# A Journey through the Secret Life of Models

Antonio Vallecillo

GISUM/Atenea Research Group. Universidad de Málaga (Spain)
`av@lcc.uma.es`

**Abstract.** Although Model Driven Software Development (MDSD) is achieving significant progress, it is still far from becoming a real Engineering discipline. In fact, many of the difficult problems of the engineering of complex software systems are still unresolved, or simplistically addressed by many of the existing MDSD approaches. In this paper we outline three of the currently outstanding problems of MDSD on which we are now working, and propose some hints on how they can be addressed. The challenges are: the specification of the behavioral semantics of metamodels; the addition of time to these specifications so that models can be properly animated, simulated and analyzed; and the use of viewpoints and correspondences for specifying large-scale software systems.

## 1 Introduction

The ideas behind Model Driven Software Development (MDSD) have been present for years, although they did not become really popular until the beginning of this century. Probably, one of the triggers of this renewed and wider popularity was the launching, in late 2000, of the Model-Driven Architecture (MDA) initiative by the OMG. The MDA proposed a new way to consider the design, development and maintenance of information systems, using models as the essential artifacts of the software development process. MDA attracted a real interest from industry because of its many potential benefits: increased productivity and quality of the resulting products, reduced development costs and efforts, reduced time-to-market, increased protection of investments, ability to modernize existing and legacy applications, etc.

Although both MDSD and MDA have experienced significant advances during the past 8 years [1], some of the key difficult issues still remain unresolved. In fact, the number of engineering practices and tools that have been developed for the industrial design, implementation and maintenance of large-scale, enterprise-wide software systems is still low—i.e. there are very few real *Model-Driven Engineering* (MDE) practices and tools. There are several reasons for that. Firstly, many of the current MDSD processes, notations and tools usually fall apart when dealing with large-scale systems composed of hundred of thousands of highly interconnected elements; and secondly, MDE should go beyond generative programming: other engineering activities (such as simulation, analysis, validation, quality evaluation, etc.) should be fully supported too.

In addition, we are currently in a situation where the industry is interested in MDE, but we can easily fail again if we do not deliver (promptly) anything really useful to them. There are still many challenges ahead, which we should soon address in order not to lose the current momentum of MDE.

In this paper we focus on three of these challenges, which we consider relevant to the real industrial adoption of MDE practices and tools, and on which we are currently working (with different levels of achievement). Basically, they are all issues that have been successfully solved by other mature engineering disciplines (civil engineering, avionics, even hardware!) and that should also be solved for MDE if a real engineering of large software systems is to be achieved.

The first issue is the specification of the behavioral semantics of metamodels (beyond their basic structure) so that models can be animated, and different kinds of analysis can be conducted on them including, e.g., simulation, validation and model checking. A second challenge is the support of the notion of time in these behavioral descriptions, another key issue to allow industrial systems to be realistically simulated and properly analyzed—to be able to conduct, e.g., performance, schedulability or reliability analysis. Finally, we need not only to tackle the *accidental* complexity involved building software systems, but we should also try to deal with their *essential* complexity [2]. In this sense, the effective use of independent but complementary viewpoints to model large-scale systems, and the specification of correspondences between them to reason about the consistency of the global specifications, is the third of our identified challenges.

## 2 Adding Behavioral Semantics to DSLs

Domain Specific Languages (DSLs) are usually defined only by their abstract and concrete syntaxes. The abstract syntax of a DSL is normally specified by a metamodel, which describes the concepts of the language, the relationships among them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules.

The concrete syntax of a DSL provides a realization of the abstract syntax of a metamodel as a mapping between the metamodel concepts and their textual or graphical representation (see the right hand side of Fig. 1). A language can have several concrete syntaxes. For visual languages, it is necessary to establish links between these concepts and the visual symbols that represent them—as done, e.g., with GMF [3]. Similarly, with textual languages it is necessary to establish links between metamodel elements and the syntactic structures of the textual DSL [4].

Current DSM approaches have mainly focused on the structural aspects of DSLs. Explicit and formal specification of a model semantics has not received much attention by the DSM community until recently, despite the fact that this creates a possibility for semantic mismatch between design models and modeling languages of analysis tools [5]. While this problem exists in virtually every domain where DSLs are used, it is more common in domains in which behavior

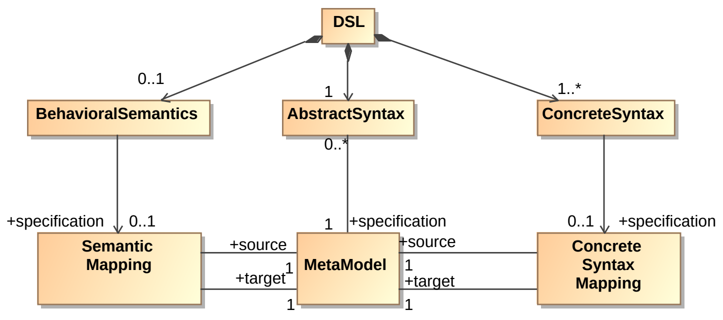

**Fig. 1.** Specification of a Domain Specific Language

needs to be explicitly represented, as it happens in most industrial applications of a certain complexity. This issue is particularly important in safety-critical real-time and embedded system domains, where precision is required and where semantic ambiguities may produce conflicting results across different tools. Furthermore, the lack of explicit behavioral semantics strongly hampers the development of formal analysis and simulation tools, relegating models to their current common role of simple illustrations.

## 2.1 A Running Example

To illustrate the ideas presented in this paper, we will use a running example: an industrial production line for assembling parts (in this case, hammers).

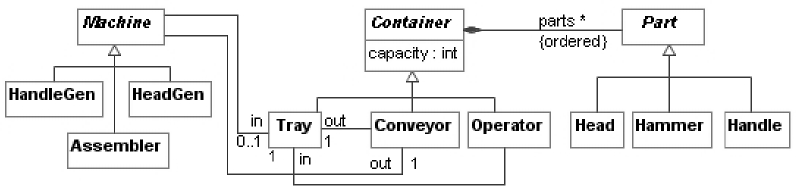

**Fig. 2.** Meta-model of a DSVL for production systems.

This kind of production systems consist of machines, containers and parts. Machines can be either generators of hammers' heads or handles, or assemblers. Generators produce parts and assemblers consume them to create new ones. Machines take their inputs from trays and put their results in conveyors, which transfer the parts between the trays. Operators collect assembled hammers (in this sense, they act as containers, too). The metamodel for this kind of production systems is shown in Fig. 2.

