Contribution to an equipped approach for the design of executable, verifiable and interoperable Domain Specific Modelling Languages for Model Based Systems Engineering

Blazo Nastov

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'UNIVERSITÉ DE MONTPELLIER

Préparée au sein de l'école doctorale Information, Structure et Systèmes I2S et de l'unité de recherche LGI2P de l'école des mines d'Alès

Spécialité : Informatique

Présentée par Blazo NASTOV

Contribution à une méthode outillée pour la conception de langages de modélisation métier interopérables, analysables et prouvables pour l'Ingénierie Système basée sur des Modèles

Soutenue le 15/11/2016 devant le jury composé de :

Mr. Eric Bonjour, Professeur, U. de Lorraine, ENSGSI	Rapporteur
Mr. Benoît Combemale, MdC HdR, U.de Rennes, IRISA	Rapporteur
Mr Loïc Lagadec, Professeur, ENSTA Bretagne, Lab-STICC	Examinateur
Mr. Jean-Michel Bruel, Professeur, U. de Toulouse, IRIT	Examinateur
Mr. Christophe Dony, Professeur, U. de Montpellier, LIRMM	Co-Directeur
Mr. Vincent Chapurlat, Professeur, IMT/EMA, LGI2P	Co-Directeur
Mr. François Pfister, MdC, IMT/EMA, LGI2P	Encadrant

CONTENTS

LIST OF FIGURES

List of Tables

CHAPTER I

CONTEXT AND PROBLEMATICS

1.1 General Introduction

This thesis is positioned in a twofold global context englobing system engineering (SE) and software engineering (SoE), and studies more precisely model-based development and its automation via dedicated, specific to a domain, modelling languages (DSML) allowing to design, check, verify, validate and simulate models of systems or software.

On the one hand, SE is an interdisciplinary and collaborative approach for the successful design and management of all kind of complex engineering systems. According to (INCOSE 2010), SE *provides the means for the realization of successful systems, focusing on customer needs and required functionality early in the development cycle, documenting requirements, and then proceeding to design synthesis and system validation while considering the complete problem*.

On the other hand, SoE is *concerned more specifically with developing and maintaining software systems that behave reliably and efficiently, are affordable to develop and maintain, and satisfy all the requirements that customers have defined for them* (Association for Computing Machinery 2015).

A current trend in both domains, SE and SoE, suggest the development of systems based on models. Within the SE domain, this trend is denoted as model-based system engineering (MBSE), whereas within the SE context is denoted model-driven engineering (MDE). Both MBSE and MDE evolve conjointly and pursue the some common goals:

- the development of automated and cost efficient solutions (INCOSE 2007; Combemale 2016);
- the multi-viewpoint modeling, verification and validation of systems where different viewpoints are used by different stakeholders (ISO/IEC 2008; OMG 2015b);

However, beyond this conjoint research evolution, we can identify several specificities.

SE tackles with globally larger, more heterogeneous and more complex systems, embedding material, physical as well as software entities; they notably deal with time in various ways (discrete / continuous / hybrid). The MBSE approach there is globally recent, especially for what concerns DSML execution and environments for the explicit manipulation of models and meta-models as well.

SoE has introduced model-driven engineering solutions earlier in time (Schmidt 2006), as a successor of computer-aided software engineering and now propose advanced solutions, languages and environments for the explicit handling of models but also of their metamodels, for model execution via executable DSML, for execution in a multiple viewpoint (on a system) context. These advantages are not yet fully integrated in the MBSE world.

The above quoted goals and specificities from the SE and SoE contexts define the problematics of this thesis.

- One the one hand there is a need to study and adapt, for MBSE, the recent advances coming from MDE on meta-modelling environments and on executable domain specific modeling language (xDSML).
- On the other hand we believe that the preceding study and notably the expected formalized solutions for verification and validation taking MBSE context into account (for example for the representation of time), will also provide, by a feedback analysis, some new contributions usable in the SoE field.

On this basis, the rest of the introduction and the chapter 2 then detail the specific problematic and expected contributions of this thesis.

1.2 SE challenges for MBSE

Within the context of organizational and engineering sciences, Systems Engineering (SE) is a key interdisciplinary and collaborative approach for the successful design and management of large scale complex systems. SE is today widely tested and used in the industry, being object of several standards such as IEEE 1220 (Doran 2006) and ISO/IEC 15288 (ISO/IEC 2008)), supported by tools (INCOSE 2016b) and currently applied in various domains, (e.g., transport, space, defense, health and energy). It involves designers and architects from different domains to design a "System of Interest" (SoI). A SoI is "*the system whose life cycle is under consideration"* (ISO/IEC 2008). Among other activities, for instance of project management, SE experts must be able to:

- *model a SoI* considering various points of views (denoted viewpoints) by designing and combining different models (at least one for each viewpoint), while respecting the stakeholder's specifications and the operational context of the SoI lifecycle;

- *formally prove* and *simulate* designed models;
- *test alternatives solutions*;
- *determine* and *justify* architectural decision, etc;

For this, SE provides concepts and principles related to System Thinking and System Sciences. It promotes various processes that offer adequate activities for system design, development, evolution and verification, delivering an optimal solution of the SoI (Doran 2006). These activities are based on models and modeling approaches. To this end, SE is applied in a model-based (or model-driven) context, denoted Model-Based Systems Engineering (MBSE). MBSE is *the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases* (INCOSE 2007).

MBSE promotes the creation and the management of various models of a SoI, each one focusing on a given aspect, i.e., viewpoints of a SoI (functional, logical, physical or behavioral). A model is *"a representation of an original system, i.e., a subject that might exist or not, containing at least one, but not all subject properties*" (Stachowiak Herbert 1973). In the MBSE context, models are designed to help experts in understanding a given SoI, as well as its behavior, and in performing various analyzes such as performance or non-functional properties also known as 'ilities (De Weck et al. 2012).

Based on designed models, experts make decisions about the SoI. It is thus imperative, prior to any decision to implement model verification and validation (V&V) activities (e.g., to justify architectural choice or to generate a test plan). The goal of the verification is to determine the correctness of a model based on the rules defined by the used modeling language. The goal of the validation is to argue the relevance and accuracy of a verified model, in representing a system as expected by stakeholders, respecting their needs and requirements. V&V activities are performed considering SoI models, first separately, and then together. When models are put together, they provide more complete and suitable representation of a SoI that includes models' mutual coherence as well as their adequacy and global fidelity to the SoI, in contrast to the information provided by one model.

Nonetheless, some of the MBSE objectives are still a subject of numerous debates and are quoted as challenges in the SE community (AFIS 2012). For example, Figure 1 gives an overview of the most significant uprising challenges in the field of SE.

Figure 1. Raising challenges in systems engineering (AFIS 2012).

We study hereafter the following:

(1) Modeling and simulation covering total system representation
(2) Verification, Validation and Qualification of complex systems
(3) Very large heterogeneous or autonomous systems: complexity management connections of in-use systems resilience
(4) Interoperability Via Integrated Architectures

The goal is to identify the objectives and current problems of each of the above selected SE challenges for MBSE and to contribute conceptually, methodologically and technically by adapting recent advances coming from MDE on meta-modelling environments and on executable domain specific modeling language (xDSML).

1.2.1 Modeling and simulation covering total system representation

When modeling a SoI, various interconnected viewpoint models are designed. Each model is dedicated and relevant for the needs of different stakeholders involved in the

design process. When all the viewpoint models are put together, they form a "composite model", covering a more expressive, realistic and complete representation of a SoI. In a similar way, the whole behavior of a SoI can be represented by mixing or aggregating the behaviors described by composing viewpoint models, even though these behaviors might be based on different functioning hypothesis (e.g., different level of details, different objectives, etc.). The V&V analyses become in this sense more relevant when considering composite models (e.g., a more realistic SoI simulation that coordinately executes all viewpoint models). However, the current MBSE modeling languages remain insufficient for the design and simulation of composite models.

For this purpose, two possible solutions can be adapted from the MDE context: General Purpose Modeling (GPM) and Domain Specific Modeling (DSM). GPM promotes the use of a General Purpose Modeling Language (GPML) for the modeling of different viewpoints of a SoI. A well-known example is the OMG's Unified Modeling Language (UML) (OMG 2011). DSM promotes the use of a Domain Specific Modeling Language (DSML) particularly tailored for a given problem, for the modeling of one viewpoint of a SoI that is used to solve a given problem.

The main difference between GPML and DSML is that the prior is used to model any SoI viewpoint for any problem, while the latter is used to model one particular SoI viewpoint for one well-defined problem. As a consequence, on the one hand, GPML provide generic concepts that are far from the end-user domain ontology. On the other hand, the genericity of GPML might overwhelm the end-use with many different ways to model an artefact. In contrary, a DSML integrates the end-used domain ontology, easing the understanding and use. Moreover, domain models are represented with an end-user friendly graphical or textual concrete syntax and provide a set of constraints dedicated to a considered domain problem that can be used to verify created models.

A customization of GPML is possible by using the UML profiles, however, obtained results remain restricted to predefined concepts and there isn't an easy way to integrate new concepts. This is inconvenient for the modeling needs and objectives of new stakeholders from different domains that have recently been added in an ongoing SE project. Namely, they would be unable to integrate their domain concepts in the current modeling environment and would be forced to use the existing concepts.

Considering the MBSE objectives and needs, this work focuses on the adaptation of DSML in MBSE context for modeling and simulation. A particular attention is given on

the semantics of DSML for "direct simulation". By direct we mean without transforming the models into external third party formal approach for simulation. The goal is to equip DSML with semantics that can furthermore be used for simulation. This kind of semantics is denoted dynamic semantics. We are considering coordinated simulation that manages all SoI models, even if created by different DSML. Such simulation mechanism must take into account the dynamic semantics of all DSML involved in the modeling of a SoI.

1.2.2 Verification, Validation and Qualification of complex systems

Among other objectives, MBSE focuses on *Verification and Validation (V&V)* activities during design process. Often called Early V&V activities, they are indeed crucial, prior to the Integration, Verification, Transition and Validation (IVTV) plan and the Qualification of a system (INCOSE 2016a), during which a SoI is implemented and after which it can be delivered to customers.

V&V are performed considering each individual viewpoint model of a SoI, first separately and then together, forming the previously discussed composite model. The here-considered "*Early Verification and Validation of complex systems*" aims to assure that: 1) each model respects the modeling rules defined by a metamodel (i.e., a model must conform to a metamodel), 2) each model is correctly represented by the mean of the representation rules defined by the concrete syntax of the used modeling language, 3) each model is well-formed, respecting the well-formedness rules defined by the static semantics of the used modeling language, 4) each model respects the needs and modeling objectives of stakeholders, i.e., is build taking in consideration the stakeholders' requirements, and 5) each model behaves correctly, i.e., its behavior provides a realistic vision of the SoI evolution and dynamics during simulation.

For this purpose, we propose to study and adapt the MDE vision on the composition of DSML (i.e., DSML syntax and DSML semantics) and on the correctness of a model based on the latter, including the means for model conformity, correct representation, simulation and formal property proof. A particular attention is given on the specification of stakeholders' requirements as formal properties and on the formal proof, i.e., the verification of such properties. We focus on "direct property proof", i.e., without transforming the SoI's viewpoint models into external third party formal approaches. Properties are designed by using a property modeling language to specify additional

characteristics that cannot be implicitly specified by the composition of a DSML (i.e., its structure, representation or behavior). The property proof must be achieved based on a model, considering also the other models of a SoI.

1.2.3 Very large heterogeneous or autonomous Systems

We focus here particularly on the modeling and model V&V of very large systems and on early complexity management based on models.

The modeling of very large heterogeneous or autonomous system involves a huge (possibly increasing) number of stakeholders in modeling activities. The new modeling activities of the new stakeholders must be included to the already supported activities, providing the means to design new viewpoint models and to interconnect these models with the already existing ones, automatically increasing the volume of modeled information of a SoI. In addition, the new viewpoint models must be considered during V&V activities (i.e., simulation and formal proof).

This leads to a huge number of DSML that have to be dynamically integrated with the already operating ones. For this purpose, we propose to study and adapt from the MDE context, a multi-viewpoint approach that allows such dynamic integration of new DSML and model V&V activities as previously discussed.

1.2.4 Interoperability Via Integrated Architectures

We focus here on providing the means for model interoperability considering several interconnected viewpoint models in a composite model.

