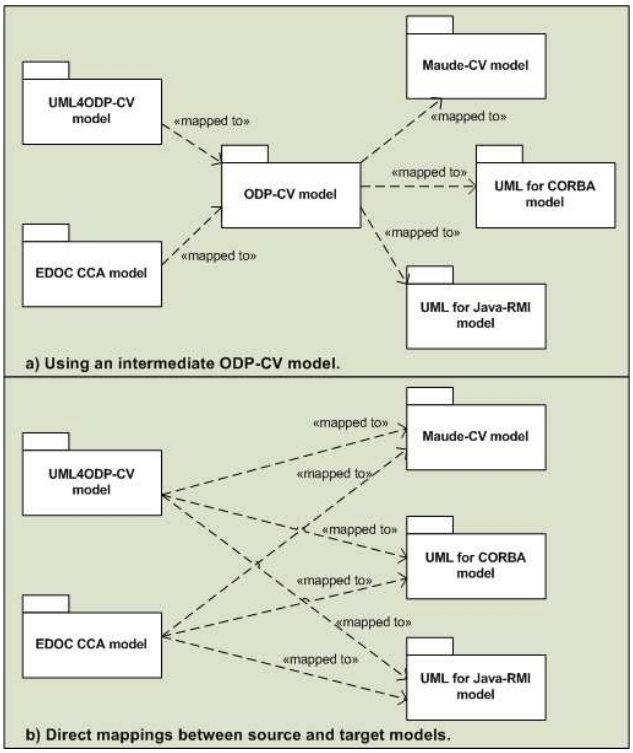

Figure 10. ODP-CV as intermediate.

In case (*a*), we would need to write $(n + m)$ transformation programs (i.e., ATL programs), where n is the number of possible representations for the source model and m corresponds to the number of specific implementation-based choices. However, in case (*b*), the number of programs

to code and, consequently, to be managed increases until $(n * m)$. However, the main problem with option (a) is that we might lose some interesting information in the transformation processes (e.g., some behavioral descriptions of the UML4ODP-CV models), that might be required specially when transforming from Maude to UML4ODP-CV. This is an issue that we'd like to explore further.

6.2 Preservation of properties

In the context of Figure 9, something which is really required is the guarantee that the transformations between models preserve some of the properties of the source model. Thus, we'd be able to guarantee that the properties that we have proved using Maude are preserved by the Java or CORBA implementation of the system. Otherwise we might be wasting our time and efforts by checking the system first. In this sense, we would like to be able to check whether a given model transformation preserves a given (safety or liveness) property of a system.

6.3 Bidirectionality of transformations

Figure 9 shows some bidirectional transformations between the UML4ODP-CV and the Maude-CV models. Reverse transformations are usually difficult to define, specially in ATL where transformations are unidirectional (e.g., we would need to separately specify the transformation program $T_{MaudeCV \rightarrow ODPCV}$ for obtaining the corresponding reverse translation). And even in QVT, dealing with bidirectional transformations is not simple (probably not from the technical side, but from the conceptual level). Do we really need to define the full reverse translation from Maude-CV to UML for ODP-CV?

In fact, what we have discovered is that the only thing we need to transform back to UML4ODP-CV are the resulting configurations or the errors found after executing or analyzing the Maude specifications. And this is much more simpler (no Maude behavior needs to be transformed).

7 Conclusions

In this paper, we have discussed how to build a bridge between the UML 2.0 and the Maude specifications of the ODP computational viewpoint, connecting both worlds. One of the major benefits of our contribution is that it allows the stakeholders of the system to use a more user-friendly graphical notation like UML to express the system's structure, requirements and behavior, and then translate them into Maude specifications. More precisely, we have shown how model transformations can be specified in ATL between the UML and Maude specifications, so model transformation engines can later implement such transformations.

The connection between graphical and formal notations is not a new problem. However, most of the existing proposals provide ad-hoc solutions, with hard-wired compilers of graphical models into particular formal notations. What we have showed in the paper is how the use of MDA principles and mechanisms, together with MDA tools for specifying and implementing model transformations, can help achieve such connection in a high-level and declarative manner, truly providing the primary goals of MDA: portability, interoperability and reusability [18].