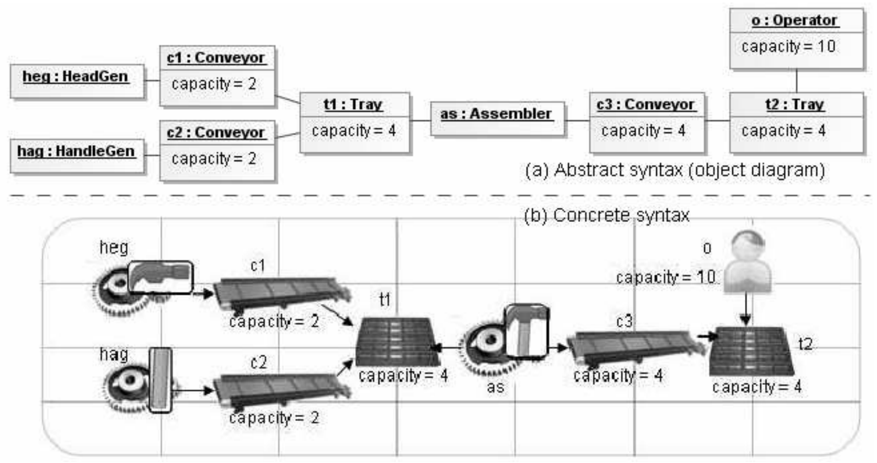

**Fig. 3.** initModel example production system.

Fig. 3 shows an example of a valid production system, using the abstract syntax and a concrete visual representation for it. Machines are represented as cogwheels with an icon showing the kind of part they generate. Conveyors, trays and users are represented using easily-recognizable icons. This model consists of two generators connected to two conveyors, which are in turn connected to one tray from which parts are assembled, deposited in a third conveyor, and then stored in another tray. An operator collects hammers from the final tray.

Of course, this DSL for production systems have an implicit semantics, because everybody has an (informal) idea about how this kind of systems work. However, this lack of explicit and precise specifications opens the door to potential semantic mismatches, and hinders any kind of formal analysis of the system. For instance, it is not clear from the production system metamodel when are handles and heads generated, or what happens when one of the trays is full: does the preceding machines stop until the tray has some free space, or do the newly generated pieces fall into the plant's floor? Can we simulate the system with different trays' capacities, in order to find the most efficient production line? How will the system perform if more generators are added, or if we change the current configuration of trays and conveyors? These are the sort of issues that need to be precisely clarified by a behavioral specification.

## 2.2 Defining DSLs Behavioral Semantics

In general, semantic has not been added to DSLs or has been done using natural languages. Although users can normally guess the meaning of most terms of the DSL (a good language designer probably chooses keywords and special symbols with a meaning similar to some accepted norm), a computer cannot act on such assumptions [6]. To be useful in the computing arena, any language (whether it is textual or visual or used for programming, requirements, specification, or design) must come with rigid rules that clearly state allowable syntactic expressions and give a precise description of their meaning.

A formal description of a language semantics is usually done using an *operational*, *denotational*, or *axiomatic* style (see [7] for a comprehensive survey of semantic description frameworks). In operational frameworks, the semantics of the language is specified as a sequence, or execution history, of state transitions, usually as operations on some hypothetical abstract machine. A denotational semantics is given by a mathematical function which maps the syntax of the language to a semantic value (a denotation). Axiomatic semantics involves rules for deducing assertions about the correctness or equivalence of language expressions and corresponding parts. Each of these frameworks has particular properties, but the distinction between them is seldom sharp: they frequently borrow features from each other.

In any case, the definition of the semantics of a language can be accomplished through the definition of a *mapping* between the language itself and another language with well-defined semantics (see the left hand side of Fig. 1) such as Abstract State Machines [8], Petri Nets [9], or rewriting logic [10]. These *semantic mappings* [6] between semantic domains are very useful not only to provide precise semantics to DSLs, but also to be able to simulate, analyze or reason about them using the logical and semantical framework available in the target domain [11]. The advantage of using a model-driven approach is that these mappings can be defined in terms of model transformations.

### 2.3 Describing Dynamic Behavior using Model Transformations

One particular way to specify the dynamic behavior of a DSL is by describing the evolution of the state of the modeled artifacts along some time model. In MDE, where models, metamodels and model transformations are the key artifacts, model transformations seem to be the natural way. In particular, model transformation languages that support in-place update [12] are perfect candidates for the job. This kind of transformations allow users to describe the permitted actions and how the model elements evolve as a result of these actions.

There are several approaches that propose in-place model transformation to describe the behavior of a DSL, from textual to graphical. Graphical notations are proving to be extremely helpful in software and systems development, since they are are intuitive to specify and to understand. Furthermore, in a theoretical sense, textual and graphical notations have no principal differences, since a formal definition can be established in both. Textual notations are instead more expressive, and usually graphical notations make use of (some kind of) them to deal with complex behavior.

One of the most important graphical approaches is Graph Grammars [9, 13], in which the dynamic behavior is specified by using visual rules. These rules prescribe the preconditions of the actions to be triggered and the effects of such actions, given both visually as models that use the concrete syntax of the DSL. This kind of representation is quite intuitive, because it allows designers to work with domain specific concepts and their concrete syntax for describing the rules [9]. There are also other graphical approaches, most of which are in turn based on graph grammars. Among them, we can find the visual representation of QVT [14] (where QVT is given in-place semantics) or the use of different (usually extended) UML diagrams [15, 16]. These approaches do not use (so far) the concrete syntax of the DSL, but an object diagram-like structure. Furthermore, most of them (including graph grammars approaches) use their own textual language to deal with complex behavior, such as Java [15] or Python [17]. Regarding textual approaches, we can find, e.g., the use of QVT (in a textual fashion) or rewrite logic rules [10].

Of course, there are other textual notations to describe behavioral semantics of models, but they do not make use of in-place model transformations. For example, Abstract State Machines are used in [8] to specify the operational semantics of metamodels, and MOF is extended with an action language (Kermeta) in [18]. Each approach has its own benefits and limitations, and is better suited for some purposes. As we shall see in Section 2.5, the possibility of defining semantic bridges between the different approaches can be of great help for obtaining the best of all worlds.

In in-place model transformation languages, source and target models are always instances of the same metamodel (i.e., they are *endogenous* transformations), and transformation rules are of the form $l : [\mathrm{NAC}] \times \mathrm{LHS} \longrightarrow \mathrm{RHS}$, where $l$ is the rule label (name); LHS and RHS are model patterns that represent certain states of the system, and NAC is a set of optional negative application conditions that forbids applying the rule if one of these patterns are found in the model. Rules are applied if an occurrence (or match) of the LHS is found in the source model, and the NAC is not found. Then, the target model is obtained by substituting the match by the RHS. More precisely, after the application of the rule the elements in the LHS that do not appear in the RHS are deleted, whereas the elements in the RHS that do not appear in the LHS are created. Elements that appear in both patterns will be modified as indicated. Rules may include attribute conditions (which must be satisfied by the match) and attribute computations. Generally, if several matches are found, one is selected randomly; the model transformation proceeds by applying the rules in non-deterministic order, until none is applicable (although this behavior can be modified by some execution control mechanisms).

## 2.4 The Example Revisited

To illustrate how to specify the dynamic behavior of the production system, this section shows some examples of the rules that need to be added to its metamodel specification. The interested reader can consult [19] or [20] for more complete descriptions of this approach.