We consider two types of interoperability between modeling languages (DSML) and between models:

- Syntactical interoperability and
- Semantical interoperability

As previously discussed, first DSMLs are created and interconnected. The interconnection consists in designing the syntactical dependencies and the semantical dependencies between different DSMLs, making them syntactically and semantically interoperable. Second, in a similar way, models created by using such DSMLs can be syntactically and semantically bound together. On the one hand, syntactically interoperable models represent the modeling covering total system representation. They

represent, not only the different aspects of a SoI, but also the syntactical interactions and dependencies between these aspects. On the other hand, semantically interoperable models can be coordinately simulated, representing a simulation covering total system representation, and can be used altogether as a base for formal proof. Such simulation and proof are much more relevant and accurate that the simulation and proof based on one model, because it takes in account all different aspects of a SoI and the semantical interactions and dependencies between these aspects.

1.3 MBSE and MDE: Identification of common issues and possible alignment

Several attempts to solve similar problems as the above discussed have been introduced in the field of Software Engineering. Similarly to MBSE, Software Engineering promote Model Driven Engineering (MDE) (Schmidt 2006) principles and practices that are concerned with modeling and early verification and validation (V&V) needs, activities and problems oriented to improve software development processes. MDE focuses on software systems in contrast to MBSE that tackles with globally larger, more heterogeneous and more complex systems, embedding material, physical as well as software entities; they notably deal with time in various ways (discrete / continuous / hybrid). However, the MBSE approach there is globally recent, especially for what concerns DSML execution and environments for the explicit manipulation of models and metamodels as well.

This thesis aims at adapting and improving MDE principles that might be of benefit for addressing, partially or completely the SE challenges discussed in the previous Section.

For instance, within the software engineering community, the GEMOC initiative (Combemale 2016) aims at “coordinating and disseminating the research results regarding the support of the coordinated use of various modeling languages that will lead to the concept of globalization of modeling languages, that is, the use of multiple modeling languages to support the socio-technical coordination required in systems and software engineering”. Namely, they highlight the problems of modeling and simulation covering total system representation by various and heterogeneous DSMLs, model V&V, i.e., coordinated simulation of models, simulation trace and analyses verification and proof of properties, etc.

In the MDE context, DSMLs are specified by their syntax and semantics (Kleppe 2007). A DSML syntax defines concepts of a domain and its relationships, denoted abstract syntax, and the way instances of these concepts are going to be (graphically or textually) represented, denoted concrete syntax. However, the key limitation for model V&V in the MBSE context is that DSML semantics is often neglected or, when needed, provided by means of transforming the DSML into third-party formalism (Chapurlat 2013).

The DSML semantics can be divided into static semantics, representing concept meaning and behavior and structural constraints (e.g., invariants pre and post conditions, derivations, etc.), and a dynamic semantics, specifying DSML behavior.

First, static semantics are formalized as a set of properties. Property proof is generally achieved based on transformation mechanisms but this technique leads to information loss, especially for composite models. Indeed, on the one hand, each of the viewpoints models must be correctly transformed into a single formal specification. On the other hand, achieved results must be correctly translated back and interpreted for each of the originating viewpoint models.

The MBSE issues addressed in this work are the specification of properties and their direct verification based on a composite model without using model transformations.

Second, dynamic semantics can be specified either as operational semantics, by using an action language (e.g., Java) or a behavioral modeling language (e.g., Statechart), or as translational semantics by using model transformation approaches (e.g., ATL). In both cases, DSMLs can be used to execute models and are thus denoted executable DSMLs or xDSMLs.

The focus of this thesis is to study operational semantics for MBSE. Operational semantics allows the specification of behavior directly on concepts, allowing simulation and animation, as early as possible with minimum of effort, improving system quality and reducing time-to-market. Nevertheless, the MBSE issues addressed in this work are

(1) to provide the means for designing DSML operational semantics for the MBSE context;
(2) to coordinately use operational semantics of different DSMLs for simulation that is based on all interconnected models of a SoI;

In addition, this thesis aims at unifying the design of different parts of modeling languages (abstract syntax, concrete syntax, static semantics and dynamic semantics) and models based on the concept “Property”.

1.4 Problematics and Objectives of this thesis

As previously shown, in the context of MBSE or MDE, models are to be created and managed, checked and simulated prior to any use for discussion, deliberation or decision. Models must support stakeholders and increase their confidence during decision making processes. Made decisions impact the development of the real system, up until its deployment and exploitation, i.e., system’s functioning, safety, security, induced costs, and so forth. It is thus very important to assure the quality of models before making any decision by applying model verification and validation (V&V) activities. So domain specific modeling languages (DSMLs) are requested for the design and management of various models each highlighting a viewpoint of the SoI, but also requested to apply various V&V techniques during SoI engineering process.

However, creating models that represent a SoI and reach and maintain a certain level of quality, as imagined by different stakeholders, faces currently several ongoing issues in the field of MBSE. This thesis contributes on the matter, focusing on two general problems: *(1) the design of modeling languages* and *(2) the verification and validation of models*.

The objective of this work is to develop a method for the design, verification and validation of models that are used by stakeholders to understand a SoI, to communicate and argue with other actors about this SoI and finally to support them and increase their confidence during decision making processes.

The method must address the above selected SE challenges in a MBSE context by considering, adapting and improving solutions coming from the MDE context. In particular, it must assure the autonomy of different stakeholders involved in the process of complex system modeling, during the process of designing, intuitively and as simple as possible, models that contain their domain knowledge, but also to verify and validate these models.

The work presented throughout the rest of this manuscript converges through the proposal of a method for the design, the verification and the validation of models. To this end, our method must first guide and assist stakeholders to design their own

modeling languages, particularly tailored for their domain knowledge and used to model a particular viewpoint of the SoI, named domain specific modeling languages (DSMLs). Second, DSML must be usable for the design of models, but also, on the one hand, for the simulation of models, and on the other hand, for the specification and verification of formal properties based on designed models.

The problematics and expected contributions are furthermore detailed at the end of Chapter II, after introducing the state of the art.

1.5 Outline of the manuscript

This manuscript describes the main components of the proposed method. It is structured as follows:

The Chapter II presents **the state of the art** related to the different domains covered by our contribution. i.e., the fundamental concepts and principles, on the one hand, of the model-driven engineering (MDE) and on the other hand, of the model based systems engineering (MBSE). It introduces also the trend of domain specific modeling (DSM) and domain specific modeling languages (DSML), discussing individually the underlying components of a DSML (i.e., DSML abstract syntax, concrete syntax, static semantics and dynamic semantics).

The Chapter III introduces the first component of the proposed method, namely, **the core concepts of our method** allowing modeling, verification and validation. It consists of a typology of properties for modeling and a formalized lifecycle for property management. The lifecycle provides stakeholders with guidelines, i.e., several phases and sub-phases, each one characterized by various constraints, expectations and rules to be considered and modeled as properties for the design and V&V of DSMLs and models.

The Chapter IV focuses on the design of executable DSMLs that allow simulation (i.e., model execution). It evaluates a well-known state of the art approach for executable DSMLs coming from the field of MDE, highlighting issues and possible improvements for its effective adaptation in the field of MBSE. Based on the feedback, Chapter IV introduces **the languages of our method**. These languages formalize the means to design and manage the concepts of our method previously introduced in Chapter III, but also they support the activities for modeling, verification and validation.

The Chapter V presents the **operating demarche of our method** for the design and V&V of models, including a mechanism for simulation based on model execution, and mechanisms for formal properties proof. The demarche put in use the languages previously introduced in Chapter IV, along with several original rules that we define.

CHAPTER II

STATE OF THE ART

This chapter presents the state of the art related to the different domains covered by our contribution. Section 0 introduces model-driven engineering fundamental concepts and principles: model, metamodel and model transformations. Section 0 introduces the model-based systems engineering presenting a general typology of models, the viewpoint representations of systems and the process of verification and validation based on models. Section 0 introduces domain specific modeling (DSM) and domain specific modeling languages (DSML). The components of a DSML, i.e., its abstract syntax, concrete syntax, static semantics and dynamic semantics, are individually discussed. Finally, Section 2.4 synthetizes the previously discussed literature and conclude, positioning the contribution of this thesis.

2.1 Model-Driven Engineering

2.1.1 Introduction

In parallel to Systems Engineering, within the field of Software Engineering, since the early 2000s, the increasing complexity of software caused an important paradigm shift. The goal of this attempt is to move from the object-oriented software engineering with a basic principle *"Everything is an object"* towards the model-driven engineering (MDE) (Schmidt 2006) with a basic principle *"Everything is a model"* where models and model-elements are first class citizens (Greenfield & Short 2003; Bézivin 2005).

The MDE aims "to increase productivity and reduce time-to-market by enabling development at a higher level of abstraction and by using concepts closer to the problem domain at hand, rather than the ones offered by programming languages" (Sendall & Kozaczynski 2003). On the one hand, MDE aims to improve software development processes by increasing the abstraction level through models at different stages of software systems development and by early verification and validation (V&V) activities based on models. On the other hand it aims to increase the level of automation, from abstraction to program deployment, using code generation techniques, eventually transforming models into code.

2.1.2 Model and Metamodel

The move towards the model technology introduced new fundamental concepts and relations, among which the main ones are "model" and "metamodel".

The term “Model” comes from the Latin word “Modulus” meaning measure, rule, pattern or an example to be followed.

- A **model** is “a representation of an original system, i.e., a subject that might exist or not, containing at least one, but not all subject properties” (Stachowiak Herbert 1973).

In general, models are used by experts to understand and reason about a system under study (i.e., a system of interests - SoI), to communicate and argue with other actors about this SoI and finally as a support that increases experts’ confidence during decision making processes.

The core concepts of the “object-oriented” paradigm are classes and instances and its core relationships are *“inheritsFrom”* between classes, and *“instanceOf”* between instances and classes. The Object technology main benefits are simplicity, generality and power of integration as a result to its two core principles, namely, an object is an instance of a class and a class inherits from another class (Bézivin 2005).

Very differently, what is important for the MDE is that a particular viewpoint (an aspect) of a system is “*representedBy*” a model that is written in the language of its metamodel, i.e., the model “*conformsTo*” the metamodel (Bézivin 2005).

- A **metamodel** is a model that defines a language to specify conforming models, i.e., a modeling language (OMG 2015a).
- A **meta-metamodel** defines a language to specify conforming metamodels, i.e., a metamodeling language. A well-known example is MOF (OMG 2015a).

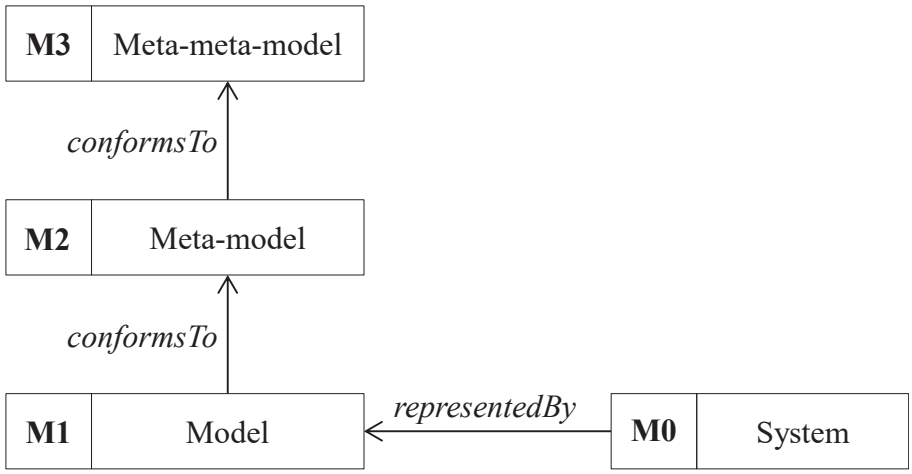


Figure 2. The OMG’s metamodeling layers.

Figure 2 illustrates the metamodeling stack and the relations between metamodeling layers, initially proposed by the Object Management Group (OMG). An example that illustrates the OMG's metamodeling stack is detailed in Figure 3, showing the modeling of the hardware aspect of a personal computer.

