Journal of Universal Computer Science, vol. 15, no. 11 (2009), 2196-2224
submitted: 18/1/09, accepted: 29/5/09, appeared: 1/6/09 © J.UCS

Checking Semantics Equivalence of MDA Transformations in Concurrent Systems

Paulo Barbosa, Franklin Ramalho,
Jorge Figueiredo and Antonio Júnior
(Federal University of Campina Grande, Campina Grande, Brazil
paulo@dsc.ufcg.edu.br, franklin@dsc.ufcg.edu.br,
abrantes@dsc.ufcg.edu.br, antonio@dsc.ufcg.edu.br)

Anikó Costa and Luís Gomes
(Universidade Nova de Lisboa, Lisboa, Portugal
akc@uninova.pt, lugo@uninova.pt)

Abstract: In a previous work we have proposed an extension to the four-layer MDA architecture promoting formal verification for semantics preserving model transformations. We analyzed semantics equivalence in transformations involving *Platform Specific Models* (*PSM*s). In this paper, considering concurrent systems domain, we show how this extended MDA architecture copes with the correctness verification of horizontal model transformations involving *Platform Independent Models* (*PIM*s). Our approach is supported by four formal techniques: behavioral equivalence relation, category theory, bisimulation and model-checking. This set of techniques allows the analysis of semantics equivalence between system model before and after transformation enabling the decomposition of the system model into a set of concurrent sub-models, considered as components. The validation of our approach occurs in a net splitting operation, where *PIM*s are defined as Petri nets models according to the *PNML metamodel* with transformations representing formal operations in this domain.

Key Words: MDA, transformations, formal semantics, concurrent systems, petri nets
Category: H.1, H.4.2, F.3.2, F.4.2

1 Introduction and Motivation

The Model-Driven Architecture (MDA) [Bettin 2004, Miller and Mukerji 2003, OMG 2009] provides several ways to define models representing systems, and transformations between these models. In MDA, a model is an abstract or concrete representation of a domain that enables communication between parts. Models are classified as platform-independent models (*PIM*s) and platform-specific models (*PSM*s). Furthermore, models are described by metamodels that specify the elements that can appear in the models. Another important concept in MDA are model transformations, a set of definition rules that describe how to generate an output model from an input model. Metamodels play an important role in the definition rules because they express the concepts and formalisms involved in the transformations. Currently, there is a wide range of tools that enables the transformation between models.

According to [OMG 2009, France and Bieman, 2001], a model transformation can have vertical or horizontal dimensions. In horizontal transformations the source and target models reside in the same level of abstraction, while in vertical transformations, they are in different levels of abstraction. Model abstraction and model refinement are examples of vertical transformations, whereas model refactoring is an example of horizontal transformation. For instance, code generation from a model into a target programming language (PIM-to-PSM) is an example of vertical transformation since some detail is added from the model to the generated code. In case of the generated code be exactly as specified in the model, we have model mappings, *i.e.*, horizontal transformations. Thus, PIM-to-PIM and PSM-to-PSM transformations may constitute vertical or horizontal transformations.

Although MDA promises to overcome important unsolved problems in software engineering, it has not specified ways to ensure that its transformations are correct. Particularly, the lack of artifacts that allows a formal representation of the involved models in the MDA architecture lead to undesirable situations of ambiguity and low reliability when comparing the behavior of the input and output models in its transformations.

The formal semantics of a model is the assignment of meanings to its sentences or components. This assignment is given by a mathematical model that represents every possible computation in the language that describes the model. The dynamic semantics has a finer-grained conception of meaning: it is the behavior of a sentence as its context potentially changes. It is expected, with the dynamic semantics description of a model, a foundation for understanding and evaluating the design issues, and a valuable reference for transformations that involve this model. In this sense, the formal definition of behavior is necessary to guarantee the preservation of properties in the output model after a transformation.

Concurrent systems are systems that have the property of having several processes or components executing at the same time, and that can interact with each other. Most popular applications, from tightly-coupled to loosely-coupled, from synchronous to asynchronous parallel and distributed systems satisfy this property. Moreover, these systems require supporting implementation of each involved process or component in a different platform, allowing heterogeneous and distributed execution of the system model. This requirement enforces the creation of several kinds of model transformations in very different contexts, requiring the application of techniques that guarantee the semantics preservation of the models involved in these transformations. There are many formal refinement techniques that are currently being implemented in the MDA framework [Wadsack and Jahnke 2002]. For instance, [Costa and Gomes 2007] adopts a set of executable MDA transformations that enable the automatic decomposition of the system model into a set of components in the concurrent systems domain. However, there is no guarantee that these (MDA) transformations are semantics-preserving.

In [Barbosa et al. 2008(b)], we have proposed *an extension of the four-layer MDA architecture* in order to incorporate formal semantics in its infrastructure. We introduced the formalization of *semantic metamodels*, *semantic models* as well as we proposed *simplification rules* and a *formal checker* to verify the equivalence of the input and output models involved in transformations and to prove properties about it, respectively. The proposed extension was analyzed and validated with some horizontal transformations involving (*PSM*s) in the *modelware* domain. More precisely, the models were represented using specific constructs of imperative and object oriented programming languages.

In this paper, the main focus is on the correctness verification of horizontal model transformations involving (*PIM*s) specifying concurrent systems, which represent a substantial topic of interest related to the MDA community. We propose and analyze a formal approach for verification of the dynamic semantics in MDA transformations to prove equivalence of concurrent and platform independent models. This approach fits well with our *extended MDA architecture*, but requires to be filled with specific techniques and tools necessary to deal with the essence of the concurrent paradigm.

The evaluation occurs in a project that explores several distinct models of computation for embedded systems' design. It proposes the use of Petri nets [Girault and Valk 2003] as the system-level specification language to model concurrent systems and components, which is verified and implemented in hardware or software using co-design techniques [Gomes and Costa, 2006]. We analyze and evaluate the MDA transformation representing the *net splitting operation*, that is able to decompose a Petri net model into Petri net sub-models using synchronous communication channels [Costa and Gomes 2007]. The presented proofs of soundness of the transformation for partitioning models and identification of sub-models intends to support the entire system development flow, from specification to implementation.

The structure of this paper is as follows. Section 2 presents background issues, with an overview of the current four layered MDA architecture, how we have extended it in order to incorporate formal semantics in its artifacts and the instantiation process of this architecture. Section 3 defines the case study to be analyzed during the explanation of the approach. Section 4 presents the instantiation of our solution for the concurrent systems domain. Section 5 gives details about our proposed approach for checking semantics equivalence between platform independent models of concurrent systems as well as presents the main formal techniques adopted to support it and validates the Splitting Operation in the context of the *FORDESIGN* project [Gomes et al, 2005]. Section 6 discusses similar works. Finally, Section 7 presents some final remarks and future research directions to the work.

2 Background

2.1 Model-Driven Architecture

Model-Driven Architecture is a software development approach that focuses on models, metamodels and transformations to define the elements of a system. Models are also key elements to direct the course of understanding, documentation and generation of artifacts that will become part of the overall solution. It is supported by the Object Management Group (OMG) [OMG 2009].

Models are primary artifacts to generate implementations by applying transformations. Three models are at the core of MDA. The first is the *Platform Independent Model* (PIM) which captures the requirements and design of a computational system independently of any target implementation platform. For instance, it describes a software system that supports some business, independently if it will be implemented with a relational database or as an application server. The second is the *Platform Specific Model* (PSM), that is the result of a transformation of the PIM. A PSM specifies the system in terms of a specific implementation technology. Moreover, we will have a generated PSM from a PIM for each specific technology according to the project requirements. Finally, we have the Code model. It is the final step in the development and is result of the transformation of each PSM. This transformation is relatively straightfoward because of the PSM completeness.

There are two kinds of transformations: *vertical* and *horizontal*. Vertical model transformations are used to refine or abstract a model; they affect the abstraction level of the model specification. The horizontal model transformations do not affect the abstraction of the model, they are used mainly to restructure it. PIM-to-PSM and PSM-to-PIM transformations are examples of vertical transformations that. The PIM-to-PSM transformations are performed once the PIM is elaborated enough to be associated to the characteristics of the platform, and PSM-to-PIM transformations are model reverse engineering transformations, they relate to abstraction of models into more general concepts.

Let us take as an example the context of systems specification through the UML language. In this case, the PIM is a UML model without any specific platform detail. Horizontal model transformations are able to refine these models using better design techniques, such as design patterns. A vertical model will translate this PIM to a PSM, generating a more detailed model in the specific plataform, such as Java, constructs. Finally, this PSM can straightfowardly be mapped into concrete syntax Java constructs through another vertical transformation, generating the final Java executable code.

All the MDA artifacts (models, metamodels and transformations) are organized according to the four-layer architecture provided by the OMG consortium [**?**]. It is presented in Figure 1, where is shown the context of two models involved in a transformation: the input model (left side) and the output model (right side). The layer M0 describes the concrete syntax of a given model. For instance, in programming languages it is the final executable code coupled to the chosen technology. M1 comprises artifacts which have similar characteristics in a model. M2 provides the metamodel which serves as a grammar to check the correctness of the model syntax developed at the layer M1. At the top, the highest layer, named M3, describes the layer M2 by using MOF. Since MOF describes itself, it does not require further metamodels. The model transformations are able to automatically generate output models from input models at the layer M1. They are defined in terms of metamodel descriptions and cope only with syntactic/structural aspects.

In addition, the OMG put forward some standards to specify the main artifacts of the MDA infrastructure, such as PIMs, PSMs, metamodels and transformations. Examples of these standards are: Meta-Object Facility (MOF), a language to specify metamodels; (ii) Unified Modeling Language (UML), as the main modeling language to specify PIMs and PSMs (by means of UML profiles), but other languages and formalisms can be used such as Petri nets (iii) the Object Constrain Language (OCL), a language to define constraints to avoid ambiguous model definitions; and (iv) Query/View/Transformations (QVT) that is a standard to define transformations, though the Atlas Transformation Language (ATL) [Bezivin et al. 2003] the most popular QVT-compliant language. Most of the OMG standards are built towards reuse and alignment between themselves.