For instance, Fig. 4 shows the behavior of the head generator, which creates a hammer head and deposits it into its outgoing conveyor. Notice that this rule can only be applied if the conveyor has room to store the newly created part, as stated by the condition on the LHS. Otherwise, the rule will not be triggered.

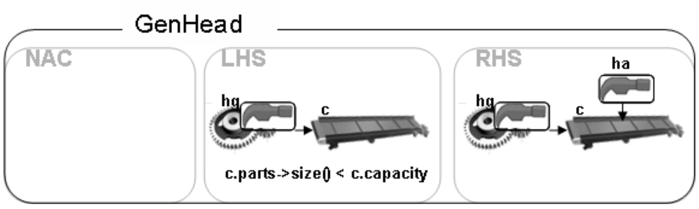

**Fig. 4.** GenHead rule.

Similarly, Fig. 5 specifies the behavior of the assemble machine. Every time that the assembler finds a head and a handle in its in-tray, it consumes these two parts (they disappear from the tray in the RHS) and deposits a newly assembled hammer in its outgoing conveyor. Again, the action specified by this rule will only occur if there is room in the conveyor to store the hammer.

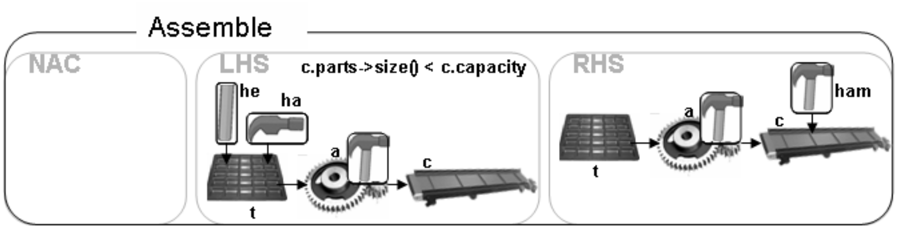

**Fig. 5.** Assemble rule.

Finally, Fig. 6 shows the movement of parts over conveyors. Notice the use of an abstract item p of class Part (represented by a hollow square) in the rule.

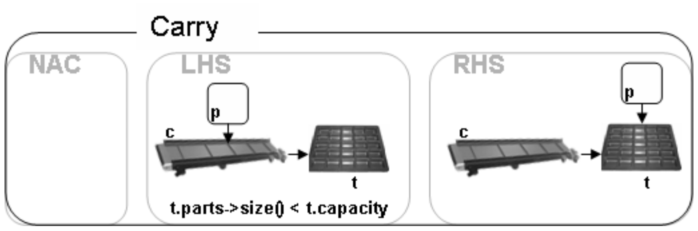

**Fig. 6.** Carry rule.

## 2.5 Model Simulation and Analysis

Having defined the behavior of a DSL, the following step is to perform simulation and analysis over the produced specifications. Simulation and execution possibilities are available for all the approaches on which behavior can be specified (including of course in-place transformations); but the kinds of analysis they provide is limited in many of them.

In general, each semantic domain is more appropriate to represent and reason about certain properties, and to conduct certain kinds of analysis. Defining the model behavior as a model will allow us to transform them into different semantic domain, depending on the kind of formal analysis we require (Fig. 7). Of course, not all the transformations can always be accomplished: it depends on the expressiveness of the semantic approach.

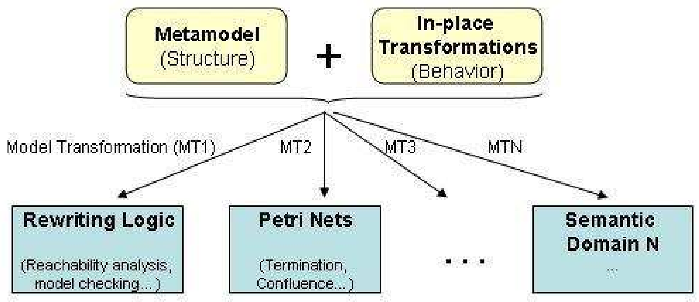

**Fig. 7.** Mapping a DSL to different semantic domains.

For example, the most common formalization of graph transformation is the so-called algebraic approach, which uses category theory to express the rewriting [21]. This approach supports a number of interesting analysis techniques, such as detecting rule dependencies or calculating critical pairs (minimal context of pairs of conflicting rules). Graph transformations have also been formalized into Petri Nets in [9], allowing termination and confluence analysis.

We have been working on the formalization of models and metamodels in equational and rewriting logic using Maude [22]. We have developed an Eclipse plug-in that allows to specify and implement some of the most common operations on metamodels, such as subtyping and difference [23], with a very acceptable performance. This formalization has also allowed us to add behavior [10] in a very natural way to the Maude specifications, and also made metamodels amenable to other kind of formal analysis, such as model checking, reachability analysis or theorem proving [20]. Furthermore, Maude offers very good properties as a logical and semantic framework, in which many different logics and formalisms can be expressed [24]. Therefore, our proposal can be used to provide a rich semantic framework in which other proposals can be mapped, hence allowing them to take advantage of the formal analysis methods and tools of Maude. For instance, we have already mapped Graph Grammars behavioral specifications of Domain Specific Visual Languages to Maude [19]. This allows performing simulation, reachability and model-checking analysis using the tools and techniques that Maude provides.

In general, we can map our behavior specifications to the semantic domain that provides the best formal analysis capabilities and tools we are looking for.

## 2.6 Analyzing the Hammer Production System

It is well known that "there are no correct programs, but untested ones". In fact, even this very simple production example is not free from errors, as can be easily uncovered when verifying the system models.

For instance, using *reachability* analysis we can look for deadlock states, e.g., final states in which the operator has collected less than 10 hammers. By observing them we realize that a possible source of deadlock is that the `t1` tray may be full of items of the same type, not allowing the assembler machine to proceed. This is due to a system design flaw, which can be easily solved by controlling that generators of pieces of type `X` do not produce new parts if tray `t1` already has `t1.capacity-1` parts of that type.

Are there more flaws left? For example, the question on whether the system may lose parts is left as an exercise for the reader . This simple question clearly shows the need for a complete set of analysis tools.

In [20] we showed some interesting examples of simulation and validation exercises on a similar production system that can be conducted with our approach, using the Maude toolkit [25].

## 2.7 End of Part 1: The (Secret) Journey of Models

Let us introduce here an intuitive analogy to illustrate our vision. As we perceive it, there is the *Universe* of behavioral semantic descriptions, which is composed of different *Worlds*, each one representing different semantic domains. For example we can have the Petri Nets world, the Process Algebra world, the Contracts world (in the sense of Eiffel contracts, with pre- and postconditions, and invariants), the Graph Grammars world, the Rewriting Logic world, etc. Each world is characterized by its underlying logic.