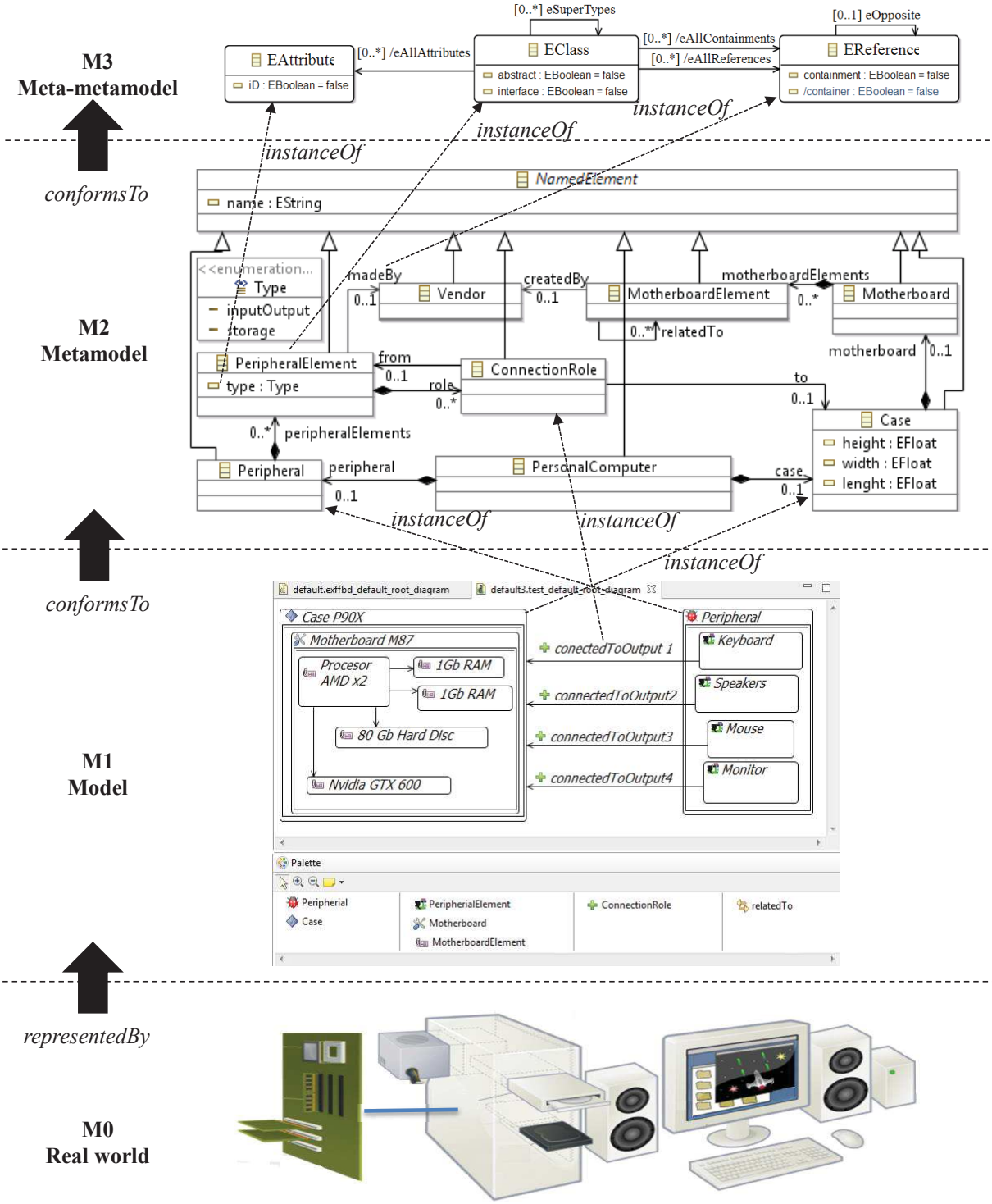


Figure 3. An example to illustrate the OMG's metamodeling stack.

The M0 layer represents “real world” systems to be designed, for instance, computer hardware. A viewpoint of this system is represented by a model that is located in the M1 layer. The model *conformsTo* a metamodel. The metamodel is placed in the M2 layer. For instance, the metamodel of Figure 3 shows the classes and relationships that model the core domain concepts and relationships of the hardware aspect of a personal computer. Note that, a modeling language, in addition to the metamodel, is composed of other parts that are not here-discussed (for more details, see Section 0). The metamodel itself *conformsTo* a meta-metamodel. The mata-metamodel is located in the highest M3 layer. For instance, the M3 layer of Figure 3 shows a part of the metamodeling language EMOF/Ecore (Steinberg et al. 2008) composed of *EClass*, *EAttribute* and *EReference*.

2.1.3 Model Transformation

One of the challenge of MDE is in transforming higher-level models to so-called platform-specific models that can be used to generate code (Sendall & Kozaczynski 2003). So, the second most important concept of the MDE is the model transformation.

Nowadays, more than thirty transformation approaches exist in the literature. A classification is proposed in (Kahani & R. Cordy 2015) distinguishing two major categories, depending on the transformation result: *(1) model-to-code transformations* and *(2) model-to-model transformations*. A third type of transformation approaches known as code-to-model transformations are not here-considered.

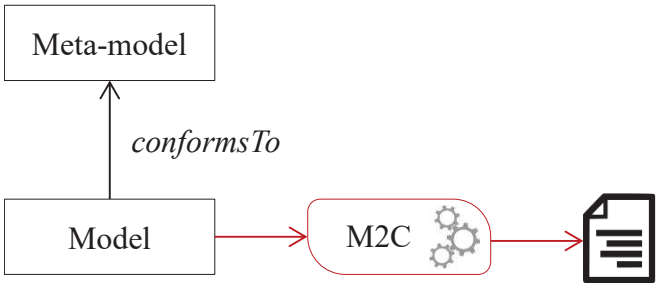


Figure 4. Model-to-Code transformation (M2C).

Model-to-code (M2C) transformations (see Figure 4) also known as “code or document generations” are used to generate code or documents from a source model. There are two types of M2C transformations: *a) visitor-based* and *b) template-based* transformations. The *visitor-based* transformation consists in providing a mechanism that visits (parses) an internal representation of a model and produces code into a text stream. The *template-based* transformation is based on *templates that consist of the*

target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion.

Model-to-model (M2M) transformations (see Figure 5) are used to transform a source model into a target model. Both, source and target may be instance of the same metamodel, denoted *"endogenous transformations"* or different metamodel, denoted *"exogenous transformations"*. According to (Czarnecki & Helsen 2003), there are 5 types of M2M transformations: *a) direct-manipulation*, *b) relational*, *c) graph-transformation-based*, *d) structure-driven* and *e) hybrid approaches*. The *direct-manipulation* transformations offer an internal model representation and an API to manipulate it, but consist mostly in implementing transformation rules and scheduling from scratch. The *relational* transformations are declarative rules based on mathematical relations that consist to specify the source and target element type of a relation using constraints. The *graph-based* transformations are grounded on the graph grammars, discussed in the next. The *structure-driven* transformations ease the work of users that are only concerned with the design of transformation rules, by providing scheduling and application strategy. The *hybrid* approach combines different approaches from the previous categories.

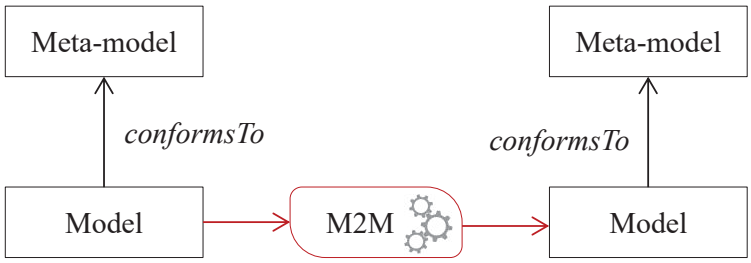


Figure 5. Model-to-Model transformation (M2M).

Within the context of MDE, model transformations are considered as an integral part of the SoI. Following the based MDE "everything a model" principle, model transformations are also considered as models, denoted "transformation models". Similarly to "classical" models, transformation models *conform to* transformation metamodels (see Figure 6). The transformation models *conform to* a metamodeling language (e.g., MOF) (Bézivin et al. 2006).

Transformation models can also be modified and extended via transformations denoted Higher-Order Transformations (HOT). A HOT is a model transformation taking as

input a model transformation and producing as output a model transformation (Bézivin et al. 2006).

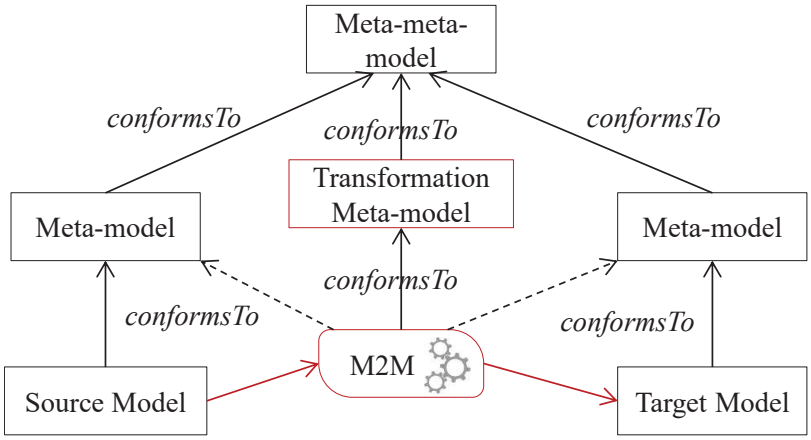


Figure 6. Model transformation process.

2.1.4 Synthesis

MDE promote the use of models during the development processes at different levels, from higher problem and functionalities related level, to lower platform and implementation related level to automatize the development and the V&V. Models are used by stakeholders to understand and reason about the modeled system, to communicate and argue with other actors about this system and finally to support them and increase their confidence during decision making processes. It is thus very important prior to any decision to verify created models ensuring that they are well-formed and correctly build.

Models are written in the language of their (*conforms to*) metamodel and are used *to represent* a particular viewpoint (an aspect) of a system. So, the two core relationships of the MDE are: the conformity relation (i.e., a model conforms to its metamodel) and the represented by relation (i.e., an aspect of a system is represented by a model).

2.2 Model-Based Systems Engineering

2.2.1 Introduction

As introduced in Chapter I, within the field of organizational and engineering sciences, the Model-Based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation

activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases (INCOSE 2007).

MBSE promotes concepts, methods and techniques for creating and managing various systems models of different viewpoints of a SoI for the purpose of stakeholders, and for reaching and improving the quality of models helping then stakeholders all along design processes to make and justify decisions with a higher level of confidence, reducing as much as possible the uncertainty. Indeed, these decisions impact downstream phases of SoI development until its realization and deployment in terms of functioning, safety, security, induced costs and so on.

2.2.2 MBSE viewpoint representations

Following the general system theory and principles (Le Moigne 1999), the modeling of a SoI is carried out through three interdependent viewpoints: *(1) functional, (2) structural* and *(3) behavioral*.

- ***Functional***: describes what the system must do in its environment. It is used to respond to the following questions: "What is the SoI for? What is the purpose of the SoI? The SoI missions and objectives?"
- ***Structural***: represents the SoI structure. It is used to respond to the question "What is this SoI made of? The used resources? How is it structured to fulfill its mission (in its moving environment)?"
- ***Behavioral***: describes the way SoI have to, or must, behaves. It responds to the following questions: "What does the dynamic of the SoI operates so that it evolves in time, for instance from one state to another, the conditions to be satisfied so that SoI reaches a certain state, etc.".

The above general system theory has been adapted and standardized by ISO (ISO/IEC 2008) considering the SE principles and the iterative nature of the SE processes (INCOSE 2010), into six viewpoints *(1) system, (2) requirements (3) functional,(4) logical, (5) physical* and *(6) organic*:

- ***System viewpoint*** represents the SoI main characteristics and its frontier with its operational environment. Among other characteristics, the system view define the SoI mission, its purpose and objectives, its functioning mode and various operational scenarios that show how does the SoI evolve when confronted to various situations. The system view defines also the SoI various operational

contexts. Each context specifies the SoI's expected services that correspond to its mission, and services that are requested by the SoI to fulfill this mission. Requested services are provided by interfaced systems from SoI's environment. So, global SoI's input and output flows and physical links are also defined in this view, specifying the SoI's frontier.