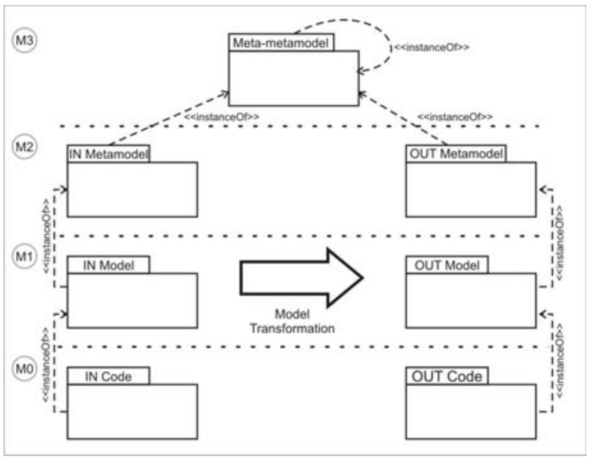


Figure 1: Four-layered MDA architecture

For instance, MOF is based on UML ad OCL and allows building metamodels as class diagrams annotated witl OCL constraints, *i.e.*, describing the concepts and their relations using classses, attributes, operations, associations and invariants on classes. Thus, MOF reuses constructs from UML class diagrams and OCL constraints at the meta-level layer.

2.2 The Extended MDA Architecture

In a previous work [Barbosa et al. 2008(b)], we have proposed an extended MDA architecture for incorporating formal semantics in its artifacts and transformations. It was evaluated for an hypothetic example on imperative languages. The proposed extension is shown in Figure 2. The new inserted modules to ensure formalization are in gray at the layers M1 and M2. Briefly, its new inserted parts are:

– *Static Semantics Module*, which provides the correct definition and representation of the involved concepts with well-formedness rules that allows to check important concepts, such as type-checking, scope verification and more.

– *Semantic Metamodel*, it is divided in two parts: *state structure* and *dynamic semantics*. The former defines the abstract syntax of the language to semantic specifications and the latter defines the dynamic semantics, capturing the state infrastructure of language constructs as well as how they use or change an existing state. For instance, a concrete representation can be a metamodel according to the EBNF grammar of a programming language defining metaclasses as semantic concepts and rules, giving meaning for all the constructs of this programming language, such as evaluation of statements, assignments, control and so on.

– *Semantic Model*, which instantiates the *state structure* of the *Semantic Metamodel*. For instance, for a given program, we have the representation of its semantic domains, representing the meaning the specific syntactic constructs. A representation of the environment of the program, the memory and its locations, the stack, are all components that together are employed to give the meaning of a program at a given moment.

– *Semantic Equations*, which define mappings, from the language's abstract syntax structures to meaning drawn from semantic models. As an example we can have mappings to automatically extract the semantic models from programs.

– *Simplification Rules*, which are able to perform the inference and computation of state configurations. As an illustration, after having the semantic models that represent the *state structure*, we have the definition of rules that enables computation of each constructs of the program, such as assignment or control. This leads to the instantiation of the dynamic semantics, because are concrete representations of the behavior of a program.

– *Formal Checker*, which is complementary to the *simplification rules*, and required to prove properties about the equivalence of the verified input and output models. For instance, we have employed theorem proving, in order to prove whether two imperative programs can reach the same state or not.

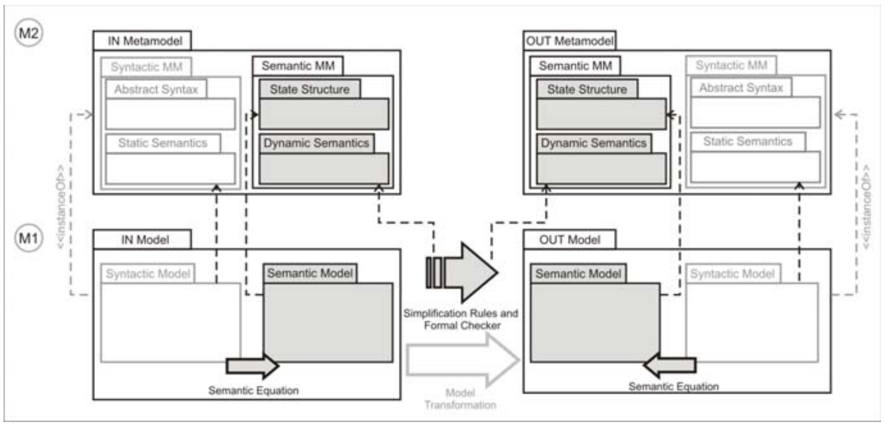


Figure 2: The extended MDA architecture to incorporate formal semantics

It is important to emphasize that the module *Formal Checker* requires different formal techniques according to the models, domains or specific platforms which it is employed. For instance, in [Barbosa et al. 2008(b)] we analyzed transformations involving *PSM*s that describe imperative programs. For them, we employed an approach following (i) *Continuation-Passing Style* (CPS) in the *rewriting logic framework*; and (ii) specific constructs represented in CPS semantics through program verification using the ITP theorem prover to the *Formal Checker* module. On the other hand, for applications involving transformations from or to the concurrent systems domain, even applied to *PIM*s, other formal techniques and an appropriate approach are required for the two aforementioned modules. These techniques are the main focus of this paper and they are presented and illustrated in Section 5.

2.3 The Instantiation Process

In order to turn the *extended MDA architecture* useful for verifying a model transformation, it is necessary to instantiate it with specific techniques and tools according to the paradigm of the involved models. Figure 3 is an UML activity diagram that represents the workflow of this process. Each swimlane contains specific activities for the existing actors: Solution Provider and MDA Client. The Solution Provider is responsible for providing a complete solution for the MDA client, hiding formal concepts and complicated aspects of techniques and tools.

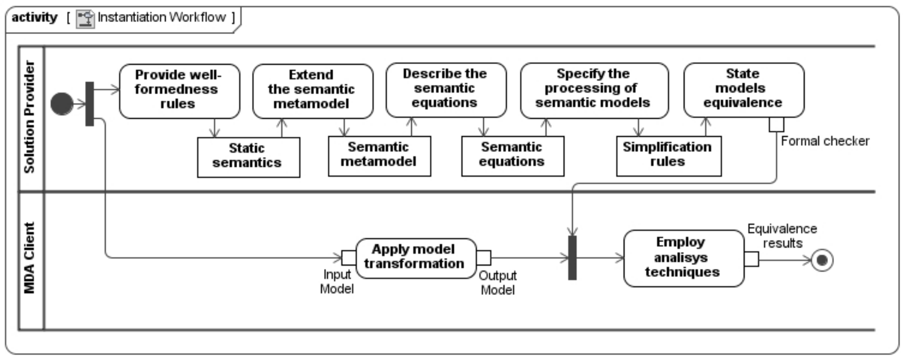

activity [Instantiation Workflow]
Solution Provider
MDA Client
Provide well-formedness rules
Extend the semantic metamodel
Describe the semantic equations
Specify the processing of semantic models
State models equivalence
Static semantics
Semantic metamodel
Semantic equations
Simplification rules
Formal checker
Apply model transformation
Input Model
Output Model
Employ analisys techniques
Equivalence results

Figure 3: Overview of the instantiation process

The activities of the Solution Provider concern to fill the architecture with specific techniques. This is better detailed in the highest part of the figure. First, the activity of providing well-formedness rules produces the *static semantics* for the models to be analyzed. This static semantic guarantees that we can deal only with correct models in our solution. Next, in order to provide *semantic models* the *semantic metamodel* must be specified with a specific theory according to the paradigm of the analyzed models. After this, the description of *semantic equations* is necessary for the automatic extraction of these *semantic models.* The processing *semantic models* is the next activity producing the simplification rules, responsible for the dynamic semantics of the models. Finally, the Solution Provider has to choose a comparison approach to check the equivalence between *semantic models.* The end of this activity produces the *formal checker.*

The product of all these subactivities is an instance of the *extended MDA architecture* for the paradigm of the involved models. Therefore, when the MDA Client has applied the model transformation for the input model, the output model is captured, which together with the input model are to be submitted to the last activity in the prototype in order to employ the analysis techniques of equivalence between these models. The equivalence result will enable detecting whether the transformation is semantics-preserving or not.

3 Case Study

In this section we introduce a case study conducted to evaluate our approach. This case study is part of a project called *FORDESIGN* [Gomes et al, 2005], that proposes an *easy-to-reason* Petri net based language for concurrent systems [Gomes et al. 2007] to describe *PIM*s in conformance with the *PNML* (Petri Net Markup Language) metamodel [NWeber and Kindler 2003]. In the context of this project, the Splitting Operation [Costa and Gomes 2007] was designed to contribute to the usage of Petri nets as the system-level specification language within the framework of hardware-software co-design of embedded systems, supporting system model partitioning into components. In this sense, the splitting operation performs a PIM-to-PIM transformation, starting from a centralized model and generating a set of concurrent sub-models allowing distributed execution of the system.

3.1 Petri Nets

Petri nets is a graphical and mathematical formalism for modeling concurrent systems [Girault and Valk 2003]. Its basic definition, also called Place/Transition nets, is defined as a 4-tuple $\langle S, T, F, W \rangle$, in which:

- S is a set of places.
- T is a set of transitions.
- $F \subseteq (S \times T) \cup (T \times S)$ is the set of arcs.
- $W : (S \times T) \cup (T \times S) \rightarrow N$ is the weight function. N denotes natural numbers.

The behavior, dynamic semantics, of a Petri net involves the concept of marking, which is a function that associates to each place a non-negative integer, called *token.* So, a marked Petri net is a tuple $\langle$ PN, M_0 $\rangle$, where PN is a Petri net and M_0 is the *initial marking.* Henceforth, the terms Petri net and marked Petri net are used interchangeably. For the sake of simplicity, let us consider that all the arcs have weight 1.