Each world is in turn composed of *Villages*, each one providing: a precise semantics, a concrete notation, and a set of tools. For example, within the Process Algebra world we find villages for the approaches that use CCS, CSP, or the various flavors of the $\pi$-calculus (normal, polyadic, etc.) for describing behavioral specifications and for specifying them with their associated tools. In the Rewriting Logic world we have the proposals based on languages and supporting systems such as OBJ, Cafe, Maude, etc. In the Petri Net world we have the approaches that use different versions and flavors of Petri Nets (basic, colored, hierarchical, etc.), each one with its precise semantics and associated notations and analysis tools.

In our analogy, defining semantic mappings means building *Bridges* between the different villages, so that the animated models can travel from one to the other, being able to use the local analysis tools to reason about their properties, conduct simulations, validations, etc. Bridges can also be very useful to provide semantics to those metamodels that lack precise semantics (e.g., UML). In these cases, the semantics of a source element is given by the semantics of the transformed element in the target domain.

Bridges are called *domestic* if they are defined between villages in the same world. In addition, bridges can go in any direction depending on the level of abstraction of the source and target villages. *Horizontal* bridges are those between villages that sit at the same abstraction level, maintaining the granularity of the specification. Bridges can also be *Abstracting* or *Refining*. Abstracting bridges are those that abstract away some of the details of the source model, leaving just the information that matters to the target domain. Refining bridges, on the contrary, add some information to the models that cross through them. For instance, the progressive refinements defined in the MDA chain (CIM→PIM→PSM) can be seen as refining transformations from high-level models into final implementations. Other approaches, such as the Architecture-Driven Modernization (ADM), use abstracting bridges through which the technology-dependent and platform-specific details are trimmed down from the low level implementations to leave just the platform-independent information. Abstract interpretation [26] is another well known technique that uses abstracting transformations to abstract away details which are irrelevant for our target analysis, while program refinement provides just the opposite approach. *Forgetful* bridges are an interesting kind of abstracting bridges which respect all elements of the abstract syntax, but forget about the semantics and/or the concrete syntax of the DSL (name inspired by the forgetful functors defined in category theory [27]). *Pruning* bridges are those forgetful bridges that also trim down some of the metamodel elements, in order to restrict the domain.

Please note as well that bridges are reusable, so once built they can used as long as the "crossing" model conforms to kind of models the bridge is able to deal with. Something that needs to be carefully considered when building bridges is that the results of the analysis conducted at the target end may need to return, and be properly represented at origin. This is an interesting challenge, which can be tackled, for instance, by annotating the transformed model with information about the source (i.e., leaving some track to allow returning) as described in [28].

Finally, the journey of a model should be kept as secret as possible (as the heading of this section suggests), so that engineers living on a particular village can be completely unaware of the fact that most of the analysis tools used to simulate and reason about their models are not actually available in that village (or even in that planet!). It is the modeling tools the ones that know the villages that models should visit to get the analysis done.

With all this, what we have is a three dimensional universe of semantic domains, inter-related by means of different kinds of bridges. We will come back to this later in Section 4.

## 3 Adding Time to Behavioral Specifications

Formal analysis and simulation are critical issues in complex and error-prone applications such as safety-critical real-time and embedded systems. In such kind of systems, timeouts, timing constraints and delays are predominant concepts [29], and thus the notion of time should be explicitly included in the specification of its behavior. Furthermore, without time information the kind of analysis that can be performed on the system are quite limited: if many real applications we need to be able to conduct performance and reliability analysis, too.

Most simulation tools that allow the modeling of time require specialized knowledge and expertise, something that may hinder its usability by the average DSL designer. On the other hand, current in-place transformation techniques do not allow to model the notion of time in a quantitative way, or allow it by adding some kind of clocks to the DSL metamodel. This latter approach forces designers to modify the DSL metamodel to include time aspects, and allows them to design rules that may easily lead the system to time-inconsistent states [29].

### 3.1 Modeling Timed Actions

One way to avoid this problem is by extending behavioral rules with their duration, i.e., by assigning to each action the time it needs to be performed. Thus, timed rules are of the form $l : [\mathrm{NAC}] \times \mathrm{LHS} \xrightarrow{t} \mathrm{RHS}$, where $t$ expresses the duration of the action modeled by the rule, in some time granularity.

Timed rules admit very rich semantics. In their simplest form, the LHS and NAC patterns express pre-conditions for the rule to be triggered. If they happen, the action specified by the rule is scheduled to happen after $t$ units of time. At that moment, the preconditions are again evaluated and, if they still hold, the rule is applied by substituting the match by the RHS (which expresses the postcondition). Otherwise, the rewrite will not take place. In another semantic variation of the rules, the LHS and NAC patterns should hold not only at the beginning and at the end of the action, but also in-between, i.e., they act as invariants (see Fig. 8). Further extensions to the rules are also possible, such as defining specific invariant patterns.

Our model of time also allows actions occurrences to be represented in rule conditions (LHS and NACs), opposite to standard in-place transformation approaches in which only system states can be specified (this makes many useful action properties inexpressible without unnatural changes to a system's specification, as discussed in [30]).

Thus, we can include *action expressions* in rule patterns to describe them. These *action expressions* are composed of the name (i.e., label) of the rule that represents the corresponding action, which may incorporate a parameter to indicate if the action is being realized or was performed previously (the present/past parameter), and optionally identifiers of matching parameters. These action expressions represent rule executions.

Modeling both state-based and action-based properties is a powerful mechanism that eases designers in the modeling of complex systems, and saves them

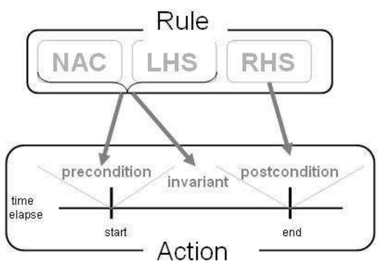

**Fig. 8.** Actions modeled with timed rules.

from including behavioral aspects in the structural specifications. As normal rules, timed rules can be applied only if an occurrence of the LHS pattern is found in the model. This allows elements to be performing several actions at the same time. In cases in which this situation is not desired (i.e., we want some elements to realize only one action at a time) or on the contrary needed (i.e., an element must be performing one action to be able to conduct another), this behavior can be easily restricted (or enforced) by including action expressions in the NAC or LHS patterns, respectively, as we show in next subsection.

## 3.2 Adding Time to the Production System Example

Let us illustrate here how the behavior of the production system example can be extended with time information. Fig. 9 shows a timed version of the Carry rule described before. Note that the only difference with standard in-place rules is the specification of the time the action consumes (in this case five time units). According to this rule, when a part is placed on a conveyor, it takes five time units to be deposited in the conveyor's out tray.