- ***Requirements viewpoint*** defines all stakeholders and SoI requirements. It allows first understanding stakeholder's expectations, constraints and roles, and second guiding design process.
- ***Functional viewpoint*** defines the SoI's functional architecture, specifying SoI's functions and their sub-functions. A function defines a transformation of input flows into output flows performed by a SoI to achieve its mission (INCOSE 2016a). It shows how do functions are dynamically arranged, their execution sequencing and how conditions for control or data-flow are taken into consideration to satisfy the requirements baseline. By the principle of iterative design, such functional architecture may evolve considering next architectures.
- ***Logical viewpoint*** defines different solutions of SoI's logical architecture, i.e., variations of arrangements of functions and their sub-functions highlighted in functional architecture and their interfaces (internal and external) (ISO/IEC 2008). In other words, a logical view shows how do the SoI's functions can be logically associated for instance by regrouping their input and output flows to optimize their future allocation to physical components, or by considering requested modularity.
- ***Physical viewpoint*** allows representing various solutions of physical architecture i.e. arrangement of physical elements (SoI' elements and physical interfaces) which provides a possible design solution for a product, service, or enterprise, and is intended to satisfy one of the proposed logical architectures and respecting system requirements (ISO/IEC 2008).
- ***Organic viewpoint*** defines the organic architecture that is similar and thus often confused with the physical architecture. The organic architecture highlights technical and configured components representing the final product put in operational context.

Let us note that some of the above discussed viewpoints implicitly define the behavior of a SoI. Namely, the Functional and Logical viewpoints characterize both a static

description of SoI's functions and a dynamic description of functions execution (e.g., sequences, synchronization, parallelism and flows control of a SoI). The Physical and Organic viewpoints are both static representations of how SoI elements are selected and interconnected, and dynamic representations of how each component evolves, supporting and executing SoI's functions.

Similarly, the System viewpoint highlights various operational contexts in which a SoI dynamically interact with other systems, highlighting also its behavior when confronted to the environment (e.g., operational scenarios) and its configurations and functioning mode sequences.

2.2.3 MBSE modeling languages

Within the context of MBSE, there are currently various modeling languages that cover one or several of the MBSE viewpoint representations discussed above.

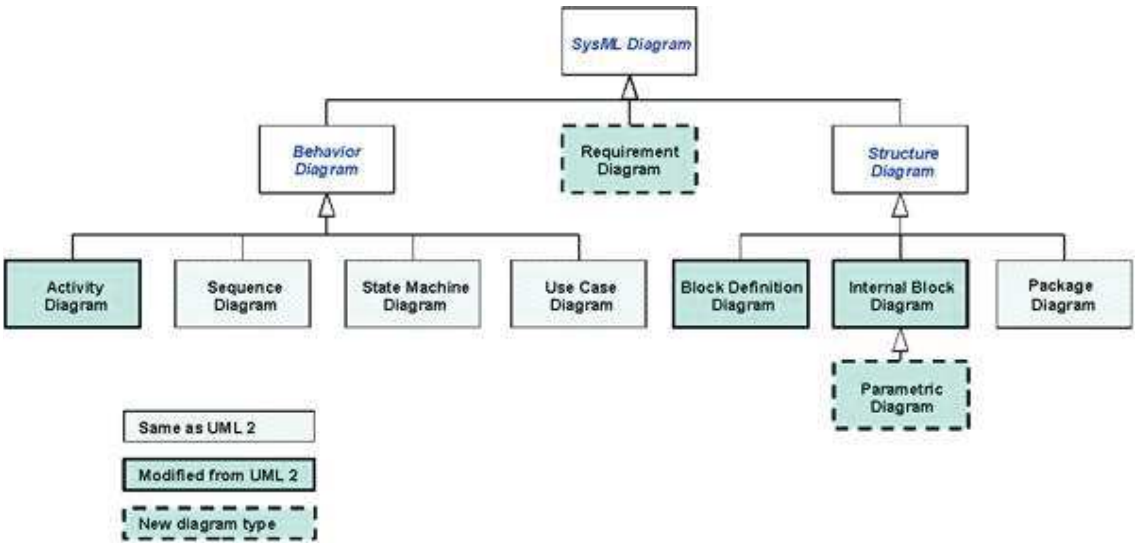


Figure 7. SysML diagram types (Friedenthal et al. 2014).

One of the most commonly used and well-known is the Systems Modeling Language (SysML) (OMG 2015b). SysML integrated several modeling languages, denoted as diagrams, for modeling the physical and behavioral architectures of a system as well as the systems requirements (see Figure 7). For intstance, the Block Definition Diagram is used to model the structure of a system through physical blocks and interfaces. The Activity or State Machine Diagrams are used to describe the behavior of a system. The activity diagram models the flow of data and control between activities, whereas the state machine diagram describes the states of a system and transitions between states

that are fired in response to events. Unfortunately, the initial SysML neglects the functional architecture, even though some research works propose to modify the activity diagram to support a flow of matter of energy (Friedenthal et al. 2014). For more details on SysML diagrams see (OMG 2015b).

Alternatively to SysML, other well-known MBSE modeling languages are:

- The *Enhanced Functional Flow Block Diagram* (eFFBD) (INCOSE 2010) is a functional-modeling language for the design of functional and behavioral viewpoints of complex, distributed, hierarchical, concurrent and communicating systems. The eFFBD is not targeted for modeling the physical viewpoints.
- In contrary, the *Physical Block Diagram* (PBD) (Long 2007) is a block-modeling language that provides systems engineers with a block-and-line diagram representing the physical components of a system or system segment and links that connect components through interfaces, offering a detailed view of an architectural composition.
- The *FCCS* (French acronym of *GRAphe Fonctionnel de Commande Etape-Transition* – GRAFCET) (IEC 1992) is a behavioral language for describing sequential automatisms such as Control Part of Manufacturing Systems. Especially, it allows parallelism description and it is a programming language available on many Programmable Logical Controllers of the market. The FCCS is not adapted for modeling the physical viewpoint of a system.
- The *Petri Net* (*place/transition net*) (Murata 1989) is a behavioral language for describing distributed systems. Petri nets have formal definition of their execution semantics, with a well-developed mathematical theory for process analysis. They are today widely used in various areas such as Systems engineering, Concurrent programming or Discrete process control, particularly for verification and validation purposes and are the subject of various works, e.g., for the verification of eFFBD models as proposed in (Seidner 2009).
- The *Interpreted Sequential Machine* (ISM) (Vandermeulen 1996) is a formal language based on discrete-event hypothesis for modeling and verifying the behavior of systems and their interactions with the environment. The ISM is not adapted for modeling the functional and physical viewpoints.
- The *continuous models* (CM) (Lee 2003) specified by a set of mathematical equation (i.e., continuous or differential equations) that define the behavior of

systems and their interactions with the environment. Continuous models allow modeling the behavior of a system based on continuous hypothesis. However, they are not adequate for modeling the physical viewpoints.

- The *Operational Mode Analysis Grid* (OMAG) (Chapurlat & Daclin 2013) is an approach that guides designers in exploring and reasoning, checking and then arguing the consistency of the operating modes of a system. The goal is to help designers to build system's functional architecture by linking operating modes, allowed configurations and operational scenarios. The OMAG is not adapted for modeling the physical viewpoint of a system.

So, the above introduced languages are used to model different SoI architectures. As previously discussed, some of the SoI architectures are suited for the structural description of a SoI (e.g., the components that build up the system, the interfaces of the system, the system flows, etc.) others are suited for the behavioral description (e.g., the functions of the system, the interactions of the system with the environment, etc.).

2.2.4 MBSE verification and validation activities

Designed models are finally used by stakeholders during decision making processes to understand a SoI and argue various architectural choices. These decisions impact on the whole SoI, i.e., its functioning, induced cost, safety, security and of course SoI engineering processes. It is thus very important, prior to any decision, to assure that used models are complete, correct and relevant. According to (Chapurlat 2008), model's *completeness*, *correctness* and *relevance* are defined as follows:

Model completeness: a model is complete if it is self-sufficient and contains all necessary information for stakeholder's objectives, i.e., to demonstrate or deny information that a stakeholder wants to highlight and analyze concerning the SoI. However, achieving model completeness (i.e., a model that covers all characteristics of a given reality that is, in our case, a SoI) is impossible by definition. Namely, models are an abstraction of a subject and should only contain characteristics that are relevant for a given study (see the definition of a model in Section 2.1.2). Therefore, modeling languages and covered viewpoint representations must act as a filter, excluding concepts that are non-relevant for the conducted study, including only the relevant ones. In such a way, the unnecessary information should be filtered away, simplifying the representation and easing the understanding by presenting to stakeholders only relevant

informations for a given study. The model completeness can be analyzed by considering the boundaries of the conducted study, information that is possessed by domain experts.

Model correctness: the correctness of a model is expressed through model's *(1) consistency*, *(2) conformity* to a metamodel, the *(3) respect to well-formedness rule*s and the *(4) correct concrete (graphical or textual) representation*.

- A model is *consistent* if it does not contain any ambiguous or contradictory information, i.e., information that based on this model is true and false at the same time, leading to non-decidability. The consistency of a model is above all partially assured by the conformity to a metamodel that restricts model designers to concepts and relationships introduced in the used modeling language. In addition, models must be checked taking into account the modeling language well-formedness rules (discussed below). A model should also become consistent with the other viewpoints models of the same SoI. This means that there is not a contradiction between different viewpoint models and that the information that is correct considering one model should stay correct considering the other models of the same SoI.
- A model *conforms to* a metamodel if it respects the metamodeling rules imposed by the DSML (i.e., by its abstract syntax). For more details on the conformity relationship, see Section 2.1.2. An example of this relationship is illustrated in Figure 2.
- A model must *respect well-formedness rules* that are defined by the semantics of the used modeling language. For more details on semantics and well-formedness rules, see Section 2.3.2.
- A model is *correctly represented*, graphically of textually, if the representation of this model respects the rules imposed by a concrete syntax. For more details on concrete syntaxes, see Section 2.3.2.

Model relevance*:* determines how accurately and correctly a model represents a viewpoint of a SoI, just as imagined by stakeholders. For this purpose, models must first be *complete* and *correct*, and moreover, models must respect rules that represent the domain knowledge and needs of different stakeholders, i.e., the functional and non-functional requirements.

Model completeness, correctness and relevance are managed by implementing model verification and model validation (model V&V) activities:

- *Model verification*: it aims to demonstrate that *a model is correctly build, well-formed and correctly represented*, taking into account the modeling rules defined into a metamodel, the well-formedness rules defined through the modeling language semantics and the representational rules defined as a concrete syntax.
- *Model validation*: it aims to demonstrate that *a model is the right one and is trustworthy*, giving an accurate representation of SoI in a viewpoint, considering this representation as sufficient respecting the stakeholders and systems requirements.

In the MBSE context, model V&V activities should consider all viewpoint representations of a SoI, taken first separately, but also pieced together providing a more complete and suitable representation. The goal is then to demonstrate the mutual coherence throughout all viewpoint representations of a SoI, as well as their adequacy and global fidelity to the SoI to support the designers' objectives with an assured level of confidence (Blazo Nastov et al. 2016b).

In the past 20 years, within the field of SE, a lot of approaches and frameworks have been developed for verification and validation (V&V) of safety and critical systems. The MBSE focuses particularly on early V&V based on models that take place during the system design processes. According to (Chapurlat 2008), MBSE approaches for V&V are based on one of the four V&V strategies: *(1) Model expertise, (2) Guided modelling, (3) Simulation and (4) Formal proof*.

Model expertise: this strategy involves domain V&V specialists that have experience in the evaluation and the appraisal of models relative to their domain of expertise. V&V experts might rely on other techniques such as simulation or formal proof. This is an efficient method for determining the quality of a given model but is relatively expensive, particularly in a multidisciplinary context requiring multiple V&V specialist with the required domain expertise.

Guided modeling: this strategy consists in guiding stakeholders based on patterns, boilerplates or feedbacks. We distinguish then: *(1) pattern-based* approaches, *(2) boilerplate-based* approaches and *(3) feedback-based* approaches.

- The *pattern-based* approaches promote the use of modeling patterns, hints and frameworks for guiding experts during a design process. The goal of pattern-

based approaches is to eliminate structural design errors by proposing possible solutions to a problem based on modeling patterns, considered to be good practices. For instance, an approach for pattern implementation for systems engineering, based on a functional architectural patterns, is proposed in (Pfister et al. 2012). This approach is formalized as a metamodel and is used for the management, application and cataloging of patterns specific to the field of systems engineering. A model-driven framework for guided design space exploration is proposed in (Hegedüs et al. 2015). This framework aims at searching, based on hints (i.e., selection criteria), through various models representing different design candidates to support activities like configuration design of critical systems or automated maintenance of IT systems.