Figure 4 shows the graphical representation of a simple Petri net with two places (circles $P1$ and $P2$) and one transition (rectangle $T1$). Focusing on $T1$, $P1$ is an input place whereas $P2$ is an output place. The initial marking is $M_0 = (1, 1)$, indicating that there is one token in place $P1$ and one token in place $P2$ (represented as black dots inside places in Figure 4).

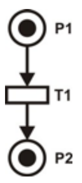


Figure 4: Example of Petri net

For operational analysis of Petri nets, a state is composed of the current marking, which is stored in a data structure called multiset. The dynamic behavior of a Petri net is defined as a relation on its markings (states). A transition of a Petri net may fire whenever there is a token in all of it input places. When it fires, tokens from input places are consumed and new tokens are added in the output places. The tokens in places that are neither input nor output places remain unchanged. A firing is atomic, *i.e.*, a single non-interruptible step. Considering the initial marking M_0 for the Petri net in Figure 4, transition $T1$ is enabled to fire. After firing, one token is removed from place $P1$ and one token is added to place $P2$. The new marking is $M_1 = (0, 2)$.

3.2 The Splitting Operation

The Splitting operation is based on the definition of a valid cutting set that promotes the division of a Petri net model into several sub-models, which communicate through synchronous channels. The decomposition of the model is achieved using a set of three rules (Rule #1, #2 and #3). Rule #1 is applied when the node in the cutting set is a place. The remaining two rules, Rule #2 and Rule #3, are applied when the node in the cutting set is a transition.

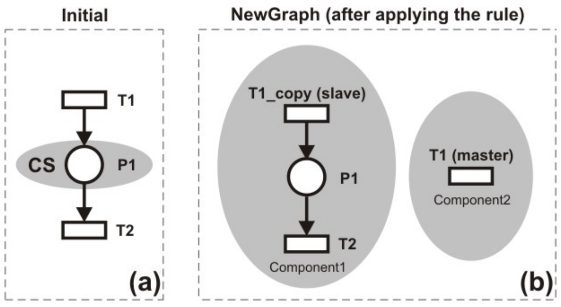


Figure 5: Application of the Rule #1

The application of Rule #1 is illustrated in Figure 5. The initial net is composed by two transitions and one place (Figure 5.a). By choosing to remove place

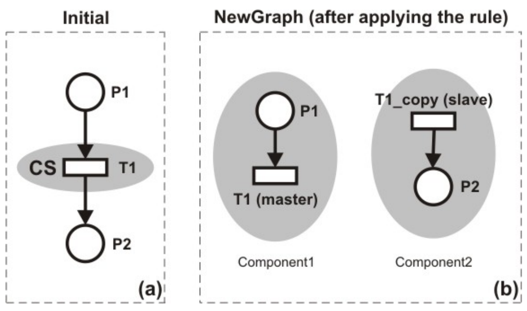


Figure 6: Application of the Rule #2

P1 (cutting set CS = {P1}) from the initial net, we have the new generated splitted net (Figure 5.b).

The application of Rule #2 is illustrated in Figure 6. The initial net is composed by two places and one transition as depicted in Figure 6.a. The cutting set node is CS = {T1} with input arcs from only one component. The result of the splitting operation is shown in Figure 6.b.

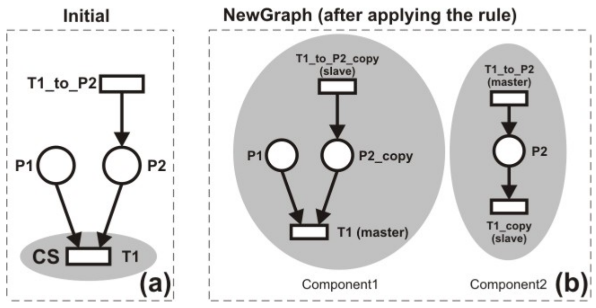


Figure 7: Application of the Rule #3

The application of Rule #3 is illustrated in Figure 7. It is applied whenever the cutting transition has input arcs from nodes which belong to different subnets after node removal. After splitting, one component will receive the attribute master while the other ones will receive the attribute slave. From the initial net (Figure 7.a) and CS = {T1}, the result of the splitting operation is illustrated in Figure (Figure 7.b).

3.3 The Parking Lot Controller Example

The Petri net that is used as an example in the rest of the paper describes a parking lot controller. The parking lot has room for four cars, one entrance and one exit. The Petri net that models the parking lot controller is shown in Figure

8. The initial marking is defined with one token in places `EntranceFree` and `ExitFree` and four tokens in place `FreePlaces`. It means that there is no vehicle inside the parking lot and entrance and exit gates are free. The entrance and the exit procedures are modeled using, respectively, the left and right parts of the model. When the entrance gate is free and a car arrives, transition `arrive+` fires and a token is removed from place `EntranceFree` and a token is deposited in place `WaitingTicket` indicating that a car is in the way to enter in the parking lot. Then, transition `Enter` is enabled to fire and after firing, a token is removed from place `FreePlaces` and one token is put on place `CarInsideZone`. This new marking indicates that there is one car inside the parking lot and still room for three more cars. Similarly, firing transition `Exit` indicates that a car is leaving the parking lot and the number of free places is incremeted by one.

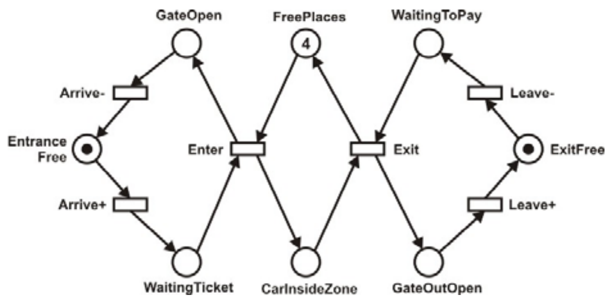


Figure 8: The net that models the parking lot controller

In order to illustrate the splitting operation, let us apply Rule #3 in the Parking lot controller Petri net model. Considering that transition `Enter` was in the cutting set, the application of splitting operation will produce a copy of transitions `Arrive+` and `Enter`. The remaining transitions are not copied.

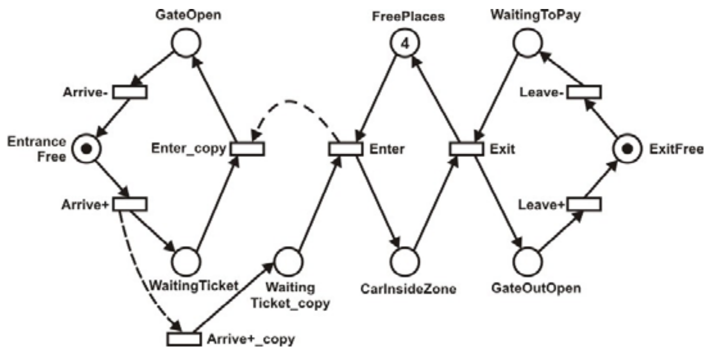


Figure 9: The splitted model after application of Rule #3

The generated splitted Petri net model is presented in Figure 9.

4 Introducing Concurrent Models in the Extended Architecture for Petri Nets

In this section we provide a complete overview of the extended MDA architecture instantiated to deal with concurrent models. In this scenario, the *semantic metamodels* must specify the *state structure* and the dynamic semantics of the concurrent model, *i.e.*, they must describe the elements captured from the semantic representation of the concurrent model involved in the transformations. A *semantic model* must be generated from the syntactic concurrent model and it must be in conformance with its state structure previously specified by the *semantic metamodel.* This generation is automatically performed by *semantic equations* introduced in the MDA architecture by transformations that state how to map each syntactic element from the concurrent model into its respective semantic elements. This process is applied to the input and output models in order to obtain an input and an output semantic model, respectively.

It is important to emphasize that we have two kinds of MDA transformations in the extended architecture: (i) Those responsible for generating the syntactical output model from the syntactical input model as prescribed by the MDA infrastructure; and (ii) Those responsible for mapping syntactical elements into semantic elements. The former are those to be proved as semantic-preserving transformations, whereas the latter are prescribed in the extended architecture as the way to automatically provide semantic models, essential for the proof task.

Once the *semantic models* have been built from the input and output models, the inference and computation of state configurations are performed by means of *rewrite rules* in order to allow proving some properties by the *formal checker* in order to state the equivalence (or not) of these models. These rewrite rules are the *simplification rules* that instantiate the dynamic semantics since they are concrete representations of the behavior of the concurrent model.

Any application involving concurrent models can deal with the extended achitecture since all its artifacts are provided. For instance, the Petri nets models and the splitting operations illustrated in Section 3 can be put together in the extended architecture, where: (i) Petri nets are concurrent platform independent models; (ii) splitting operations are PIM-to-PIM transformations that map a centralized model to a set of concurrent sub-models allowing distributed execution of the system; (iii) synctactical elements of Petri nets are specified by the *PNML* (Petri Net Markup Language) metamodel [NWeber and Kindler 2003], whereas (iv) semantic elements are specified by the category theory metamodel that describes its mathematical structures and the relations between them; and finally (v) ATL rules automate the generation of (iv) from (iii);

The execution of soundness proofs for the PIM-to-PIM transformations specified in (ii) is realized by the formal checker that must be able to verify whether the input and output models preserve some properties like liveness and deadlock. However, these proofs only can be done since a rewrite rule engine, like the Maude rewrite system [Clavel et al. 2000], perform the inference and computation of configurations of model states, implementing refactoring laws of the corresponding *semantic models.* Therefore, the formal checker is complementary to the rewrite rules.

Figure 10 illustrates how these pieces fit together in the extended MDA architecture.

As previously mentioned, the module *Formal Checker* in the extended MDA architecture requires different formal techniques according to the models, domains or specific platforms which it is employed. Therefore, for applications involving transformations from or to the concurrent systems domain, even applied to *PIM*s, other formal techniques and an appropriate approach, like *bisimilarity* through *model-checking*, are required. This is the point this paper is focused on. While the following subsections give details for each specific realized module, Section 4 details the proposed techniques to be implemented in the formal checker module in order to verify soundness of the MDA transformations applied to concurrent models.