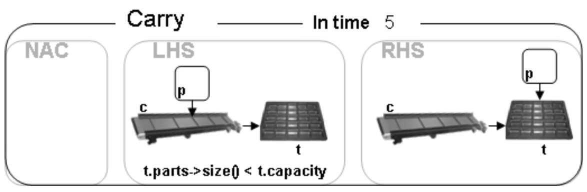

**Fig. 9.** Carry rule with time.

Now, what happens if a piece is placed on the conveyor when it is carrying another piece? As previously mentioned, timed rules can be applied if an occurrence of the LHS pattern is found in the model. Since there is no restriction about this situation, conveyors can move several pieces at the same time, and each of them will stay five time units on the conveyor.

Fig. 10 shows a timed version of the behavior of a head generator, which creates a hammer head every two time units. Since we do not want this rule to be triggered while it is executing, we restrict the GenHead rule with the inclusion of an action expression in its NAC. This action expression forbids the matched generator hg to start creating a new head if it is already generating one, and thus, making heads to be generated every two time units. In case of more head generators in the system, rule GenHead will be applied to each of them separately—this is the reason why the parameter is required.

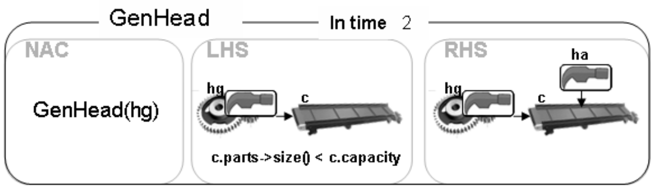

**Fig. 10.** GenHead rule with time.

In this example, action expressions have been simplified by omitting the present/past parameter and the instantiation objects role. Thus, every action expression will refer to actions that are being realized in the present, and objects will perform the role dictated by its class (since there are no patterns composed of several objects that belong to the same class).

Double Pushout (DPO) and Single Pushout (SPO) formalizations of in-place transformation rules, as well as non-injective rules, are also possible in our approach, with the usual semantics given in standard graph transformations [19].

## 3.3 Analysis of timed rules

In order to analyze the system behavior when it is specified using these timed rules, the usual theoretical results and tools defined for graph transformations are not easily applicable. However, other semantic domains are better suited. We are now working on the definition of a semantic mapping from our timed rules to real-time Maude's rewrite logic [31].

This mapping brings several advantages: (1) it allows to perform simulation, reachability and model-checking analysis on the specified real-time systems; (2) it permits decoupling time information from the structural aspects of the DSL (i.e., its metamodel); and (3) it allows to state properties over both model states and actions, easing designers in the modeling of complex systems.

Simulation and formal analysis (parts of the first advantage) are performed using Maude's toolkit, which includes tool support for simulation, reachability analysis and real-time model checking [25]. In addition, the way in which we have specified timed rules allows to re-use many of the existing theoretical results for timed-automata [32].

The second and third properties are achieved thanks to our Maude encoding of timed rules. A timed rule is encoded as three Maude rewrite rules, each one modeling one of the rule statements: the precondition, the postcondition and the invariant. *The precondition* Maude rule creates a timer object when the timed rule precondition is satisfied. This timer represents a countdown to the finalization of the rule, and gathers all the information needed about the rule instantiation: its name and the identifiers of the objects that participate in it. *The postcondition* Maude rule performs the action described by the timed rule (i.e., it replaces LHS with RHS) provided that the rule invariant is satisfied and the rule time consumed (i.e., the timer set to zero). (When an action is performed, it corresponding timer is saved for history purposes.) *The invariant* Maude rule interrupts the action (deleting it corresponding timer) whenever the invariant of the timed rule breaks. This invariant is initially defined using the same patterns as for the precondition: the LHS and negative application conditions (NACs).

Thus, our Maude encoding of timed rules makes DSL objects to be completely unaware of the notion of time, and time elapse must be only defined over the timers (using a Maude tick rule [31]). This also allows modeling elements that can perform several actions simultaneously in a natural way.

### 3.4 Further Challenges in the Analysis of Complex Systems

Being able to represent (timed) behavior represents just one small step in the hard task of being able to analyze, validate and reason about the models of the systems. In our vision, software designers should be able to do their work using engineering practices, like hardware designers and other traditional engineers have been doing for many years: you build the models of your systems first (probably one for each of the concerns you want to focus on—see Section 4), and then start performing simulations, validations, and many other kinds of analysis until they feel confident with their designs. At that moment, they are ready to *transform* their models into real implementations[1]. In the following we outline some (just a few) of the challenges involved in realizing this vision.

**Further analysis capabilities.** In addition to being able to simulate and model-check the software specifications to validate some liveness and safety properties of the modeled systems, we also need to address the analysis of their non-functional properties.

Since decades performance and reliability experts have built models for validating software/hardware systems against their non-functional requirements. In order to fill the gap between software development and non-functional validation, in the last few years the research has faced the challenge of automated generation of quantitative models for non-functional validation from software artifacts. Several methodologies have been introduced that all share the idea

[1] These transformations are much harder in other engineering disciplines. We, software engineers, are luckier because we do not need to change the media—and these transformations are now a core part of MDE!

of annotating software models with data related to non functional aspects and then translating the annotated model into a model ready to be validated (for a comprehensive survey of these proposals, see [33]).

In this sense, there is the need to connect design models (possibly annotated with time, probabilities and other non-functional properties) with existing non-functional analysis frameworks to conduct, for instance, performance and reliability analysis on the models (along the lines of the works described in [34]).

We have started studying how to extend our proposal to be able to model non-functional properties, and how to connect our specifications with these non-functional analysis frameworks. This includes first the appropriate representation of QoS characteristics (such as throughput, delay, overhead, scheduling policies, deadlines or memory usage) within the system specifications. In the context of UML, the new UML Profile for MARTE [35] is a valid step in this direction. Including this kind of QoS information in particular DSLs is something worth investigating.

Simulation languages and frameworks have also been present for years. For instance, Modelica [36] is an object oriented-language supported by a set of powerful simulation tools. The definition of transformations from our models to the Modelica system will allow us to make use of its simulation capabilities, something which we plan to do as part of our future work.

**Runtime monitoring.** This is particularly important in execution environments in which the level of quality of service (QoS) is constantly changing (as it happens with mobile or ubiquitous applications) and end-user service level agreements need to be guaranteed (essential in case of critical embedded systems, such as those used in the aerospace and automotive industries, nuclear plants, etc.). These facts make validation a very difficult task.

The use of MDE techniques for validating and monitoring runtime behavior can yield significant benefits here, something which has not been fully exploited yet. We believe that MDE can be effectively used not only to synthesize the system code from the models, but also for the derivation of all the instrumentation and monitoring code required for the effective QoS runtime management in disparate and dynamically changing execution environments.

**Modularity and scalability.** Finally, some of the major limitations of rule-based specifications are caused by the difficultly of managing the rules when their number is large. Modularity and composition mechanisms are required to be able to handle them in an orderly and controlled manner.

A good example of the use of modularity for timed systems in MoTif [37]. Based on the DEVS (Discrete Event System Specification) formalism, MoTif is a modular language for controlled graph rewriting, in which models and submodels can be re-used and composed to build larger specifications in a component-oriented fashion.