- The *boilerplate-based* approaches introduce template models that contain crucial, already validated information of a given domain. The goal of boilerplate-based approaches is to ease the work of designers by providing a solid starting point basis with pre-verified information. For instance the European CESAR project (CESAR 2012) proposes boilerplates-based requirement specification language for the design of requirement models. Another example is proposed in (Stålhane et al. 2011) where an approach for system safety analysis based on requirements is proposed. Similarly to the CESAR project, in this approach, the safety requirements are designed on top of boilerplate models, specifically tailored for safety analyses. The EARS (Mavin et al. 2009) approach (Easy Approach to Requirements Syntax) introduces boilerplates for state-transition-based behavior requirements, limiting non-desired system behavior as early as possible.
- The *feedback-based* approaches promote the reuse of models and examples that are considered to be, at least, verified and validated, or, at best, standardized in a given domain. The goal of feedback-based approaches is to share the domain experience (problems, causes, and possible solutions) with designers of the same domain that attempt to solve similar problems. Whether it is intended to solve a problem or to abstract a general solution, a mechanism to process the examples and to obtain information or knowledge from them, is needed. The choice of this mechanism depends on the problem's nature, on how general the solution is expected to be and also on how much information about the solution is known

beforehand. For instance, in (Faunes Carvallo 2013), it is proposed to improve the automation in the model-driven engineering, based on examples.

Simulation: this strategy consists in observing the simulated behavior of a SoI. The simulation has numerous benefits. It is generally cheaper, safer, faster and more ethical than conducting experiments on real-world systems. Simulations can become more realistic if required by increasing the number parameters taken into account and the model hypothesis (discrete-events, continuous or hybrid). There are currently many tools for simulation. Among the most effective and well-known are: Ptolemy, Simulink and Modelica. Ptolemy (Lee 2003) is a modeling and simulation environment for the design of concurrent, real-time and embedded systems, based on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. Simulink (Mathworks 2014) is a programming environment for modeling, simulating and analyzing multi-domain dynamic systems, offering integration with the rest of the MATLAB environment. Simulink is widely used in automatic control and digital signal processing for multi-domain simulation and Model-Based Design. Modelica (Hilding Elmqvist 1997) is an object-oriented, declarative, multi-domain modeling language for modeling and simulation of complex systems. The Modelica Association develops a free Modelica language "OpenModelica" and a free Modelica Standard Library that contains about 1360 generic model components and 1280 functions in various domains.

Formal proof: consists in the use of formal methods, languages and tools. Formal methods are mathematically based methods for the specification, development and verification of systems. They leverage the use of formal languages that have solid mathematical semantics. As a result, formal system specifications are unambiguous and can be used to perform mathematical analysis, contributing to the reliability and robustness of a design. Formal methods are based on two different approaches for formal verification: *(1) model-checking* or *(2) theorem proving*.

- *Model-checking* is an approach to verify (to check) if a given specification of a system (in the context of this work a system specification defines one or several of the SoI viewpoints introduced in Section 2.2.2) respects some properties (Bérard et al. 2013). It consists first in specifying the system through a formal specification and then the requirements to be verified as formal properties. Second, specified properties are verified based on a systematically exhaustive

exploration of the system specification, i.e., by exploring all possible states of this specification. Well-known tool-supported solutions that allow model-checking are SPIN (Holzmann 1997) and UPPAAL (Larsen et al. 1997).

- *Theorem proving i*s a technique for formal verification that consists in generating a collection of mathematical proof obligations from a system specification. These obligations imply conformance of the system to its specification. They can be formally proven by using a theorem prover. Well-known tool-supported approaches that allow formal proof are: the B-method (Abrial 2005), VDM (Alagar & Periyasamy 2011), Coq (Bertot 2006), Isabelle (Nipkow et al. 2002), etc.

For more details on the state of the art of formal methods see the following survey paper (Woodcock et al. 2009).

2.2.5 Synthesis

During the early system development phase, the MBSE promotes concepts, methods and techniques that allow stakeholders to create and use models. These models support stakeholders in understanding a SoI and in communicating and arguing with other stakeholders about this SoI, before making any decision.

Nowadays, there are two major issues in the context of MBSE, the first is related to the design of models that can effectively cover and represent different viewpoints of a SoI, whereas the second is related to the Verification and Validation (V&V) of these models. For the purpose of modeling, the general system theory promotes three viewpoints: *(1) functional, (2) structural* and *(3) behavioral*. This theory is furthermore adapted within the context of MBSE, promoting six viewpoints: (*1) system viewpoint, (2) requirements viewpoint, (3) functional viewpoint, (4) logical viewpoint, (5) physical viewpoint* and *(6) organic viewpoint*. Modeling languages (e.g., SysML, eFFBD, PBD, etc.) are then used to cover each of these viewpoints. However, prior decision-making processes, stakeholders must, on the one hand, verify models, i.e., to demonstrate that they are *correctly build, well-formed and correctly represented,* and on the other hand, to validate model, i.e., to demonstrate that they are the right ones and are *trustworthy*, representing sufficiently accurately a viewpoint of a SoI, considering also the domain knowledge of stakeholders. V&V activities must take into account each of the SoI models, first separately, and after pieced together with the other models of the same SoI,

providing a more complete and suitable representation of it. Model V&V activities are based on the following strategies: *(1) Model expertise, (2) Guided modelling, (3) Simulation and (4) Formal methods*.

So, (1) the design of viewpoint models stress the need for *modeling languages* that are particularly tailored and adapted to a given viewpoints, and (2) achieving a sufficient level of model quality through V&V analyses stresses up the need to adapt and suite the used modeling languages for V&V along with various techniques and tools.

2.3 Domain Specific Modeling Languages

2.3.1 Introduction

As mentioned before, models play a dominant role within the problematic of this work. Models are created by using a modeling language and conform to a metamodel that is embedded in this modeling language (see Section 2.1.2). There are two main paradigms for modeling: 1) *General-Purpose Modeling (GPM)* and 2) *Domain-Specific Modeling (DSM)*. GPM promotes the use of a *General Purpose Modeling Language (GPML)* for the modeling of different viewpoints of a SoI. A well-known example is the OMG's Unified Modeling Language (UML). DSM promotes the use of a *Domain Specific Modeling Language (DSML)* particularly tailored for a given problem, for the modeling of one viewpoint of a SoI that is used to solve a given problem. The main difference between GPML and DSML is that the prior is used to model any SoI viewpoint for any problem, while the latter is used to model one particular SoI viewpoint for one well-defined problem. As a consequence, on the one hand, GPML provide generic concepts that are far from the end-user domain ontology. On the other hand, the genericity of GPML might overwhelm the end-use with many different ways to model an artefact. In contrary, a DSML integrates the end-used domain ontology, easing the understanding and use. Moreover, domain models are represented with an end-user friendly graphical or textual concrete syntax (discussed below) and provide constraints dedicated to a considered domain problem that can be used to verify created models (discussed below).

Considering the MBSE issues discussed in the previous Section, the focus here is on designing and managing DSMLs for multi-viewpoint modeling (discussed in Section 2.3.2), and on extending DSML along with different techniques and tools for the purpose of model Verification and Validation (discussed in Section 2.3.3).

2.3.2 DSML for multi-viewpoint modeling

The first issue related to the design of models that can effectively cover and represent different aspects of a SoI, stresses the design, use and management of DSMLs. Generally, the design of a DSML consists in creating *1) an abstract syntax* and *2) a concrete syntax*.

Abstract syntax: the original meaning of the term abstract syntax comes from natural language, where it means the hidden, underlying, unifying structure of a number of sentences (Chomsky 1965). Generally, the abstract syntax is hidden, presented as in-memory form that obtains a concrete form when shown on a screen for the purpose of language users (Kleppe 2007). Its concrete form may vary, depending on the associated concrete syntax (detailed below). In the field of MDE, an abstract syntax is given by a *metamodel* (see Section 2.1.2) representing, through a graph of classes, the concepts of a domain and their relationships. Metamodels are created by using metamodeling languages such as the standard MOF (OMG 2015a). MOF is tool-supported for instance as Ecore in the *Eclipse Modeling Framework* (EMF) (Steinberg et al. 2008).

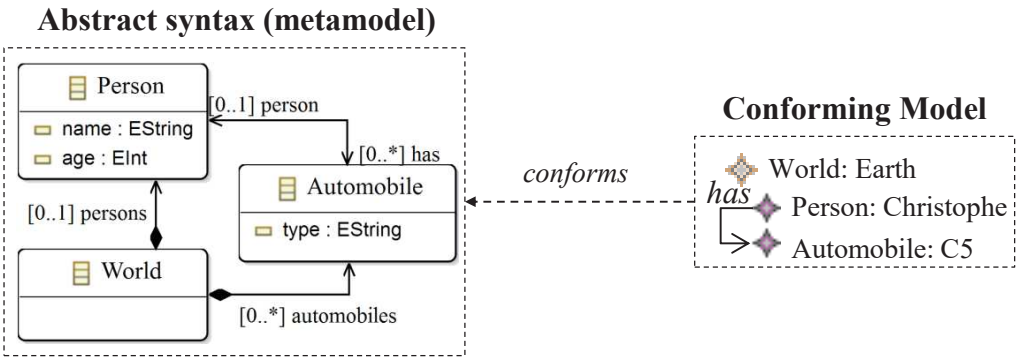


Figure 8. An example of an abstract syntax (metamodel) and a conforming model.

Figure 8 shows an example of an abstract syntax in the form of a metamodel created by the metamodeling language MOF and a conforming model. The metamodel is composed of three classes: World, Person and Automobile. The class World is composed of persons and automobiles. Each person might have one or several automobiles and each automobile might be possessed by one person at most. The conforming model shows the world "Earth" with the person Christophe and the automobile C4 (possesses by Christophe) in it.

Concrete syntax: a *Concrete syntax* defines the textual or graphical representation of a model. The graphical representation of metamodels is indeed well-known and similar to

the one of UML class diagram (see M2 layer of Figure 3). Models (instances of a metamodel) however, have also an abstract syntax (i.e., AST) and a concrete syntax so they can be understood by engineers (see M1 layer of Figure 3). The information that defines the representation of models is their concrete syntax. This information defines how to represent, not the classes and their relationships, but the instances of classes and the instances of relationships. Depending on the nature of a concrete syntax that might be graphical of textual, editors support either textual or graphical notations. For instance, Figure 9 shows a model composed of its AST and two representations, a graphical and a textual.

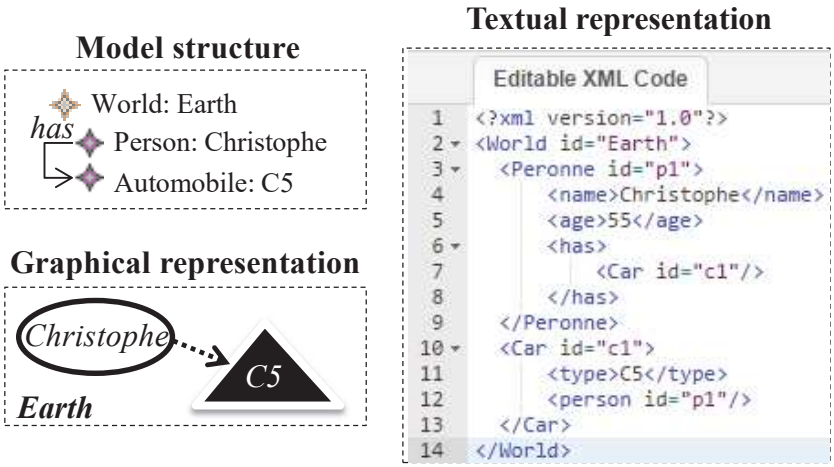


Figure 9. An example of a model with its structure, a graphical representation and a textual representation.

There are currently several tool-supported solutions for the design of graphical and textual concrete syntaxes, such as Diagraph (Pfister et al. 2014) and Sirius (Juliot & Benois 2010) for graphical concrete syntaxes or xText (Bettini 2013) for textual concrete syntaxes.

Composability: following the theory and principles of multi-viewpoint modeling, various interconnected models are designed for a given SoI as suggested in Section 1.2.1 and Section 2.2.2, which when put together, form a "composite model", covering a more expressive, realistic and complete representation of a SoI. The design of such interconnected models is possible only if the used DSMLs are syntactically interconnected. This consists in defining the dependencies between the abstract syntaxes of each DSML, but also between their concrete syntaxes. Examples of such syntactical dependencies are shown in Section 3.3.1 and are illustrated in Figure 30 and Figure 31.