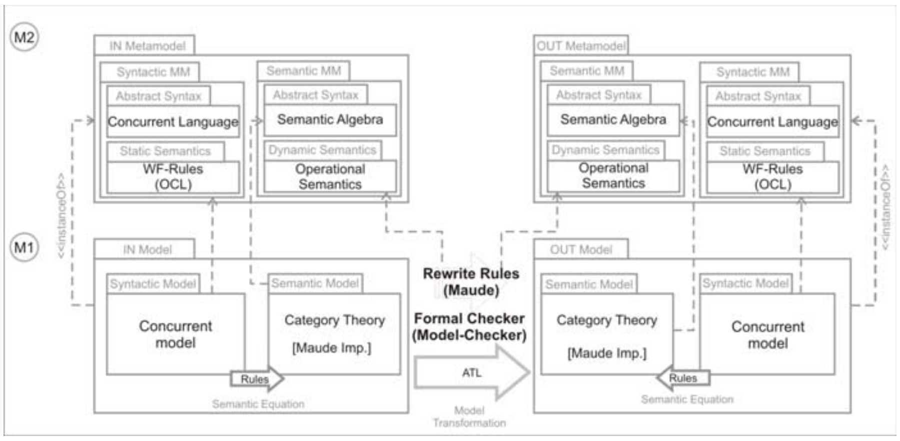


Figure 10: Instantiating the extended MDA architecture for concurrent models verification nets models

4.1 PNML Metamodel

Figure 11 describes the *Petri net metamodel.* It is responsible for the abstract syntax of Petri nets. A state is a *Configuration* which is composed by many fragments of *Petri net* (many because in some formalisms, the nets are splitted). A *Petri net* by a set of *Place*s, a set of *Transition*s and a set of *Arc*s. Finally, an *Arc* can link a *Place* to a *Transition* (*PlaceToTransArc* or a *Transition* to a *Place* (*TransToPlaceArc*).

4.2 Semantic Metamodel and Models

For the definition of the *semantic models*, we view Petri nets as ordinary, directed graphs equipped with two algebraic operations corresponding to parallel and

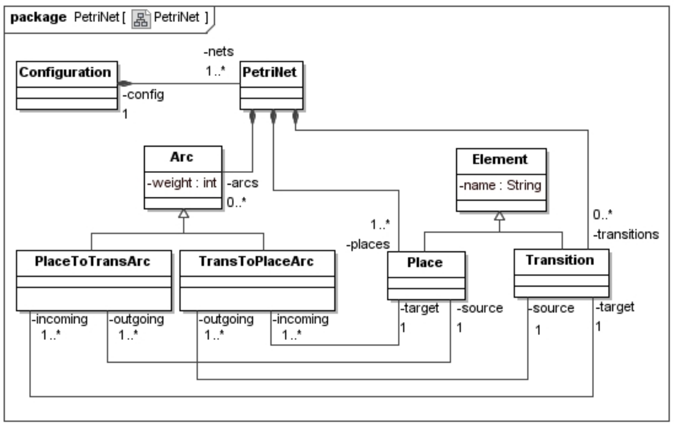


Figure 11: Excerpt of the syntactic metamodel for Petri nets

sequential composition of transitions. As an example, we can see the two models produced by the operation described in Figure 6 as an ordinary graph whose set of nodes is an algebraic structure generated by the set of places and the morphisms are produced by the transitions. Given that we have one token in each place (`P1` and `P2`):

– $M_{in}\ T1 : P1 \oplus P2 \rightarrow 2'P2$

– $M_{out}\ (Comp_1\ T1 :\ P1)$ **;** $(Comp_2\ T1\_copy :\ P2)$
  $\rightarrow\ (Comp_2\ 2'P2)$

In this case, it was necessary to adapt two algebraic operations inherited from [Meseguer and Montanari 1988]: $\oplus$ and **;**. The former corresponds to the union of elements and the latter to the concatenation of arcs as sequential composition. This means that in the output, the main syntactic observed differences are: there are two fragmented net components, it is not allowed the union operation of multisets over the markings of the places `P1` and `P2` and there now exists a sequential composition between the transition `T1` and the new inserted transition `T1_copy`. The *category theory* seems useful to explain these aspects using a formal algebraic notation.

The *Semantic Algebra Metamodel* proposed in [Barbosa et al. 2008(a)] is extended in Figure 12 in order to incorporate particular concepts of the categorical representation of Petri nets. The new inserted metaclasses are in gray. Petri nets models are viewed as *Graph*s, or more precisely *CategoryGraph*s. A *Graph* has two *FunctionSpace*s corresponding to the *source* and *target* functions. A *CategoryGraph* has its structure described as a *CommutativeMonoid*, that extends the algebraic type *Monoid* and has the *Identity*, *Idempotence*, *Closure*, *Associative* and *Commutative* properties. The morphisms of the *CategoryGraph* are implemented as *Rules* from *Operations*.

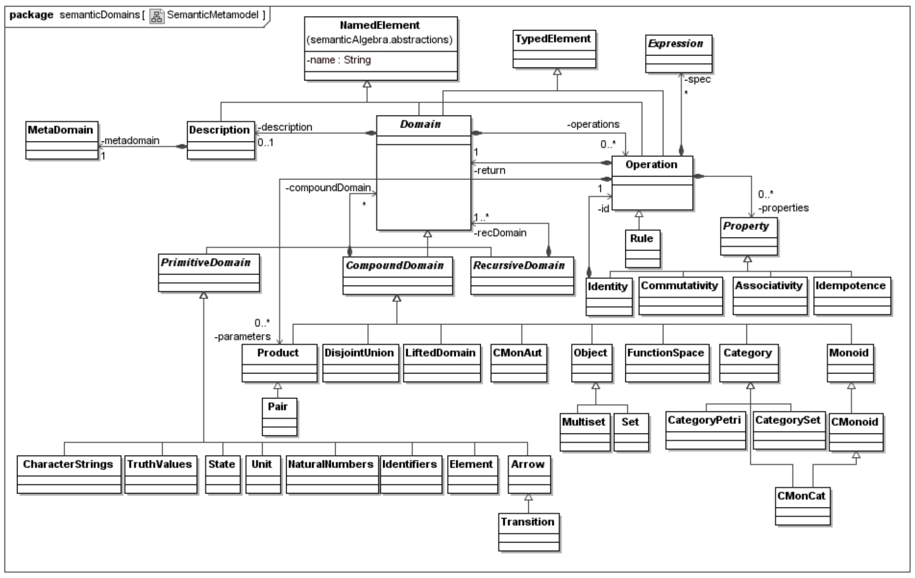


Figure 12: Excerpt of the Extension of the Semantic Algebra Metamodel for the Category Petri

The semantic model is a concrete representation of a *CategoryGraph*, which is composed by a *Monoid*, that satisfies its basic properties, as *Identity*, *Closure* and *Associativity* plus the *Commutative* property as a *CommutativeMonoid*.

As a result, any semantic model representing the semantic domain of a given Petri net must be in conformity with the aforementioned semantic metamodel. Thus, we guarantee that these semantic models are well-formed models. In addition, the semantic metamodel is essential to the task of automatically generating the semantic models from syntactic models since they are pivotal elements on which underlie the MDA tranformations rules by means of which we introduce semantic equations in the architecture.

4.3 Semantic Equations

In this section, we discuss our approach to introduce *semantic equations* as formal specifications in the MDA infrasctructure. We reuse elements of the *extended MDA architecture*, such as a semantic codomain and semantic models. Thus, we consider MDA transformations as the pivotal element to specify semantic equations with a well-established and appropriate extraction from the syntactic constructs to semantic domains. Although QVT is the current OMG's purpose for specifying transformations in the MDA vision, we have adopted ATL for specifying the *semantic equations*. This is due main because QVT tools have still low robustness and its use is not widely disseminated. On the other hand, ATL has a framework widely used by an increasing and enthusiast community, with full support to model operations, where, in an integrated way, one can specify and instantiate metamodels as well as specify and execute transformation on them. In addition, ATL has a wide set of transformation examples available at the literature and discussion lists, in contrast of QVT, whose documentation is poor and not didactic.

The next code fragment describe a part of the semantic equations for the Petri nets domain (line 1). It addresses the presented issues for *semantic equations*, mapping from a PNML to a SemanticAlgebra (line 2). It presents the rule *ConfigToSemAlgebra* (line 3) which is the entry point of our transformation. It creates, from a Petri net *Configuration* (lines 4-5), the main structure of a *SemanticAlgebra* (lines 6-7), which contains a *CategoryPetri* (lines 8-10). The *CategoryPetri* has a name that is retrieved from the *Configuration*'s *PetriNet* name. It also has compound domains. They are: a *Multiset* composed of *Rule*s, two *FunctionSpace*s and a *CMonoid* (commutative monoid) (lines 11-14). The imperative part of the rule (lines 15-22) iterates over the transitions of the nets to map to a rule, creates the *FunctionSpace*s of the arc input and output functions and fills the commutative monoid of the markings.

```
1: module Petri2Semantics;
2: create SemanticAlgebra : SEM from PNML : PN;
3: rule Config2SemAlgebra {
4:   from
5:     input : PN!Configuration
6:   to
7:     sa : SEM!SemanticAlgebra(domains <- cp),
8:     cp : SEM!CategoryPetri(name <- input.nets->first().name,
9:     compoundDomain <- rules, compoundDomain <- fsIN,
10:    compoundDomain <- fsOUT, compoundDomain <- cm),
11:    rules : SEM!Multiset,
12:    fsIN : SEM!FunctionSpace(name <- 'in'),
13:    fsOUT : SEM!FunctionSpace(name <- 'out'),
14:    cm : SEM!CMonoid
15:  do {
16:    for (r in input.nets->first().transitions) {
17:      self.addRule(rules,r.name);
19:    }
20:    self.makeFunSpace(fsIN,fsOUT,input.nets->first());
21:    self.makeCMonoid(cm,input.nets->first().places);
22:  }
23:}
```

4.4 Dynamic Semantics

For dynamic semantics, we define the firing of a transition as a rewrite rule. We recover the definitions presented in Section 2.3 as the possible concrete reptoresentation of behavior to guide the analisys in the *formal checker*. The rewrite rule will be able to automatically detect the firing of a transition using pattern matching and their concrete representations will be presented in the concrete example that follows in Section 5.