It is also important to note that all aspects that need to be considered when building the specification of a complex system cannot be treated in the same model—each concern should be dealt with independently (ideally using a specialized DSL) and then combined with the rest of the concerns. This leads us to the next (and final) section of our paper.

## 4 Viewpoint Integration and Consistency

Large-scale heterogeneous distributed systems are inherently much more complex to design, specify, develop and maintain than classical, homogeneous, centralized systems. Thus, their complete specifications are so extensive that fully comprehending all their aspects is a difficult task. One way to cope with such complexity is by dividing the design activity according to several areas of concerns, or *viewpoints*, each one focusing on a specific aspect of the system, as described in IEEE Std. 1471 [38].

Following this standard, current architectural practices for designing open distributed systems define several distinct viewpoints. Examples include the viewpoints described by Krutchen's "4+1" view model [39], Viewpoints [40], OpenViews [41], Dijkman's framework [42], or the growing plethora of Enterprise Architectural Frameworks (EAF): the Zachman's framework [43], ArchiMate [44], the US Department of Defense Architectural Framework (DoDAF), The Open Group Architectural Framework (TOGAF), the Federal Enterprise Architecture Framework (FEAF), or the Reference Model of Open Distributed Processing (RM-ODP), among others.

In particular, RM-ODP [45] is the enterprise architectural framework proposed by ISO/IEC and ITU-T for the specification of large, open and distributed systems. It provides five generic and complementary viewpoints on the system and its environment. Each viewpoint addresses a particular concern, and normally uses its own specific (viewpoint) *language*, which is defined in terms of a set of concepts specific that concern, their relationships, and their well-formed rules (i.e., a metamodel). A *view* (or *viewpoint specification*, in ODP terms) is a representation of the whole system from the perspective of a viewpoint.

Although separately specified, developed and maintained to simplify reasoning about the complete system specifications, viewpoints are not completely independent: elements in each viewpoint need to be related to elements in the other viewpoints in order to ensure the *consistency* and *completeness* of the global specifications. The questions are: how can it be assured that indeed *one* system is specified? And, how can it be assured that no views impose contradictory requirements? The first problem concerns the conceptual *integration* of viewpoints, while the second one concerns the *consistency* of the viewpoints.

Currently, most viewpoint modeling approaches to system specification (including the IEEE Std. 1471 itself and the majority of the existing EAFs) do not address these problems.

### 4.1 Correspondences between viewpoints

The most general approach to viewpoint consistency is based on the definition of *correspondences* between viewpoint elements. Correspondences do not form part of any one of the viewpoints, but provide statements that relate the various viewpoint specifications—expressing their semantic relationships [46]. Hence, we could initially say that a proper system specification consists of a set of viewpoint specifications, together with a set of correspondences between them.

The problem is that existing proposals and EAFs do not consider correspondences between viewpoints, or assume they are trivially based on name equality between correspondent elements, and are implicitly defined. This is a serious problem for large-scale distributed systems in which the viewpoints are indeed separately specified, and in which this simplistic assumption does not hold. Making an analogy with the common 2D representation of 3D figures, this is like drawing independently the three orthographic views of a figure but without defining any correspondence lines between them. As we all know, the consistency and completeness of the specification of the 3D figure cannot be guaranteed unless the appropriate correspondences between the three 2D views are described.

Furthermore, the majority of approaches that deal with the problem of inconsistency among viewpoints (see, e.g., [40, 42, 47–51]) assume that we can build an underlying metamodel containing all the views, which is not normally true. From a theoretical perspective, the use of a common global metamodel greatly helps maintaining the coherence and conceptual integration among viewpoint elements. However, the definition of such an underlying metamodel presents some problems. Firstly, should the metamodel consist of the intersection or of the union of all viewpoints elements? Some proposals (e.g., ArchiMate [44]) use the first approach (i.e., the intersection), while others, e.g., Dijkman [42] or Grosse-Rhode [50], use the second. Both approaches have serious problems with the extensibility and expressiveness of the basic elements of the global metamodel (not to mention complexity of the second approach—think for instance in the UML 2.0 metamodel). Secondly, the granularity and level of abstraction of the viewpoints can be arbitrarily different. Finally, the viewpoints may have very different formal semantics, which greatly complicates the definition of the common underlying metamodel.

The RM-ODP defines independent viewpoints for specifying open distributed systems, related by correspondences between them. In ODP, a *correspondence* is a statement by which some terms or other linguistic constructs in the specification of a viewpoint are associated with (e.g. describe the same entities as) terms or constructs in the specification of a second viewpoint.

There are two situations, depending on whether correspondences can be defined in terms of model transformations (i.e., functions) between the two related viewpoints, or not (i.e., just as mappings between the viewpoint elements).

## 4.2 Correspondences as model transformations

In the first case, the RM-ODP explicitly states that correspondences can be used to define transformations between viewpoint elements to implement consistency checks: “One form of consistency involves a set of correspondence rules to steer a transformation from one language to another. Thus given a specification $S_1$

in viewpoint language $L_1$ and specification $S_2$ in viewpoint language $L_2$, where $S_1$ and $S_2$ both specify the same system, a transformation $T$ can be applied to $S_1$ resulting in a new specification $T(S_1)$ in viewpoint language $L_2$ which can be compared directly to $S_2$ to check, for example, for behavioral compatibility between allegedly equivalent objects or configurations of objects" [45].

This approach has been proposed by several authors for relating concepts from different viewpoint at the metalevel (as initially suggested by Akehurst [52] using relations defined in OCL), and then further refined by Romero *et al* in [53] using QVT as transformation language. The main benefit of this approach is that it allows checking pairwise consistency between related viewpoints using standard mechanisms and tools (e.g., behavioral subtyping, bisimulation, etc.).

However, this approach presents two important limitations: ($a$) pairwise consistency is not enough for ensuring global consistency; and ($b$) there are many situations in which correspondences can not be specified as model transformations because they are not functions—rather, they are just data mappings between related elements but without any transformation or change propagation mechanism defined between them. In this latter case, techniques similar to model weaving [54] are more appropriate than model transformations.

## 4.3 Specification of viewpoint correspondences

ISO/IEC and ITU-T have defined a very expressive metamodel and profile (informally called UML4ODP) for specifying ODP viewpoints and correspondences between them, which allows to cover both cases [55]. In this approach, as shown in Fig. 11, a *correspondence specification* is composed of a set of correspondence *rules* and a set of correspondence *links*. In ODP, a *term* is a linguistic construct which may be used to refer to an entity. When a correspondence rule and a correspondence link are related, this means that the constraint in the correspondence rule must be enforced by the set of terms referenced by the correspondence link.