Examples of syntactical interconnection between models is shown in Section 3.3.2 and illustrated in Figure 34 and Figure 35.

There are currently different methods / approaches for the design of DSML. Among the more relevant for the purpose of this work are: Kermeta (Fleurey 2006), Eclipse Modeling Framework – EMF (Steinberg et al. 2008), GEMOC studio (Combemale 2016), Sirius (Juliot & Benois 2010) and Diagraph (Pfister et al. 2014). Table 1 compares these methods / approaches based on the following criteria: 1) does the given method / approach provides the means for the design of abstract syntaxes; 2) does the given method / approach provides the means for the design of concrete syntaxes; 3) does the given method / approach provides the means for composing abstract syntaxes of different DSML; 4) does the given method / approach provides the means for composing concrete syntaxes of different DSML; 5) is the given method / approach tool-equipped.

Table 1. Comparison of several approaches for the design of DSML.

Methods / Approaches	***Design of abstarct syntaxes***	***Design of concrete syntaxes***	***Composability of abstarct syntaxes***	***Composability of concrete syntaxes***	***Is tool-equipped***
Kermeta	*Yes*	*No*	*Yes*	*No*	*Yes*
EMF	*Yes*	*No*	*Yes*	*No*	*Yes*
GEMOC studio	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Sirius	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Diagraph	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*

So, Kermeta and EMF focus on the design and composition of abstract syntaxes and semantics (e.g., executable semantics), neglecting the design and composition of graphical and textual concrete syntaxes. For this purpose, Sirius and Diagraph are layers on top of the EMF that focus primarily on the design and composition of graphical concrete syntaxes. Finally, GEMOC studio relies on EMF and Sirius for the design and composition of abstract syntaxes and graphical concrete syntaxes.

2.3.3 DSML for model Verification and Validation

The second issue related to model V&V, stresses the need for extending or adapting designed DSMLs for the purposes of simulation (i.e., model execution) and formal proof (i.e., verification of formal properties). To this end, along with its syntax (abstract and concrete), a DSML must include semantics. According to (Combemale et al. 2009), the DSML semantics can be divided into: *static semantics,* representing concept meaning (abstract and concrete syntaxes) and behavior independent structural constraints (pre and post conditions, invariants, etc.), and *dynamic semantics*, dealing with the way models behave.

Static semantics: the whole domain knowledge cannot be captured by an abstract and a concrete syntax. For instance, considering the abstract syntax shown in Figure 8, the following information "only major persons (age>18) can have an automobile" cannot be defined with a metamodeling language. For this purpose, static semantics define such restrictions and additional information for the syntax or the behavior (the dynamic semantics) here-referred as "*static semantics properties*" or simply "*properties*". Properties are specified by using a "*property modeling language"* such as OCL (OMG 2014), TOCL (Ziemann & Gogolla 2003), LTL (Pnueli 1977), etc. The used property modeling language determines the type of properties that can be designed (e.g., temporal or a-temporal). In addition, an adequate model-checking tool is needed to check if the designed models respect the specified properties. For instance, the OCL interpreter can be used to verify the model illustrated in Figure 9 respects the OCL property illustrated in Figure 10.

context *Person* **inv:**
self.has->size() > 0 *implies* self.age > 18

Figure 10. A static semantics property specified as an OCL constraint.

Considering composability: following the theory and principles of multi-viewpoint modeling, properties should also be specified and verified based on composite models as suggested in Section 1.2.1 and Section 2.2.2. This is only possible if the syntaxes of considered DSML are already interconnected. Examples of such properties are illustrated in Section 3.3.1.

The methods / approaches discussed above (i.e., Kermeta, EMF, GEMOC studio, Sirius and Diagraph) integrate also one or several property modeling languages for the

specification and verification of properties. Table 2 compares these methods based on the following criteria: 1) does the given method / approach provides the means for the design of static semantics; 2) who is/are the proposed property modeling language(s) for the specification of properties; 3) is the verification of properties achieved directly on models or by transformation to other third party approaches; 4) is composability as described above possible; 5) is the given method / approach tool-equipped.

Table 2. Comparison of several approaches for the design of DSML based on their ability to allow property specification and verification.

Methods / Approaches	***Design of static semantics***	***Property modeling language***	***Direct verification***	***Composability***	***Is tool-equipped***
Kermeta	*Yes*	*OCL*	*Yes*	*Yes*	*Yes*
EMF	*Yes*	*OCL*	*Yes*	*Yes*	*Yes*
GEMOC studio	*Yes*	*OCL*	*Yes*	*Yes*	*Yes*
Sirius	*Yes*	*OCL*	*Yes*	*Yes*	*Yes*
Diagraph	*Yes*	*OCL*	*Yes*	*Yes*	*Yes*
xDSML design pattern	*Yes*	*OCL / LTL*	*Yes for OCL* *No for LTL*	*Yes for OCL* *No for LTL*	*Yes for OCL* *No for LTL*

Before discussing Table 2, we compare the following property modeling languages based on the types of properties they allow specifying: OCL (OMG 2014), TOCL (Ziemann & Gogolla 2003), LTL (Pnueli 1977). We consider four types of properties: those that concern the structure of a DSML (i.e., the abstract syntax) denoted *Structural properties*; those that concern the behavior of a DSML (i.e., the dynamic semantics) denoted *Behavioral properties*; and those that include or not a temporal dimension denoted respectively *Temporal* or *A-temporal properties* (e.g., temporal properties are important for simulation and should be verified each step of the simulation or at specific time step). The results of the comparison are shown in Table 3.

Table 3. Comparison of property modeling languages.

Properties	***OCL***	***TOCL***	***LTL***
Structural	*Yes*	*No*	*No*
Behavioral	*No*	*Yes*	*Yes*
Temporal	*No*	*Yes*	*Yes*
A-temporal	*Yes*	*No*	*No*

So, all methods allow the design and verification of static semantics based on OCL with exception to the xDSML design pattern (discussed hereafter) that allow the specification of LTL properties. However, OCL can only be used for the specification of a-temporal properties and structural properties. Other types of properties such as temporal properties or behavioral properties are out of the scope of these methods / approaches (with exception of the xDSML design pattern).

Dynamic semantics: the second information that cannot be captured by an abstract syntax or a concrete syntax is the behavior. For this purpose, a DSML must define dynamic semantics, also known as "executable semantics". Dynamic semantics is generally neglected from the specification of a DSML. However, for the purpose of model dynamic V&V, it is mandatory, becoming a crucial point in the specification of a DSML. DSML that include dynamic semantics are denoted executable DSMLs or xDSML. xDSMLs can be used to execute designed models allowing simulation as a way for model V&V. There are currently several ways to design xDSMLs. For instance, a design pattern for xDSMLs is proposed in (Combemale et al. 2012), allowing a state-based execution. This approach is synthetized in Figure 11 as a composition of five metamodels related to each other.

- The *Doman Definition MetaModel (DDMM)* defines the structural part of a DSML (i.e., the abstract syntax), composed of domain classes and references. The behavioral part, i.e., the execution-related information, is spread across the other four parts.
- The *State Definition MetaModel (SDMM)* defines a set of states for a set of preselected domain classes from the DDMM, denoted "evolving classes". Each

state represents the possible result in which instances of evolving classes can evolve during execution. Consequently the classe's behavior is represented as a successive change of states provoked by stimuli.

- The different types of stimuli (events) and their relationship with domain classes are defined in the *Event Definition MetaModel (EDMM)* package. Two types of stimuli are distinguished: *exogenous stimuli*, this type of stimuli are injected by the environment (e.g., an interaction is requested by the user), and *endogenous stimuli*, this type of stimuli are produced internally by another evolving concept.
- The relationship between the state model defined in the SDMM package and its reaction provoked by stimuli from the EDMM package is defined in the fourth *Semantics* package. The semantics package defines when stimuli are sent and the consequent reaction. It either be defined as operational semantics or as translational semantics (discussed below).
- Last but not least is the *Trace Management MetaModel (TM3)* package. TM3 provides monitoring mechanism for model execution trace.

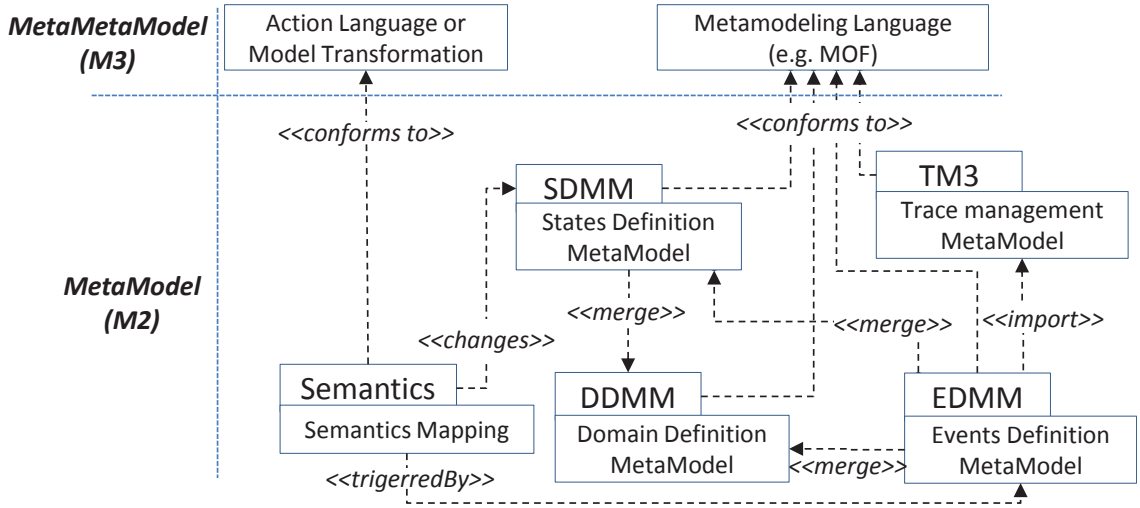


Figure 11. The executable DSML Pattern (Combemale et al. 2012)

An xDSML metamodel is then naturally equipped to support state-based execution, containing the classes' states, triggering events and a trace mechanism. The real behavior however (i.e., the mechanism that defined when transitions are fired and the produced reaction) is defined in the Semantics package.

It is also possible to design executable DSML without necessarily following this design pattern. However, in this case the metamodel of the DSML contain only structure-related information (similarly to the DDMM), excluding any execution-related

information (e.g., states, transitions, trace mechanism, etc.). The dynamic semantics of such DSML must implicitly define the execution related information and the way this information is computed. This way of building xDSML is for instance discussed in (Muller et al. 2005), proposing the design of xDSML based on "execution weaving" using the executable-metamodeling language Kermeta. Dynamic semantics that is directly provided to a DSML is denoted operational semantics. In contrary, dynamic semantics might be provided by other third-party executable approaches, based on transformations. For instance, the approach proposed in (Rivera & Vallecillo 2007) is targeting the Maude formal environment for model execution.

Operational semantics describes the behavior of a DSML and is used to execute (i.e., to interpret) models using the virtual machine of the language that is used to define the operational semantics. There are three different techniques to define operational semantics: 1) by an *endogenous transformation*, 2) by an *action language* and 3) by a *formal behavioral modeling language.*

Endogenous transformation is a declarative and rule-based technique for specifying transformations rules between concepts of the same metamodel, as discussed in Section 2.1.3. For instance, Figure 12 shows the behavior of the process of aging of the concept Person from Figure 8. There are currently several frameworks based on endogenous transformations, applied in a MDE/MBSE context such as (Markovic & Baar 2008) or (Hausmann 2005).