5 An Approach for Checking Formal Semantics Equivalence for Concurrent Systems

In this section we give specific details about our proposed techniques for checking formal semantics equivalence in MDA transformations involving *PIM*s that describe concurrent systems. It makes use of the *extended MDA architecture* instantiated for the concurrency domain to guide the entire verification process. As previously indicated, the most significant efforts to cope with this architecture must be concentrated in the *Formal Checker* module. The *Formal Checker* module is the adaptable part of our approach that must be in accordance with the paradigm of the models under verification. The choice of the correct technique in this case is the main issue of technical work that requires expertise.

Our approach to define the *Formal Checker* module of the extended MDA architecture to cope with concurrent systems is supported by four formal techniques. The first one concerns the execution of the transformation, in which, for each applied rule, a syntactical analysis provides a *syntactic equivalence relation* between models. The second states that *semantic models* have an algebraic representation according to *category theory*. Finally, we reuse two techniques of the operational semantics approach: *bisimulation* [Park 1981] and *model-checking* [Clarke et al. 1990]. Bisimulation is used in order to show that two systems have the same behavior, enabling the composition of both state structures that represent the behavior of both systems. Model-checking algorithms are used to conclude that the particular instance of the transformation preserved the behavior correctly. All these techniques were chosen according to the requirements of the FORDESIGN project. We have realized that the properties for equivalence between models specified by this project could be easily expressed using these techniques (*e.g.* temporal logic from model-checking) and can be approached in a suitable way.

The following four subsections present details on how these four formal techniques are used in the definition of the *Formal Checker* module.

5.1 Syntactic Equivalence Relation

Most MDA transformations consists of a set of rules, and each rule assumes the format of a `from` declaration, which takes information from the input model (M_{in}) according to the input metamodel, and a `to` declaration, that puts the processed data to the output model (M_{out}), according to the output metamodel. The behavioral equivalence analysis extracts and stores syntactic observations in a special data structure called *equivalence table* in which every component relation is provided by the executed rules during the execution of the analyzed MDA transformation. To this purpose, the transformation language must support the declarative and imperative paradigms, allowing matching and iteration over all the metamodel elements. The equivalence table metamodel is shown in Figure 13. As stated in the figure, an equivalence table is a table composed by lines that are in turn composed of two ordered cells identifying one state from the input model and another one from the output model.

Concerning the verification of semantic equivalence between both the initial (M_{in}) and splitted (M_{out}) models for the Parking Lot Controller example, we are

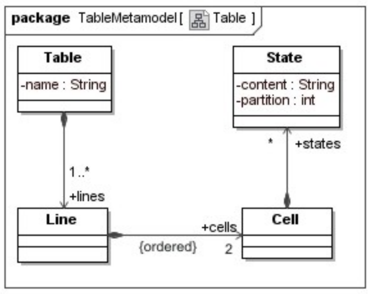


Figure 13: The equivalence table metamodel

able to apply the Splitting ATL transformation to the input model producing the *equivalence table* shown in Table 1. This produces a syntactic notion of equivalence between the two models, useful for the next steps.

Input Model	Output Model
Arrive-	Arrive-
Arrive+	Arrive+;Arrive+_copy
Enter	Enter;Enter_copy
Exit	Exit
Leave-	Leave-
Leave+	Leave+

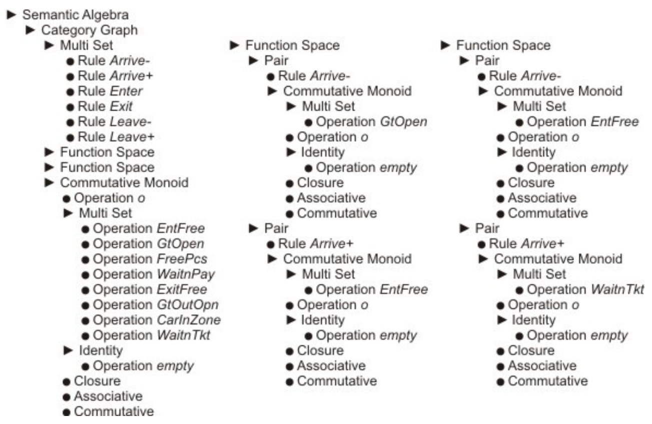
Table 1: Equivalence table for the Parking Lot Controller

5.2 Extracting the Semantic Representation of the Models

After applied the transformation and having available M_{in} and M_{out}, as well as the *equivalence table*, it is required a proper representation of the meaning of the models in order to apply formal techniques for verification of equivalence. The formal definition of the behavior is the first step to verify its preservation. One appropriate alternative to structure *semantic models* in the concurrency domain is the *category theory* [Lawvere and Schanuel 1997]. The category theory is an abstract way of dealing with mathematical structures and the relations between them. [Stehr and Csaba 2001] argue that it provides an abstract language for expressing very different models and allows the translation of constructions and properties between models via *adjunctions*, which are a way of describing a particular relationship between categories of algebraic structures. Therefore, the categorical view of models for concurrency provides the definition of behavioral equivalence between models.

Basically, categories consist of: (i) a set of objects, that can represent processes in the concurrency domain; and (ii) a set of morphisms, in which each morphism f has an unique source object a and an unique target object b ($f{:}a \rightarrow b$), and represents a relationship between one process and another, or the current behavior. Moreover, categories must have a binary operation (**;**), called *morphism composition*, satisfying the identity and transitivity properties, that could represent the process composition or the mapping of atomic arcs into whole computations.

For the Parking Lot Controller example, the application of the *semantic equations* in both models produces the *semantic model* as presented in Figure 14 in the Ecore format [Budinsky et al. 2003] for the input model, for example. In the first column we have the entire *semantic model* with four subitems that is a *Category Graph*. The first subitem is a multiset with the six rules that represent the behavior generated by the transitions. The second and third items are explored in the second and third columns. These columns shows some fragments of the *source* and *target* functions for each rule. Each one that has *CommutativeMonoids* as domain satisfies the required properties. Finally, the fourth item represent the node of the *Category Graph*. It is composed by a multiset of tokens that represents the places and the sum of these items also complies with the properties of a *CommutativeMonoid*. Each presented item is very important for the next step: the code generation as *Platform Specific Semantic Model* for verification purposes.



```
► Semantic Algebra
  ► Category Graph
    ► Multi Set
      ● Rule Arrive-
      ● Rule Arrive+
      ● Rule Enter
      ● Rule Exit
      ● Rule Leave-
      ● Rule Leave+
    ► Function Space
    ► Function Space
    ► Commutative Monoid
      ● Operation o
      ► Multi Set
        ● Operation EntFree
        ● Operation GtOpen
        ● Operation FreePcs
        ● Operation WaitnPay
        ● Operation ExitFree
        ● Operation GtOutOpn
        ● Operation CarInZone
        ● Operation WaitnTkt
      ► Identity
        ● Operation empty
      ● Closure
      ● Associative
      ● Commutative
```

```
► Function Space
  ► Pair
    ● Rule Arrive-
    ► Commutative Monoid
      ► Multi Set
        ● Operation GtOpen
      ● Operation o
      ► Identity
        ● Operation empty
      ● Closure
      ● Associative
      ● Commutative
  ► Pair
    ● Rule Arrive+
    ► Commutative Monoid
      ► Multi Set
        ● Operation EntFree
      ● Operation o
      ► Identity
        ● Operation empty
      ● Closure
      ● Associative
      ● Commutative
```

```
► Function Space
  ► Pair
    ● Rule Arrive-
    ► Commutative Monoid
      ► Multi Set
        ● Operation EntFree
      ● Operation o
      ► Identity
        ● Operation empty
      ● Closure
      ● Associative
      ● Commutative
  ► Pair
    ● Rule Arrive+
    ► Commutative Monoid
      ► Multi Set
        ● Operation WaitnTkt
      ● Operation o
      ► Identity
        ● Operation empty
      ● Closure
      ● Associative
      ● Commutative
```

Figure 14: Semantic Model for the net of the Parking Lot Controller

The *semantic models* are automatically represented in the Maude rewrite system, based on the second technique for the *Formal Checker*, the formalization of Petri nets using *category theory* for rewriting logic. The next code fragment is the Maude representation of the semantic model of the input model. All employed places in the net are operations (lines 4-5), Marking is defined as empty or as concatenation of other Markings (lines 6-7), and the existing transitions are generated as rules (lines 9-14).

```
1: mod INPUT-PARKING-LOT is
2:  sorts Place Marking .
```

```
3:  subsort Place < Marking .
4:  ops EntFree GtOpen FreePcs WaitnPay ExitFree
5:  GtOutOpn CarInZone WaitnTkt : -> Place .
6:  op empty : -> Marking .
7:  op __ : Marking Marking
8:  -> Marking [assoc comm id: empty] .
9:  rl[Arrive-] : GtOpen => EntFree .
10: rl[Arrive+] : EntFree => WaitnTkt .
11: rl[Enter] : FreePcs WaitnTkt => GtOpen CarInZone .
12: rl[Exit] : CarInZone WaitnPay => FreePcs GtOutOpn .
13: rl[Leave-] : GtOutOpn => ExitFree .
14: rl[Leave+] : ExitFree => WaitnPay .
15:endm
```