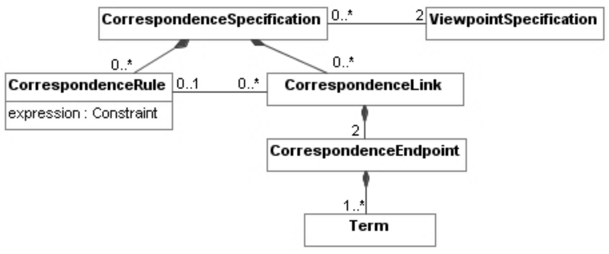

**Fig. 11.** ODP correspondence metamodel (from [55])

In UML4ODP, a correspondence rule is expressed by a constraint that must be enforced by a set of terms belonging to two specifications from different viewpoints. A correspondence link is established between two specifications from different viewpoints. Each end of the correspondence link is called a *correspondence endpoint*, which is composed of terms involved in the consistency relationship.

Something we recently discovered when working with this profile is that modeling views and correspondences between them is not enough. Well-formed rules on the set of correspondence specifications are also required to establish when the set of correspondences defined in a system specification is correct (see [56]). That is, this set of *well-formed rules* specifies the constraints that the set of correspondences should fulfil. OCL constraints can be used to express these rules in our context.

This gives rise to the following definition of multi-viewpoint specification of a system [56]:

**Definition 1.** *A* multi-viewpoint system specification *consists of a set of views* $V = \{V_1, \ldots, V_n\}$*, a set of correspondences* $C = \{C_{(1,2)}, C_{(1,3)}, \ldots, C_{(n-1,n)}\}$ *between the views, and a set of rules* $R = \{r_1, \ldots, r_k\}$ *that describe the constraints that the correspondences of* $C$ *should fulfil in order for a specification to be well-formed. Each view* $V_i$ *is a model that conforms to a metamodel* $\mathcal{M}_i$ *(the viewpoint language). Correspondences are also models, and* $C_{(i,j)}$ *conforms to a correspondence metamodel* $\mathcal{C}$*. Rules are expressed as constraints on the correspondence elements, using any constraint language (e.g., OCL).*

Our efforts are currently focused on the development of a generic framework and a set of tools to represent viewpoints, views and correspondences, which are able to manage and maintain viewpoint synchronization in evolution scenarios, as reported in [57], and that can be used with the most popular existing EAFs.

These tools are initially based on the ODP viewpoints, and on the UML4ODP profile for expressing the views and their respective correspondences. They allow validating the views in order to ensure that they conform to their respective metamodels (intra-viewpoint consistency), that the set of user-defined correspondences is well-defined, and to check that all required correspondences are properly fulfilled by the system specification.

## 5 Epilogue: A Hitchhiker's Guide to Metamodels

The MDE community is managing to know the answers to many of its current problems and limitations. These answers are expressed in terms of goals that need to be achieved for providing good engineering practices and tools for the industrial development of complex large-scale software system. As in Douglas Adams's book [58], we already know the answers—but now there is the need to formulate the questions so that these answers can be effectively implemented.

This paper provides a small step in this direction, by formulating three of the current challenges of MDE so that they can be properly addressed.

In the last section we discussed the importance of defining independent views of the system, each one focusing on one particular concern, and the use of Domain Specific Languages (DSLs) for specifying them. Up to now, DSLs have been defined by their abstract and concrete syntaxes only. But there is also the need to animate models, something for which we need precise description of the behavior of their metamodels. One way to specify the dynamic behavior of a DSL is by describing the evolution of the state of the modeled artifacts along some time model. In-place model transformations seem to be well-suited for this aim, extending metamodels (structural aspects of a DSL) with behavior. Furthermore, if these transformations use the concrete syntax of the DSL, the behavioral specifications become intuitive and natural both to specify and understand, because it allows designers to work with domain specific concepts. This kind of behavioral specifications also enables the addition of time information to the specifications in a natural way, as we have shown in Section 3.

We do not live in a Universe of isolated worlds, and thus we have also discussed the importance of building bridges that allow models to constantly travel to different worlds, being able to use the local analysis tools to reason about their properties, conduct simulations, validations, etc., and go back to their home villages with the results of the analysis—the life of models is quite traveled and hectic in our vision. Defining the behavior of a DSL as a model also permits us to exploit (using semantic mappings implemented by model transformations) the formal analysis and tools that each semantic domain provides. Bridges can also be very useful to provide precise semantics to those metamodels that lack them.

Finally, once we have a set of worlds composed of villages interconnected with bridges, we need to ensure the consistency between the set of models that comprise a multi-viewpoint specification of a complex software system, which live in different villages and may have different semantics. Correspondences can be of great help in this context. Another important problem is how to maintain this consistency between viewpoints in evolutionary scenarios.

But this is another story to be told...

**Acknowledgements.** The author would like to acknowledge the work of many people who have been involved in investigating and addressing the problems of the specification and modeling of large-scale software systems. Although the views in this paper are the author's solely responsibility, they could not have been formulated without many hours of detailed discussions with the rest of members of the GISUM/Atenea group (particularly José E. Rivera and José Raúl Romero, who are closely working with the author in developing these ideas and their associated tools); with the ISO experts on ODP, particularly Peter Linington, Akira Tanaka, Bruno Traverson, Sandy Tyndale-Biscoe and Bryan Wood; and finally, with a set of very intelligent and thought-provoking colleagues: Francisco Durán, Alfonso Pierantonio, Jeff Gray, Jean Bézivin, Juan de Lara, Iman Poernomo, Jean Marie Favre, José Meseguer, and many others.

The title of the paper has been borrowed from a great music album from Stevie Wonder, while Douglas Adams's masterpieces have served as an excellent source of inspiration for many of the ideas used to illustrate the proposal.

This work has been partially supported by Spanish Research Projects TIN2005-09405-C02-01, PET2006-0682-00 and P07-TIC-03184.