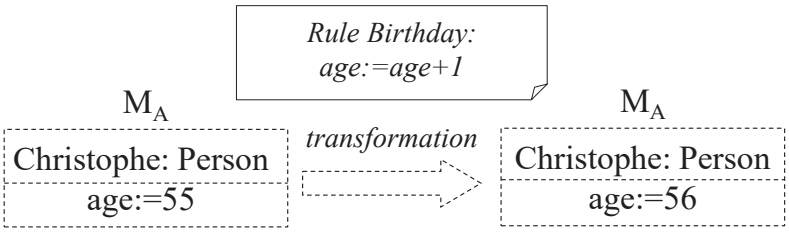


Figure 12. Operational semantics designed by endogenous transformations

Action language such as Java or Kermeta can be used to design operational semantics as a set of operations, methods or functions (depending on the used technique). Figure 13 illustrates the aging process of a person designed by the action language Kermeta. There are currently several frameworks equipped with an action language and applied in a MDE/MBSE context such as EMF (Steinberg et al. 2008), the Kermeta framework (Fleurey 2006), etc. The EPROVIDE framework (Sadilek & Wachsmuth 2009) allows

the specification of operational semantics for a DSML and is not related to a single technology, allowing a choice between Java, Prolog, ASM or QVT.

```
@aspect "true"
class Person{
        attribute age : Integer

        operation birthday() : Void is do
                age := age + 1
        end
                …
end
```

Figure 13. Operational semantics designed by action languages (Kermeta).

Formal behavioral modeling language such as Statecharts (Harel 1987), Petri Nets (Murata 1989), or Finite Automata (Kohavi & Jha 2009) when integrated with a metamodeling language, can be used to express operational semantics for a DSML. Instead of operations, in this case operational semantics is defined through behavior models. So rather than programming, a behavior is, in this case, modeled. Figure 14 shows an example of operational semantics designed by the Finite Automata language. The designed automata-like behavioral model defines the aging process of a person. Among the principle effective and currently used solutions based on formal behavioral modeling are: Real-Time UML (Douglass 2002), Scheidgen's approach for human comprehensible specifications of operational semantics (Scheidgen & Fischer 2007) and xMOF (Mayerhofer et al. 2013).

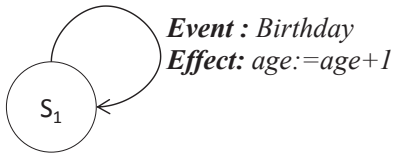


Figure 14. Operational semantics designed by the State machine a formal behavioral modeling language.

Translational semantics. Apart from the realm of modeling languages, there are several tool-equipped environments based on automata-like formalisms: StateMate (Harel & Naamad 1996), UPPAAL (Larsen et al. 1997), the finite state model of computation of Ptolemy (Lee 2003) or the Stateflow module in The MathWorks Simulink framework (Mathworks 2014). They provide graphical editor for simulation and animation

purposes, active states, fireable transitions and simulation trace. However, there is a gap between these approaches and the realm of modeling languages. This gap can be bridged by using model transformation techniques, as discussed in Section 2.1.3. So the dynamic semantics of a DSML are provided by a target approach that is usually formal and tool-equipped allowing various simulation but also property proof. This type of transformation is also called *exogenous transformations*, i.e., transformations between models expressed in different languages (Mens & Van Gorp 2006) and can be specified by using a graph transformations technique (Rozenberg & Ehrig 1997).

Composability: following the theory and principles of multi-viewpoint modeling, various interconnected models are designed for a given SoI as suggested above, which when put together, form a "composite model", covering a more expressive, realistic and complete representation of a SoI. In a similar way, the whole behavior of a SoI can be represented by mixing or aggregating the behaviors described by composing viewpoint models, even though these behaviors might be based on different functioning hypothesis (e.g., different level of details, different objectives, etc.). The V&V analyses become in this sense more relevant when considering composite models (e.g., a more realistic SoI simulation that coordinately executes all viewpoint models). This consists in interconnecting the dynamic semantics of designed DSMLs and in using these semantics simultaneously to execute composite models. However, the current MBSE modeling languages remain insufficient for the design and simulation of composite models.

Table 2 compares same of the methods for the design of xDSML discussed above based on the nature of the used behavioral language (action language or formal behavioral modeling language) and the ability to compose various dynamic semantics. In addition, we classify behavioral modeling languages into three categories (discrete-events, continuous or hybrid). The composability characteristics when using behavioral modeling language is divided into three categories: 1) when composing behavioral models of same type (e.g., only discrete-events) that are create by the same behavioral modeling language; 2) when composing behavioral models of same type that are create by different behavioral modeling languages (e.g., state machine and petri-net behavioral models); 3) when composing behavioral models of different types (e.g., discrete-events and continuous) that are create by different behavioral modeling languages.

Table 4. Comparison of several approaches for the design of DSML.

Methods / Approaches	***Action language***		***Modeling language***							***Is tool-equipped***
				Type			*Composability*			
	Yes/No	*Composability*	*Yes/No*	*discret*	*continous*	*hybrid*	*same type and lan.*	*same type dif. lan.*	*dif. type and lan.*	
Kermeta	*Yes*	*Yes*	*No*	*/*	*/*	*/*	*/*	*/*	*/*	*Yes*
EMF	*Yes*	*Yes*	*No*	*/*	*/*	*/*	*/*	*/*	*/*	*Yes*
GEMOC studio	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*Yes*
xMOF	*No*	*/*	*Yes*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*Yes*
fUML	*No*	*/*	*Yes*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*Yes*
xDSML design pattern	*Yes*	*Yes*	*?*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*Yes*

So, Kermeta and EMF relay on action languages for the design of xDSML, allowing also composability. xMod and fUML relay on discrete-events behavioral modeling languages, allowing composability. GEMOC studio and the xDSML design pattern allow both action languages and behavioral modeling languages.

2.3.4 Synthesis

Considering the first problematics of this work (introduced in Section 0) related to the design of models we focus on the design, use and management of DSMLs. For this purpose a DSML is defined by an *abstract syntax* that define the domain concepts and relationships through a set of classes and references, and a *concrete syntax* that defines the representation of the DSML (i.e., the representation of models created by a DSML).

Considering the second problematic of this work (also introduced in Section 0) related to model V&V analyses, we focus on the design of V&V suitable DSML. The lack of semantics from the DSML specification is, according to (Chapurlat 2013), the main limitation preventing the deployment of successful model V&V strategy. Namely, in addition to an abstract syntax and a concrete syntax, a DSML must also integrate semantics. Semantics define the domain knowledge that cannot be implicated by an abstract syntax and a concrete syntax, i.e., a set of constraints and additional

information concerning the structure or the behavior, named *static semantics*, and the behavior, named *dynamic semantics*.

Dynamic semantics can either be directly defined for a DSML, denoted *operational semantics*, or provided by third party formalisms thought transformations, denoted *translational semantics*.

The main benefit of the approaches based on translational semantics is the reuse of appropriate formal tool-supported target space usually based on Automata-like formalisms. This allows them, on the one hand, an easy access to V&V processes (i.e., model simulation and animation, simulation trace, property verification, etc.), but on the other hand, the analysis results are only available in the target spaces, so they should always be interpreted back to the source space, to compare the result based on the source model. The relevance between source and target models should be demonstrated to assure that the behavior defined by the target model corresponds to the one of the source model. In addition, a good knowledge and expertise in the chosen target domain and in transformation languages and tools is required.

In contrast, since the domain space is well-known to designers, it is easier to define the domain behavior directly on a given DSML, rather than using third party formalisms. This is the purpose of operational semantics, allowing model simulation and animation, as early as possible with minimum effort improving system quality and reducing time-to-market. Operational semantics are preferable for prototyping in particular for simple behavior that can be expressed through discrete states.

2.4 Conclusion and Contributions of this thesis

The objective of this work is to develop a method for the design, verification and validation of models that are used by stakeholders to understand a SoI, to communicate and argue with other actors about this SoI and finally to support them and increase their confidence during decision making processes.

The method must address four SE challenges introduced in Chapter I. In particular, it must assure the autonomy of different stakeholders involved in the process of complex system modeling, during the process of designing, intuitively and as simple as possible, models that contain their domain knowledge, but also to verify and validate these models. A critical analysis of the relevant literature concerning the design, the verification and the validation of models is previously presented.

The work presented throughout the rest of this manuscript converges through the proposal of a method for the design, the verification and the validation of models. To this end, our method must first guide and assist stakeholders to design their own modeling languages, particularly tailored for their domain knowledge and used to model a particular viewpoint of the SoI, named domain specific modeling languages (DSMLs). Second, DSML must be usable for the design of models, but also, on the one hand, for the simulation of models, and on the other hand, for the specification and verification of formal properties based on designed models.

The scientific positioning of this approach is discussed in the next section, considering the context of this work presented in Chapter I and the relevant literature presented in this chapter.

2.4.1 Scientific positioning

The method that we propose is intended for stakeholders that take part in a project of complex systems engineering, particularly in the upstream processes of system specification and modeling. Motivated by the current rising challenges in systems engineering that were identified by the AFIS (AFIS 2012) and discussed in Chapter I, this method aims to contribute in the following:

- To provide architects and engineers with the means for modeling, checking and simulating covering total system representation as requested in large and heterogeneous systems engineering processes.
- To improve model V&V respecting the MBSE principles.

Similar challenges, related to systems modeling and early verification and validation based on models to improve the software development processes, have been studies in the field of Software Engineering for Complex and Cyber-physical systems. A good example is the ongoing GEMOC initiative (Combemale 2016). The goals of this initiative are “to coordinate and disseminate the research results regarding the support of the coordinated use of various modeling languages that will lead to the concept of globalization of modeling languages, that is, the use of multiple modeling languages to support the socio-technical coordination required in systems and software engineering”. In other words, they highlight the problems of modeling and simulation covering total system representation by various and heterogeneous DSMLs, coordinated simulation of models, simulation trace, verification of properties, etc.

Our method is intended for the systems engineering community. As a starting hypothesis, we consider that systems engineering stakeholders are much less competent with programming and behavioral coordination languages, or with omniscient debugging, then software engineering stakeholders. Our goal is to assist and guide systems engineering stakeholders to design their own DSML and to relate them with the DSMLs of other stakeholders, to create models that can be simulated and animated considering also the models of other stakeholders, but in addition, to specify and verify properties considering either one viewpoint model or all viewpoint models of a SoI.

2.4.2 Expected contribution

The contributions of this thesis are here-after discussed from three different perspectives, i.e., from *conceptual perspective, methodological perspective* and *technical perspective*.

The conceptual contribution of this thesis is a metamodeling language that allows the design and integration of DSMLs suitable used to model, verify and validate different complementary viewpoints of a SoI. Such DSMLs are composed of:

- *Heterogeneous and Dependent abstract syntaxes:* abstract syntaxes that capture all concepts and relationships of different and heterogeneous viewpoints of a SoI through metamodels, but also the dependencies between different metamodels, providing an overall composite abstract syntax that covers the whole SoI.
- *Heterogeneous and Dependent concrete syntaxes:* concrete syntaxes that define the representation of concepts and relationships of a given viewpoint, but also concept dependencies between different viewpoints, providing a complex multi-viewpoints SoI representation that allows the navigation from one SoI viewpoint to another.
- *Heterogeneous and Dependent property specifications:* property specifications that contain properties for each individual viewpoint, but also properties that cover the dependencies between viewpoints.
- *Heterogeneous and Dependent operational semantics:* operational semantics that define the behavior of a viewpoint DSML, but also the behavioral dependencies with other viewpoint DSMLs.

The methodological contribution of this thesis is presented in a form of an approach that allows modeling a SoI, considering different viewpoints for different stakeholders,

by different DSMLs. These stakeholders are provided with the means first to create DSMLs and second to specify the dependencies (syntactically and semantically) between different DSMLs. Such DSML can be used to create models for different viewpoints of a SoI, but also to specify the dependencies between different viewpoints. Our approach should allow:

- *The simulation of different viewpoints* – a synchronized model execution, considering the operational semantics from all DSML that are used to model different SoI viewpoints and a new execution mechanism that integrate the blackboard design pattern and several rules that we introduce.
- *The formal proof of different viewpoints* – a formal verification of properties based on the SoI models, first considering each model individually and then together with the other models of the SoI.

Considering ***the technical contribution*** of this work we propose a complete implementation of the approach within the Eclipse environment through several deployable plugins.

2.4.3 Illustrative examples

Throughout the rest of the manuscript, we illustrate our contributions based on three case study examples:

- The first is a DSML denoted WaterDistrib for modeling water storage and distribution systems. This DSML is used to demonstrate the design of operational semantics using a behavioral modeling language, allowing experts to observe the changing water level in a water tank. Briefly, this DSML introduce the following concepts: a water tank, a water-source that is connected to the tank with pipes and a control station. A house is supplied with water by the mean of the tank. There are valves on each of the pipes, controlled (opened or closed) by a control station, based on the water request and the water level inside the tank.
- The second is the *Interpreted Sequential Machine* (ISM) (Vandermeulen 1996). ISM is a formal language based on discrete-event hypothesis for modeling and verifying the behavior of systems and their interactions with the environment, in particular, it allows describing sequential automatisms such as Control Part of Manufacturing Systems. This DSML contains a predefined formal semantics and is used to demonstrate the design of operational semantics using our formal

rule-based language. The idea is to rewrite the predefined formal semantics with slide changes using the rule-based language and to use them for simulation.