The next code fragment provides the semantic model of output model. Transitions `Arrive+` and `Enter` carry the messages to be exchanged between the nets (line 12), and their rules (lines 18-23) change concurrently the markings of both nets according to the specification. Complementary, the remaining transitions are represented as internal actions (lines 16-17 and 25-30) because their firing does not change any marking out of the scope of the owner net.

```
1:  mod OUTPUT-PARKING-LOT is
2:   inc CONFIGURATION .
3:   sorts Place Marking .
4:   subsort Place < Marking .
5:   ops EntFree GtOpen FreePcs WaitnPay ExitFree
6:   GtOutOpn CarInZone WaitnTkt WaitnTktM : -> Place .
7:   op empty : -> Marking .
8:   op __ : Marking Marking
9:   -> Marking [assoc comm id: empty] .
10:  op IOPT : -> Cid [ctor] .
11:  op m :_ : Marking -> Attribute [ctor gather (&)] .
12:  ops Arrive+ Enter : Oid Oid -> Msg [ctor] .
13:  vars Comp1 Comp2 : Oid .
14:  var C : Configuration .
15:  var Any : Marking .
16:  rl[Arrive-] : <Comp1:IOPT | m:GtOpen> =>
17:  <Comp1:IOPT | m:EntFree> .
18:  rl[Arrive+;Arrive+_copy] : Arrive+;Arrive+_copy(Comp1,Comp2)
19:  <Comp1:IOPT | m:EntFree> <Comp2:IOPT | m:Any>
20:  => < Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT | m:Any WaitnTktM> .
21:  rl[Enter;Enter_copy] : Enter;Enter_copy(Comp1,Comp2)
22:  <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT | m:Any WaitnTktM FreePcs>
23:  => <Comp1:IOPT | m:GtOpen> <Comp2:IOPT | m:Any CarInZone> .
25:  rl[Exit] : <Comp2:IOPT | m:CarInZone WaitnPay Any> =>
26:  <Comp2:IOPT | m:FreePcs GtOutOpn Any> .
27:  rl[Leave-] : <Comp2:IOPT | m:Any GtOutOpn> =>
28:  <Comp2:IOPT | m:Any ExitFree> .
29:  rl[Leave+] : <Comp2:IOPT | m:Any ExitFree> =>
30:  <Comp2:IOPT | m:Any WaitnPay> .
31: endm
```

5.3 Bisimulation and Bisimilarity

Bisimulation [Park 1981] is a widely employed technique for studying the structures that represent the behavior of concurrent systems. It is a binary relation between state-transition systems, that associates similar behavior in the sense that one system simulates other and the converse is also true. If we need to compare the equivalence of systems at a level that is not too fine-grained, we call this technique as *weak bisimulation.* In our context, a simulation can be derived through refinements of morphisms between algebraic structures, which are mappings intended to preserve structure.

Bisimilarity is the union of all bisimulations. It was proved as being a fundamental semantic equivalence in the algebraic theory of concurrent systems. The bisimilarity is employed to abstract some details of the systems. This abstraction enables to deal with infinite state spaces as equivalent finite structures. We define bisimilarity through a generic LTS (Labelled Transition System) as a tuple $(S, L, \varDelta, s_0)$ where: S is a set of states, with the initial state s_0, L is a set of labels, and $\varDelta \subseteq S \times L \times S$ is the transition relation.

Given a LTS, a binary relation $\mathcal{R}$ over the states of a LTS is a bisimulation if, always that $s_1 \mathcal{R} s_2$ happens, we have:

- For all s_1' with $s_1 \xrightarrow{\mu} s_1'$, there is a s_2' such that $s_2 \xrightarrow{\mu} s_2'$ and $s_1' \mathcal{R} s_2'$.
- The same for the transitions incoming from s_2.

From [Park 1981], bisimilarity, which is written as $\sim$, is the union of all bisimulations. Therefore, $s \sim t$ is true if there is a bisimulation $\mathcal{R}$ with $s \mathcal{R} t$.

Following the Parking Lot Controller example, we specify

```
eq initial =
(EntFree ExitFree FreePcs FreePcs FreePcs FreePcs)
```

as the initial state of the system model (input model) and

```
eq initial = Enter;Enter_copy(Comp1, Comp2) <Comp2:IOPT |
m:(FreePcs FreePcs FreePcs FreePcs ExitFree)>
Arrive+;Arrive+_copy(Comp1, Comp2) <Comp1:IOPT | m:EntFree >
```

as the initial state of the partitioned model. The partitioned model requires the specification of the channels `Enter;Enter_copy` and `Arrive+;Arrive+_copy` for the communication between `Comp1` and `Comp2`. The specification of the initial states is the first part of defining the bisimulation. The other part is the specification of equivalence for all states, which is based on the definition of marking. This second part is achieved through Model-checking technique and detailed in the next subsection.

5.4 Model-Checking

Model-checking [Clarke et al. 1990] is a technique to automatically verify formal models. After having a formal behavioral description of the model M, properties are specified in order to check their veracity. These specifications are generally built using propositional temporal logic as metalanguage. Let φ as an example of a property. The model-checker tool executes the process algorithmically and produces a truth value as result to indicate whether the specification was satisfied ($M \vDash \varphi$) or not ($M \nvDash \varphi$). In the case of the non satisfiability of this property, the tool must provide a list, called CE, from counter-example, of chained states ($s_0 s_1 s_2 ... s_n$) that were reached which demonstrate that the specification was not valid for this model.

In order to check if M_{out} of a MDA transformation is behaviorally equivalent to M_{in}, M_{out} must preserve the same properties of M_{in} that one should be interested in. This verification can be obtained through the application of model-checking at both levels according to the *equivalence relation* as a form of bisimulation. Given a property called φ that both models need to satisfy, we must have $(M_{in} \vDash \varphi) \Rightarrow (M_{out} \vDash \varphi)$ or $(M_{in} \nvDash \varphi) \Rightarrow (M_{out} \nvDash \varphi)$, with both counter examples, CE_{in} and CE_{out} being equivalent with respect to the chained states. They must respect the *equivalence table* for arcs between states and bisimulation laws. We admit $CE_{in} \subseteq CE_{out}$ in the case of M_{out} represent a concretization of M_{in} or $CE_{out} \subseteq CE_{in}$ in the case of a abstraction.

We Conclude the verification in the Parking Lot Controller example, performing the verification of semantic equivalence between the semantic models. In our case study, both models will be in equivalence if the following two main conditions are satisfied: (i) if the *deadlock freeness* and/or *liveness* occurs for the initial model then it must occur for the splitted model; and (ii) both models must preserve the same event order, *i.e.*, for a given sequence of transitions, they must produce traces with the same order of fired transitions, reaching the same final marking. In this sense, we perform the verification of the enumerated properties for each model as bisimulation through model-checking complementing the usage of the *Formal Checker* with its fourth technique.

Deadlock Freeness and Liveness. For the system model we use the command in line 1 of the next output fragment, starting from the initial marking. After this, the last line outputs `No solution.`, which means that our specification is *deadlock free.*

```
1: search in INPUT-PARKING-LOT : initial =>! Any:Marking .
2: No solution.
3: states: 54 rewrites: 102 in 0ms cpu (~rew/sec)
```

The same property must be satisfied for the generated partitioned components. By applying the command in lines 1-4 of the next output fragment, we ask if the partitioned system reaches a state with no successors. In the end, no state without successors is found, representing the same behavior.

```
1: search in OUTPUT-PARKING-LOT : Enter;Enter_copy(Comp1, Comp2)
2: <Comp2:IOPT | m:(FreePcs FreePcs FreePcs FreePcs ExitFree)>
3: Arrive+;Arrive+_copy(Comp1, Comp2))
4: <Comp1:IOPT | m:EntFree> =>! C:Configuration .
5: No solution.
6: states: 35 rewrites: 55 in 0ms cpu (~rew/sec)
```

Complementary to the *deadlock freeness* property, the *liveness* ensures that in both models, globally all states have markings with at least one enabled transition. The property `enabled` was specified again in a separated module using Maude. By observing the next two output fragments, both models show the same output when submitted to check this property.

```
1: reduce in INPUT-PARKING-LOT : modelCheck(initial, []enabled) .
2: rewrites: 8 in 0ms cpu (0ms real) (~ rew/sec)
3: result Bool: true
```

```
1: reduce in OUTPUT-PARKING-LOT :
2: modelCheck(initial, []enabled) .
3: rewrites: 26 in 0ms cpu (0ms real) (~ rew/sec)
4: result Bool: true
```

Preserving Events Order. The verification was performed automatically for all generated traces of behavior of the models. In order to show that a number of transitions can be fired in a given sequence, we use the firing of transitions `Enter, Arrive-, Arrive+, Leave+, Exit`. Lines 1, 2 and 3 in the next two output fragments illustrate how this is specified in LTL. The same sequence is fired, according to the *equivalence table* and both models reaches the same state. For example, the reached state is in line 13 for the `INPUT-PARKING-LOT` and at lines 22-23 for the `OUTPUT-PARKING-LOT`. Therefore, we conclude positively about the analyzed semantic equivalence according to bisimilarity.

```
1: reduce in INPUT-PARKING-LOT:modelCheck(initial,
2: ~ <> (Enter /\ O(Arrive- /\ O (Arrive+
3: /\ O (Leave+ /\ O Exit))))) .
3: rewrites: 255 in 5ms cpu (48000 rew/sec)
4: result ModelCheckResult:
5: {FreePcs FreePcs FreePcs FreePcs ExitFree WaitnTkt,'Enter}
7: {GtOpen FreePcs FreePcs FreePcs ExitFree CarInZone,'Arrive-}
9: {EntFree FreePcs FreePcs FreePcs ExitFree CarInZone,'Arrive+}
11:{FreePcs FreePcs FreePcs ExitFree CarInZone WaitnTkt,'Leave+}
13:{FreePcs FreePcs FreePcs WaitnPay CarInZone WaitnTkt,'Exit}
```

```
1: reduce in OUTPUT-PARKING-LOT :
2: modelCheck(initial,
3: ~ <> (Enter;Enter_copy /\ O(Arrive- /\ O (Arrive+;Arrive+_copy
4: /\ O (Leave+ /\ O Exit))))) .
5: rewrites: 92 in 3ms cpu (47000 rew/sec)
6: result ModelCheckResult:
7: {Arrive+;Arrive+_copy(Comp1, Comp2) Enter;Enter_copy(Comp1, Comp2)
8   Enter;Enter_copy(Comp1,Comp2)
9: <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT | m:(FreePcs FreePcs
10:FreePcs FreePcs ExitFree WaitnTktM)>,'Enter;Enter_copy}
11:{Arrive+;Arrive+_copy(Comp1, Comp2) Enter;Enter_copy(Comp1, Comp2)
12:<Comp1:IOPT | m:GtOpen>
13:<Comp2:IOPT | m:(FreePcs FreePcs FreePcs ExitFree CarInZone)>,
14:'Arrive-}
15:{Arrive+;Arrive+_copy(Comp1, Comp2) Enter;Enter_copy(Comp1, Comp2)
16:<Comp1:IOPT | m:EntFree> <Comp2:IOPT | m:
17:(FreePcs FreePcs FreePcs ExitFree CarInZone)>,'Arrive+;
18:Arrive+_copy}
19:{Enter;Enter_copy(Comp1, Comp2) <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT |
20:m:(FreePcs FreePcs FreePcs ExitFree CarInZone WaitnTktM)>,
21:'Leave+}
22:{Enter;Enter_copy(Comp1, Comp2) <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT |
23:m:(FreePcs FreePcs FreePcs WaitnPay CarInZone WaitnTktM)>,'Exit}
```