Apart from validating our proposal with more examples and applications, addressing the discussed issues and evaluating its performance and scalability, there are some lines of work that we plan to address shortly. First, we want to embed the transformations into a (UML) tool that can provide access to the Maude toolkit from the UML environment. That would allow systems engineers to execute and model-check their UML ODP computational specifications, without being aware that they are using Maude or its toolkit. We also want to study how to define transformations between the UML models and some commercial implementation platforms, such as CORBA or .NET, that can preserve the properties that have been validated with Maude. In that way, we will be able to ensure that the system implementation will satisfy a set of properties that have been previously proved using the Maude toolkit. Thus, we will be able to decide whether a given model transformation from a repository can be used to produce target models that satisfy that property, if the source models do.

Acknowledgements The authors would like to thank the anonymous referees for their insightful comments and suggestions. This work has been supported by Spanish Research Project TIN2005-09405-C02-01.

References

[1] A. Albarrán, F. Durán, and A. Vallecillo. From Maude specifications to SOAP distributed implementations: A smooth transition. In *Proc. of JISBD'01*, Almagro, Ciudad Real (Spain), November 2001.

[2] A. Albarrán, F. Durán, and A. Vallecillo. Maude meets CORBA. In *Proc. of ASSE'01)*, Argentina, September 2001.

[3] ATLAS group, LINA and INRIA, Nantes, France. *ATL User Manual*, Feb. 2006. version 0.7.

[4] J. Bézivin, E. Breton, P. Valduriez, and G. Dupé. The ATL transformation-based model management framework. Research Report 03.08, IRIN, University of Nantes, 2003.

[5] C. Bock. UML 2 activity and action models part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

[6] H. Bowman, J. Derrick, P. Linington, and M. W. Steen. FDTs for ODP. *Computer Standards & Interfaces*, 17:457–479, Sept. 1995.

[7] G. Caplat and J. Sourrouille. Model Mapping in MDA. In *Workshop in Software Model Engineering — WISME 2002*, Dresden, Germany, 2002.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[9] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *online proceedings of the 2nd OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA*, Anaheim, Oct. 2003.

[10] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

[11] J. B. I. Kurtev and M. Aksit. Technological Spaces: An Initial Appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial Track*, Irvine, 2002.

[12] ISO/IEC. *RM-ODP. Reference Model for Open Distributed Processing*. Geneva, Switzerland, 1997. International Standard ISO/IEC 10746-1 to 10746-4, ITU-T Recommendations X.901 to X.904.

[13] ISO/IEC. *Information technology – Open distributed processing – Use of UML for ODP system specifications*. International Standards Organization, Geneva, Switzerland, 2006. ISO/IEC FCD 19793, ITU-T Recommendation X.906.

[14] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track*, Dijon, France, 2006.

[15] P. F. Linington. What foundations does the RM-ODP need? In *Proc. of the 1st International Workshop on ODP in the Enterprise Computing (WODPEC)*, pages 17–22, Monterey, California, Sept. 2004.

[16] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[17] OMG. *A UML Profile for Enterprise Distributed Object Computing V1.0*, Aug. 2001. OMG doc. ad/2001-08-19.

[18] OMG. *Model Driven Architecture (MDA) Guide*, 2003. OMG doc.ab/2003-06-01.

[19] OMG. *MOF QVT Final Adopted Specification*. Object Management Group, Nov. 2005. OMG doc. ptc/05-11-01.

[20] OMG. *Unified Modeling Language Specification 2.0: Superstructure*, 2005. OMG doc. formal/05-07-04.

[21] J. R. Romero and A. Vallecillo. Modeling the ODP Computational Viewpoint with UML 2.0. In *Proceedings of the 9th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, Enschede, The Netherlands, Sept. 2005. IEEE CS Press.

[22] J. R. Romero and A. Vallecillo. UML 2.0 Profile for the ODP Computational Viewpoint. Technical Report TR-05-03, Universidad de Málaga, Feb. 2005.

[23] R. Romero and A. Vallecillo. Formalizing ODP computational specifications in Maude. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, pages 212–233, Monterey, California, Sept. 2004. IEEE CS Press.