## References

1. Bézivin, J., Vallecillo, A., García-Molina, J., Rossi, G.: Special issue on MDSD: "MDA at the age of seven: Past, present and future". UPGRADE **IX**(2) (2008) 4–45 `http://www.upgrade-cepis.org/issues/2008/2/upgrade-vol-IX-2.pdf`.
2. Brooks, F.P.: No Silver Bullet – Essence and Accident in Software Engineering. In: Proceedings of the IFIP Tenth World Computing Conference. (1986) 1069–1076
3. Eclipse: Graphical Modeling Framework (2008) `http://www.eclipse.org/modeling/gmf/`.
4. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proc. of GPCE'06. (2006) 249–254
5. Kleppe, A.G.: A language description is more than a metamodel. In: Proc. of the Fourth International Workshop on Software Language Engineering (ATEM 2007), Nashville, USA (2007) `http://megaplanet.org/atem2007/ATEM2007-18.pdf`.
6. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? Computer **37**(10) (2004) 64–72
7. Zhang, Y., Xu, B.: A survey of semantic description frameworks for programming languages. SIGPLAN Not. **39**(3) (2004) 14–30
8. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: Proc. of Model Driven Architecture: Foundations and Applications (ECMDA-FA 2005). Volume 3748 of LNCS., Springer (2005) 115–129
9. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Proc. of FASE 2008. Volume 4961 of LNCS., Springer (2008) 77–92
10. Rivera, J.E., Vallecillo, A.: Adding behavioral semantics to models. In: Proc. of EDOC 2007, IEEE Computer Society (2007) 169–180
11. Cuccuru, A., Mraidha, C., Terrier, F., Grard, S.: Enhancing UML extensions with operational semantics: Behaviored profiles with templates. In: Proc. of MoDELS 2007. Volume 4735 of LNCS., Springer (2007) 271–285
12. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)
13. Kastenberg, H., Kleppe, A.G., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Proc. of FMOODS 2006. Number 4037 in LNCS, Springer (2006) 186–201
14. Marković, S., Baar, T.: Semantics of OCL Specified with QVT. Journal of Software and Systems Modeling (SoSyM) (2008)
15. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language. In: Proc. of the $6^{th}$ International Workshop on Theory and Application of Graph Transformation. (1998)
16. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Proc. of UML 2000. Volume 1939 of LNCS., Springer (2000) 323–337
17. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. Journal of Visual Languages and Computing **15**(3–4) (2006) 309–330
18. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proc. of MoDELS/UML'2005. Volume 3713 of LNCS., Springer (2005) 264–278

19. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with maude (2008) Submitted for publication (manuscript available at `http://www.lcc.uma.es/~av/Publicaciones/08/sle08.pdf`).
20. Rivera, J.E., Vallecillo, A., Durán, F.: Analysing models with Maude. Technical report, Universidad de Málaga (2008) `http://atenea.lcc.uma.es/images/e/eb/AnalysingModels.pdf`.
21. Ehrig, H., Karsten, Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
22. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with Maude. Journal of Object Technology **6**(9) (2007) 187–207
23. Rivera, J.E., Vallecillo, A.: Representing and operating with model differences. In: Proc. of TOOLS Europe 2008. Volume 11 of LNBIP., Springer (2008) 141–160
24. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In Gabbay, D., Guenthner, F., eds.: Handbook of Philosophical Logic. Volume 9. 2 edn. Kluwer Academic Publishers (2002) 1–87
25. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Number 4350 in LNCS. Springer, Heidelberg, Germany (2007)
26. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, ACM Press, New York, NY (1977) 238–252
27. Mac Lane, S.: Categories for the Working Mathematician. Springer (1971)
28. Guerra, E., de Lara, J.: Model view management with triple graph transformation systems. In: Proc. of ICGT 2006. (2006) 351–366
29. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: Causality and logical clocks. In: Proc. of ICGT 2002. (2002) 120–134
30. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Concurrency, Graphs and Models. Volume 5065 of LNCS., Springer (2008) 354–382
31. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196
32. Lynch, N.: Simulation techniques for proving properties of real-time systems. In: A Decade of Concurrency. Reflections and Perspectives. Volume 803 of LNCS., Springer (1994) 375–424
33. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Softw. Eng. **30**(5) (2004) 295–310
34. Cortellessa, V., Marco, A.D., Inverardi, P.: Integrating performance and reliability analysis in a non-functional MDA framework. In: Proc. of FASE 2007. Volume 4422 of LNCS., Springer (2007) 57–71
35. OMG: UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE). Object Management Group. (2008) OMG doc. ptc/08-06-08.
36. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press (2003)
37. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. In: Proc. of ICMT 2008. Volume 5063 of LNCS., Zurich, Switzerland, Springer (2008) 91–106

38. IEEE: Recommended Practice for Architectural Description of Software-Intensive Systems, New York, USA. (2000) IEEE Std. 1471.
39. Kruchten, P.: Architectural blueprints — The “4+1” view model of software architecture. IEEE Software **12**(6) (1995) 42–50
40. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: a framework for integrating multiple prespectives in systems development. SEKE journal **2**(1) (1992) 31–58
41. Boiten, E.A., Bowman, H., Derrick, J., Linington, P., Steen, M.W.: Viewpoint consistency in ODP. Computer Networks **34**(3) (2000) 503–537
42. Dijkman, R.M., Quartel, D.A., van Sinderen, M.J.: Consistency in multi-viewpoint design of enterprise information systems. Information and Software Technology **50**(7-8) (2008) 737–752
43. Zachman, J.A.: The Zachman Framework: A Primer for Enterprise Engineering and Manufacturing. Zachman International, La Cañada (CA), USA (1997) `http://www.zifa.com`.
44. Lankhorst, M., ed.: Enterprise Architecture at Work. Springer (2005)
45. ISO/IEC: RM-ODP. Reference Model for Open Distributed Processing. ISO and ITU-T, Geneva, Switzerland. (1997) ISO/IEC 10746, ITU-T Rec. X.901-X.904.
46. Linington, P.: Black Cats and Coloured Birds What do Viewpoint Correspondences Do? In: Proc. of WODPEC 2007, Maryland, USA, IEEE Digital Library (2007)
47. Easterbrook, S., Nuseibeh, B.: Using viewpoints for inconsistency management. Software Engineering Journal (1996) 31–43
48. Egyed, A.: Instant consistency checking for the UML. In: Proc. of ICSE’06, New York, NY, USA, ACM (2006) 381–390
49. Egyed, A.: Fixing inconsistencies in UML design models. In: Proc. of ICSE’07, Washington, DC, USA, IEEE Computer Society (2007) 292–301
50. Große-Rhode, M.: Semantic Integration of Heterogeneous Software Specifications. Springer-Verlag, Berlin (2004)
51. Straeten, R.V.D., Simmonds, J., Mens, T.: Detecting inconsistencies between UML models using description logic. In: Proc. of the International Workshop on Description Logics (DL’03). Volume 81 of CEUR Proceedings. (2003) `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-81/vander%straeten.pdf`.
52. Akehurst, D.H.: Proposal for a model driven approach to creating a tool to support the RM-ODP. In: Proc. of WODPEC 2004, Monterey, California (2004) 65–68
53. Romero, J.R., Moreno, N., Vallecillo, A.: Modeling ODP Correspondences using QVT. In: Proc. of MDEIS’06. (2006) 15–26
54. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Valduriez, P.: Applying generic model management to data mapping. In: Proc. of Bases de Donnés Avancées (BDA05), Saint-Malo, France (2005) 17–20
55. ISO/IEC: Information technology – Open distributed processing – Use of UML for ODP system specifications. ISO and ITU-T, Geneva, Switzerland. (2008) ISO/IEC FDIS 19793, ITU-T X.906.
56. Romero, J.R., Vallecillo, A.: Well-formed rules for viewpoint correspondences specification. In: Proc. of WODPEC 2008, Munich, Germany (2008)
57. Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A.: Change management in multi-viewpoint systems using ASP. In: Proc. of WODPEC 2008, Munich, Germany (2008)
58. Adams, D.: The Hitchhiker’s Guide to the Galaxy. Ballantine Books (1979)