6 Related Work

The main related topic is the direct investigation of semantics preserving transformations. [Baresi et al. 2006] ensures that transformations, if given by rules, are seen as graph transformations. They demonstrate this by using the AGG tool in executable business processes. In [Ehrig et al. 2007], the information is preserved with the bidirectionality requirement. [Narayanan and Karsai 2008] goes directly to the investigation topic, by checking whether a graph transformation preserves the behavior of a given instance as input. As a first step, this conformance is analyzed for the reachability property through an equivalence relation between two graphs. This equivalence is obtained by bisimilarity, proving that the input model behaves exactly as the output model. The main difference of these approaches and ours is that we have the definition of semantic models through metamodels for the MDA infrastructure. They concentrate on transformations only as operational rules. Our work represents an evolution of these techniques according to the following views:

– *We go beyond graph transformations.* We propose to approach model transformations, with models being described according to several paradigms and metamodels. The only requirement is to belong to the MDA framework.

– *We formalize semantic models and metamodels.* In the previously mentioned works there was no extraction of the denotational meaning of models or graphs. There were only simulations. In our case, we cover since the inception of the meaning of the model, proposing semantic metamodels according to any formal technique and the correct instantiation of the models.

– *We are immersed in MDA.* We approach MDA model transformations, describe the semantic equations using MDA model transformation languages, reuse the MOF standard to describe the semantic metamodels, describe static analysis using OCL, and other techniques and tools. In this scenario, we apply MDA in the project for co-design of embedded systems as a complete methodology.

– *We check several properties.* In the previously mentioned work, only one or few specific properties are suggested as equivalence proof between the models. We deal with more sophisticated techniques, such as static and dynamic semantics verification, model-checking, theorem provers, in addition to bisimulation in order to obtain a complete proof according to several views of the transformation.

Some works combine formal verification techniques. [Ray and Sumners 2007] discuss an approach to enable the two disparate techniques, theorem proving and model checking, to complement one another. This work enables automation in invariant proofs, while preserving the expressiveness and control afforded by theorem proving. As a more specific approach, [Mokhati et al. 2007] presents a framework supporting formal verification of UML diagrams, by using object oriented and concurrent Maude capabilities. In the domain of programming languages, [Neuhausser and Noll 2007] uses the Rewriting Logic framework to the formal verification of programs written in the concurrent functional language ERLANG. The authors verify properties implementing the formalized operational semantics of this language.

Concerning projects, the TOPCASED [TOPCASED 2009] is close to this work because it developed a toolkit for critical applications systems development using open-source concepts, starting from several modeling languages as metamodels (namely UML, SDL, SysML, AADL, PDL) and use model transformation tools supported by an intermediate format, able to provide support and interoperability for a set of verification tools. There are verification tools also based on Petri nets. The main difference to our work is the formal semantics approach we have chosen to follow, including the conception of new formal modules for MDA, differing from the methodology of the TOPCASED. While TOPCASED, which relies in the Eclipse platform, is focusing in more pragmatic questions, such as allowing connection between model operations, dealing with the requirement of working with more than one tool conjointly and offering facilities of deploying, managing and communicating plugins we are interested in bringing more traditional concepts of formal semantics to MDA, attempting to provide representations of these theories in this framework.

Related to verification of Petri nets using Maude, [Farwer and Leuschel 2004] investigates Object Petri nets (OPNs), a class of Petri nets that provides a natural and modular method for modeling many real-world systems. [Stehr et al. 2001] proposes rewriting logic as a unifying framework for Petri nets models. It also proposes a mapping from the nets into rewriting logic specifications as assumed here. The main difference from our work, in this case, is that we propose the mapping as a MDA transformation, observing syntactic rules proposed in the *PNML* metamodel. In this sense, our approach is more pragmatic, although less general. The comparison of the Maude model-checker is made with several other tools. In [Kazuhiro and Kokichi 2007] it is compared with the SAL toolkit, a tool for analyzing transition systems with different tools. This work states clearly the main advantages provided by the Maude model-checker. It strengthens our choice for the Maude model-checker solution.

There are still several other works focusing on providing formal semantics to specific languages involved in the MDA infrastructure, such as MOF or UML. For instance, [Weisemoller and Schurr 2008] proposes the MOF formalization that can be useful in our approach for future works. In this work, the authors follow the same idea presented here of reusing the category theory modularization concepts from algebraic specification languages. This provides a graph-transformation-based formalization of MOF, upgrading MOF with new interfaces and a composition operator based on graph morphisms. It is the same idea reused here, however is a proposal for the meta-metalanguage of the MDA framework, and not for a specific domain as we made for Petri nets. [Kelsen and Ma 2008] proposes a novel approach for the definition of a formal semantics to languages based on the Alloy language. It intends to turn semantics definitions easier, encouraging the adoption of formal techniques. This work advocates that its approach provides two main advantages: uniform notation and a mechanism for analysis. However it is not enough to ensure that MDA transformations are semantics-preserving, as is our goal. Finally, [Arevalo et al. 2006] concentrates in some specific problems commonly found when using MDA such as refactorings and extraction of meanings of UML models. This work makes use of mathematical techniques, specially the lattices theory, to discover abstractions from a set of formal objects. This seems very useful for providing sound refactorings. The similarity with our work is that *semantic equations* can also be viewed as ATL transformations, when mapping a UML diagram to a formal context (the semantic model).

Conversely to these works, we tackle the whole MDA infrastructure, providing formal techniques to each one of the artifacts covered by the MDA four-layer architecture and integrating all these techniques in order to allow verifying if a transformation is semantics-preserving or not. In this context, the *extended MDA architecture* plays an important role, promoting the integration of formal techniques for all the MDA artifacts.

7 Conclusions and Future Work

We instantiated the extended MDA architecture that incorporates formal semantics for checking transformations involving models that represent concurrent systems. We described in details the method to check whether the execution of a certain transformation produces an output model preserving the semantics of the input model. We presented a case study evaluating our approach in a real case from the FORDESIGN project that adopts Petri net as concurrent models and implements the Splitting Operation as MDA transformations. In this case study, we have verified which of these transformations are semantics-preserving or not. Our approach was also successfuly applied to more compex Petri nets models from FORDESIGN project, considering different properties and transformations.

For analysis results, we have not evaluated characteristics of non functional requirements of the employed techniques (e.g. speed-up, state explosion, etc).

Our approach is almost fully automated. However, some efforts have still to be made when specifying the *equivalence table* in the transformation code and establishing the *bisimulation* axioms between states of the input and output models. Currently, we are analyzing ways to alleviate this sort of effort.

With regard to the chosen execution platform, although Maude tools have very efficient implementations, we are investigating the adoption of others tools. There are some platforms that arise as strong candidates for the generation of *semantic models*, such as the SPIN model-checker through the Promela language.

The main trend of future works concerns the validation of PIM-to-PSM transformations also proposed by the *FORDESIGN* project. They propose automatic code generation from Petri nets models to *ANSI C* and *VHDL* models in order to be executed in several platforms of embedded systems [Gomes et al, 2005]. This proposal requires further analysis in order to unify the *semantic models* extracted from these languages and specific aspects of the transformations.

References

[Arevalo et al. 2006] Arvalo, G., Falleri, J., Huchard, M. and Nebut, C.: "Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach"; In Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, Volume 4199/2006, 513-527.

[AtlanticZoo 2009] The Atlantic Zoo, 2009. `http://apps.eclipse.org/gmt/am3/zoos/atlanticZoo/`.

[Barbosa et al. 2007] Paulo Barbosa, Cassio Rodrigues, Jorge Figueiredo and Dalton Guerrero. Distributed Model-Checking: Investigating the Use of Computational Grids. Proceeding of the IEEE IECON'07, Pages 248–256. Taipei, Taiwan, 2007.

[Barbosa et al. 2008(a)] Paulo Barbosa, Franklin Ramalho, Jorge Figueiredo and Antonio Junior: "Incorporating Semantic Algebra in the MDA Framework"; Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT). Special Session on Metamodelling - Utilization in Software Engineering, 2008, isbn 978-989-8111-52-4, pages 330-336, Porto, Portugal.

[Barbosa et al. 2008(b)] Paulo Barbosa, Franklin Ramalho, Jorge Figueiredo and Antonio Junior: "An Extended MDA Architecture for Ensuring Semantics-Preserving Transformations"; Proceedings of 32nd Annual IEEE Software Engineering Workshop, 2008, Kassandra, Greece.

[Baresi et al. 2006] Luciano Baresi and Karsten Ehrig and Reiko Heckel: "Verification of Model Transformations: A Case Study with BPEL", TGC, Lecture Notes in Computer Science, 2006, vol. 4661, pages 183-199, Springer. http://dblp.uni-trier.de/db/conf/tgc/tgc2006.html#BaresiEH06.

[Bettin 2004] Bettin, J. : "Model-Driven Software Development: An emerging paradigm for industrialized software asset development"; 2004, http://www.softmetaware.com/mdsd-and-isad.pdf.

[Bezivin et al. 2003] J. Bezivin and E. Breton and Valduriez, Patrick and Dupr, Grigoire: "The ATL Transformation-Based Model Management Framework", IRIN, 2003 http://www.lina.sciences.univ-nantes.fr/Publications/2003/JBVD03.

[Bhm 1975] Corrado Bhm: "Lambda-Calculus and Computer Science Theory", Proceedings of the Symposium Held in Rome, Lecture Notes in Computer Science, 1975, vol. 37, Springer.

[Boronat 2006] Boronat, Artur, Carsí, Jos and Ramos, Isidro: "Algebraic Specification of a Model Transformation Engine" Fundamental Approaches to Software Engineering (2006), pgs 262-277 http://dx.doi.org/10.1007/11693017_20.

[Boronat et al. 2007] Artur Boronat, Jos . Carsí, Isidro Ramos and Patricio Letelier: "Formal Model Merging Applied to Class Diagram Integration"; Electron. Notes Theor. Comput. Sci. vol. 166 (2007), issn 1571-0661, pgs 5-26, Elsevier Science Publishers B. V., Amsterdam, The Netherlands http://dx.doi.org/10.1016/j.entcs.2006.06.013.

[Budinsky et al. 2003] Frank Budinsky and Stephen A. Brodsky and Ed Merks. Eclipse Modeling Framework. ISBN 0131425420. Pearson Education, 2003.

[Clarke et al. 1990] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled: "Model Checking", MIT Press, 1999, ISBN 0-262-03270-8.

[Clavel et al. 2000] M. Clavel, S. Eker, P. Lincoln and J. Meseguer: "Principles of Maude", Electronic Notes in Theoretical Computer Science, vol. 4, Elsevier Science Publishers, 2000.

[Costa and Gomes 2007] Anikó Costa and Luís Gomes: "Petri net Splitting Operation within Embedded Systems Co-design", Proceedings of INDIN (5th IEEE International Conference on Industrial Informatics, 2007.

[Ehrig et al. 2007] Hartmut Ehrig and Karsten Ehrig and Claudia Ermel and Frank Hermann and Gabriele Taentzer: "Information Preserving Bidirectional Model Transformations", FASE, Lecture Notes in Computer Science, 2007, vol. 4422, pages 72-86, Springer. http://dblp.uni-trier.de/db/conf/fase/fase2007.html#EhrigEEHT07.

[Farwer and Leuschel 2004] Berndt Farwer and Michael Leuschel: "Model Checking Object Petri nets in Maude and Prolog", PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, 2004, ACM, isbn 1-58113-819-9, pages 20-31, Verona, Italy. http://doi.acm.org/10.1145/1013963.1013970.

[France and Bieman, 2001] France, R. and Bieman, J. Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In Proceedings of Internacional Conference on Software maintenance (ICSM 2001).

[Girault and Valk 2003] Girault, Claude, Valk, Rudiger. Petri Nets for Systems Engineering, 2003, XV, 607 p., Hardcover. ISBN: 978-3-540-41217-5.

[Gomes et al. 2007] Luis Gomes and Joao Paulo Barros and Anikó Costa and Ricardo Nunes: "The Input-Output Place-Transition Petri Net Class and Associated Tools", Proceedings of the 5th IEEE International Conference on Industrial Informatics, 2007, pages 23-26, Vienna, Austria, IEEE Computer Society Press.

[Gomes et al, 2005] Luis Gomes, Joao Paulo Barros, Anikó Costa, Rui Pais, Filipe Moutinho; Formal methods for Embedded Systems Co-design: the FORDESIGN project; ReCoSoC05- Reconfigurable Communication-centric Systems-on-Chip - Workshop Proceedings; Gilles Sassatelli, Manfred Glesner, Lionel Torres, Leandro Soares Indrusiak, Thomas Hollstein (Editors); ISBN 2 9517 4611 3; 27-29 Junho 2005, Montpellier, France;

[Gomes and Costa, 2006] Luis Gomes, Anikó Costa; Petri nets as supporting formalism within Embedded Systems Co-design; SIES2006 - 2006 IEEE International Symposium on Industrial Embedded Systems; 18-20 October 2006, Nice, France; IEEE

Catalog Number: 06EX1451; ISBN 1-4244-0777-X

[Kazuhiro and Kokichi 2007] Kazuhiro Ogata and Kokichi Futatsugi: "Comparison of Maude and SAL by Conducting Case Studies Model Checking a Distributed Algorithm", IEICE Transactions, 2007, vol. 90-A, pages 1690-1703. `http://dblp.uni-trier.de/db/journals/ieicet/ieicet90a.html#OgataF07`.

[Kelsen and Ma 2008] Kelsen, P. and Ma, Q.: "A Lightweight Approach for Defining the Formal Semantics of a Modeling Language"; 2008, In Proceedings of the 11th international Conference on Model Driven Engineering Languages and Systems, Springer-Verlag, Berlin, Heidelberg, 690-704.

[Lawvere and Schanuel 1997] Lawvere, F.W. and Schanuel, S.: "Conceptual Mathematics: A First Introduction to Categories", Cambridge: Cambridge University Press, 1997.

[Meseguer and Montanari 1988] Jose Meseguer and Ugo Montanari: "Petri nets are monoids: a new algebraic foundation for net theory", Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science (LICS 1988), 1988, pages 155-164, Edinburgh, Scotland, UK.

[Meseguer and Rosu 2004] Jos Meseguer and Grigore Rosu: "Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools"; IJCAR, 2004, 1-44 `{http://springerlink.metapress.com/openurl.asp?genre=article{\&}issn=0302-9743{\&}volume=3097{\&}spage=1}`.

[Miller and Mukerji 2003] J. Miller and J. Mukerji: "MDA Guide Version 1.0.1"; Object Management Group (OMG), 2003.

[Mokhati et al. 2007] Farid Mokhati and Patrice Gagnon and Mourad Badri: "Verifying UML Diagrams with Model Checking: A Rewriting Logic Based Approach", QSIC, 2007, IEEE Computer Society, isbn 1-58113-819-9, pages 356-362. `http://dblp.uni-trier.de/db/conf/qsic/qsic2007.html#MokhatiGB07`.

[Narayanan and Karsai 2008] Anantha Narayanan and Gabor Karsai: "Towards Verifying Model Transformations", Electron. Notes Theor. Comput. Sci., 2008, vol. 211, issn 1571-0661, pages 191-200, Elsevier Science Publishers B. V.. `http://dx.doi.org/10.1016/j.entcs.2008.04.041`.

[Neuhausser and Noll 2007] Martin Neuhausser and Thomas Noll: "Abstraction and Model Checking of Core Erlang Programs in Maude", Electron. Notes Theor. Comput. Sci., 2007, vol. 176, issn 1571-0661, pages 147-163, Elsevier Science Publishers B. V.. `http://dx.doi.org/10.1016/j.entcs.2007.06.013`.

[NWeber and Kindler 2003] NWeber, Michael and Kindler, Ekkart: "The Petri Net Markup Language", Journal Petri Net Technology for Communication-Based Systems, 2003, pages 124-144 `http://www.springerlink.com/content/6b23lvlm7kl5g1l5`.

[OMG 2009] OMG(Object Management Group): "Model-Driven Architecture"; accessed 2009; `http://www.omg.org/mda/`.

[Park 1981] David Park: "Concurrency and Automata on Infinite Sequences"; Proceedings of the 5th GI-Conference on Theoretical Computer Science, 1981, 3-540-10576-X, 167-183, Springer-Verlag, London, UK.

[Ray and Sumners 2007] Sandip Ray and Rob Sumners: "Combining Theorem Proving with Model Checking through Predicate Abstraction", IEEE Des. Test, 2007, IEEE Computer Society Press, vol. 24, issn 0740-7475, pages 132-139. `http://dx.doi.org/10.1109/MDT.2007.38`.

[Scott 1970] Dana S. Scott. Outline of a Mathematical Theory of Computation, Programming Research Group, Technical Monograph PRG2, Oxford University, 1970.

[Scott and Strachey 1971] D.S. Scott and C. Strachey: "Towards a Mathematical Semantics for Computer Languages", Proceedings of the Symposium on Computers and Automata, 1971, vol. 21, Polytechnique Institute of Brooklyn.

[Stehr et al. 2001] Mark-Oliver Stehr and José Meseguer and Peter Csaba lveczky: "Rewriting Logic as a Unifying Framework for Petri Nets", In Unifying Petri Nets, Advances in Petri Nets, 2001, isbn 3-540-43067-9, pages 250-303, Springer-Verlag, London, UK.

[Stehr and Csaba 2001] Mark-Oliver Stehr and Peter Csaba: "Rewriting logic as a unifying framework for Petri nets", Unifying Petri Nets, LNCS, 2001, 250–303, Springer.

[TOPCASED 2009] The Topcased Project; accessed 2009; `http://www.topcased.org`.

[Wadsack and Jahnke 2002] J. P. Wadsack and J. H. Jahnke. Towards Model-Driven Middleware Maintenance. OOPSLA conference on Object Oriented Programming, Systems, Languages and Applications, 2002.

[Weisemoller and Schurr 2008] Weisemoller, I. and Schurr, A.: "Formal Definition of MOF 2.0 Metamodel Components and Composition"; In Proceedings of the 11th international Conference on Model Driven Engineering Languages and Systems, Springer-Verlag, Berlin, Heidelberg, 386-400.