- The third is composed of two languages from the MBSE community: eFFBD (Enhanced Functional Flow Block Diagram) (INCOSE 2010) and PBD (Physical Block Diagram) (Long 2007). eFFBD is a functional-modeling language for the design of functional and behavioral aspects of complex, distributed, hierarchical, concurrent and communicating systems. PBD is a block-modeling language that is complementary to eFFBD. It provides systems engineers with a block-and-line diagram representing the physical components of a system or system segment and links that connect components through interfaces, offering a detailed view of an architectural composition. The goals of this final case study are to demonstrate the specification of syntactical as well as semantical dependencies between different DSMLs and how these dependencies are considered during simulation and property proof.

CHAPTER III

MODELING BASED ON PROPERTIES

This chapter presents a part of the conceptual and the methodological contributions of this work. A map of Chapter's outline with respect to the type of contributions is shown in Figure 15.

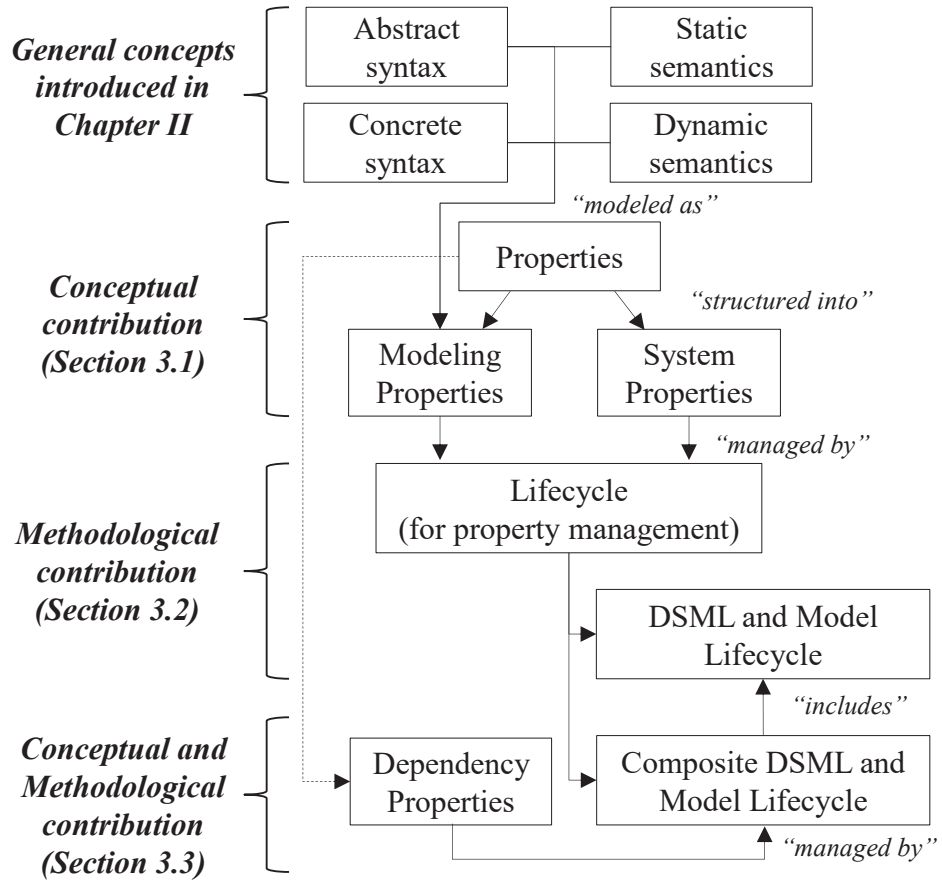


Figure 15. Map of conceptual and methodological contributions of Chapter III.

It is structured as follows. First, Section 3 introduces the core concept "Property" along with a property typology. Section 3.2 describes a formalized lifecycle for property management. The lifecycle provides stakeholders with guidelines, i.e., several phases and sub-phases, each one characterized by various constraints, expectations and rules to be considered and modeled as properties for the design and V&V of DSMLs and models. Section 3.3 introduces our vision on the multi-viewpoint modeling (i.e., modeling of a system considering simultaneously multiple viewpoints) based on the concept of property along with a modified version of the lifecycle for property management. Finally, Section 3.4 concludes this chapter.

3.1 The concept of "Property"

A property is defined as follows:

Definition 1: A ***property*** is a provable or evaluable (i.e. quantifiable or qualifiable) characteristic of an artefact [that is 1) a system S, or 2) a model M of S built for achieving a design objective] that translates all or part of stakeholder expectations to be satisfied by this artefact (Chapurlat 2013).

Depending on whether properties are used for the design of modeling artefacts or for the specification of requirements (defined in the next), they are structured into *modeling properties* and *system properties*.

Modeling properties are defined as follows:

Definition 2: A ***modeling property*** expresses the characteristic of a modeling artifact. It is used to conceptualize domain knowledge through modeling languages but also to concretize this domain knowledge through models.

The purpose of modeling properties is to support and answer some of the stakeholders' questions about the model of a future system. This allows verification of both model and SoI (see Section 0 for more details on verification).

System properties are defined as follows:

Definition 3: A ***system property*** expresses a part of the requirements that can furthermore be checked based on a modeling artefact that is defined by modeling properties.

The terms "requirements", "system requirements" and "stakeholder requirements" are standardized by (ISO/IEC 2008) as follows:

Definition 4: A ***requirement*** is a statement that identifies an operational, functional or design characteristic or constraint (of a product or process), which is unambiguous, testable or measurable, and moreover necessary for product or process acceptability.

Definition 5: A ***stakeholder requirement*** is a requirement for a system that can provide the services needed by users and other stakeholders within a defined environment.

Definition 6: A ***system requirement*** is a statement that transforms the stakeholder's user-oriented view of desired capabilities into a technical view of a solution that meets the user's operational needs. System requirements are specified by designers, either based on existing standards, best practices, or

induced by technological choices or existing technical solutions, e.g., COTS (Maiden & Ncube 1998).

A requirement must be clear, unambiguous and well-defined prior to any use then prior to any translation of corresponding system properties. These properties are then used to assume a part of validation of SoI models (see Section 0 for more details on validation).

3.1.1 Modeling properties

Modeling properties are structured into two categories:

1) Modeling properties used to *conceptualize domain knowledge* through modeling languages (DSMLs)
2) Modeling properties used to *concretize domain knowledge* through a model (created by using a DSML that conceptualize domain knowledge)

The modeling properties used to conceptualize domain knowledge are classified into:

- Structural properties (SP)
- Representational properties (RP)
- Behavioral properties (BP) and
- Constraint properties (CP)

Structural properties (*SP*) are defined as follows:

> ***Definition 7**: A **structural property** expresses characteristics about the structure of a domain, conceptualizing domain knowledge through a set of concepts denoted domain concepts, and relations that bound together these concepts. The set of SP defines the abstract syntax of a DSML.*

A domain concept is defined by a set of common characteristics and specifies various representatives from a given domain knowledge, e.g., a Function or a Flow as shown in the next illsutrative example. These representatives are called in the next domain objects, e.g., the functions ‘close the door’ or ‘empty the store’.

There are different techniques to formalize structural properties, e.g., by a metamodel, by an ontology, etc. This work, for the design of structural properties focuses on metamodels. Metamodels are designed by a metamodeling language such as for example the OMG’s standard MOF (OMG 2015a) (see Section 2.1.2 for more details).

To illustrate, we show in Figure 16 a metamodel that represents a part of the structural properties of the eFFBD language (INCOSE 2010) introduced in Section 2.4.3. Among the core concepts of the eFFBD are Function, Item Flow and Resource Flow, a set of typed attributes detailing each of these concepts (e.g., quality and quantity of a Resource, purpose of a function) and a set of relationships between them (e.g., a relationship inputs between Item and Function). They are formalized through **classes** and various **relationships** (references, compositions and inheritances) as shown in Figure 16.

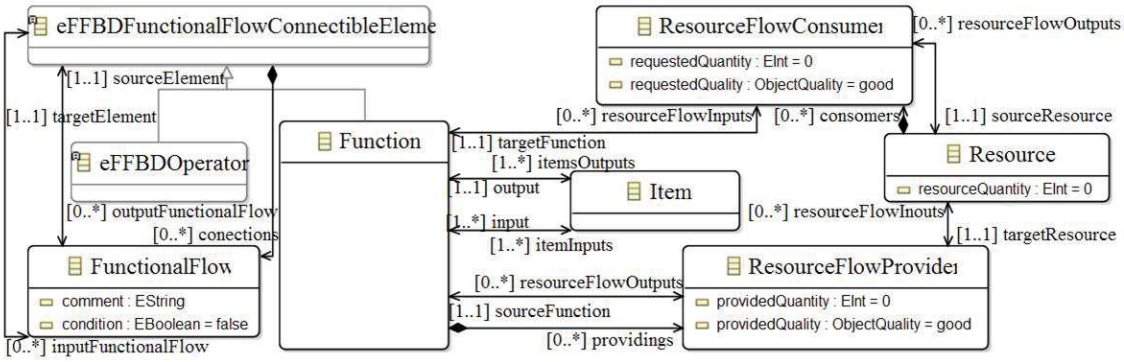


Figure 16. A metamodel that specify a part of the *SP* of the eFFBD language.

Structural properties are formally defined as $SP := \langle CPT, REL \rangle$, where:

- $CPT := \{cpt_i | cpt_i \in C \vee cpt_i \subset C, i \in \mathbb{N}\}$ is a set of domain concepts and C is a set of classes. Domain concept can either be simple, modeled by a single classes ($cpt_i \in C$) or more complex, modeled by several classes ($cpt_i \subset C$). For instance, considering the example discussed above (see Figure 16), the core concept Function is modeled by one class (i.e., the class Function), whereas the concept Resource Flow is modeled by several classes (i.e., Resource Flow Provider, Resource Flow Consumer and Resource). Details about the formal specification of classes and class related information (e.g., mutable and immutable attributes, class attributes, etc.) are available at (Weisemöller & Schürr 2008) and (OMG 2015a).
- $REL := \langle S, T, type \rangle$ is a set of relationships between classes where:
  - $S \in C$ defines the source class
  - $T \in C$ defines the target class
  - $type \in \{'reference', 'composition', 'inheritence'\}$ defines the relationship type. Details about the formal specification of different types

of relationships are available at (Weisemöller & Schürr 2008) and (OMG 2015a).

Representational properties (*RP*) are defined as follows:

> ***Definition 8**: A **representational property** expresses characteristics about the representation of domain concepts and relations. The set of RP defines the concrete syntax of a DSML.*

Representational properties are formalized by a concrete syntax language. There are two categories of concrete syntax languages, one for the design of graphical concrete syntaxes (e.g., Diagraph (Pfister et al. 2014) or Obeo Designer (Juliot & Benois 2010)), and the other for the design of textual concrete syntaxes (e.g., xText (Bettini 2013)). Section 2.3.2 provides more details on this topic.

To illustrate, Figure 17 shows the graphical representational properties for the metamodel (structural properties) illustrated in Figure 16.

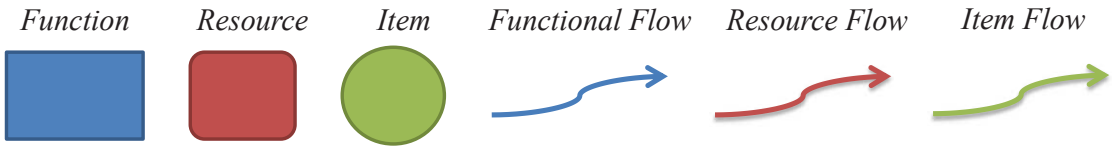


Figure 17. Graphical RP for the elements of the eFFBD language.

For instance, the graphical representation of the concept Function is defined as a blue rectangular form. An eFFBD model is graphically represented based on these graphical representational properties as shown in Figure 20. Note that the graphical representational properties shown in Figure 17 are only schematized and must furthermore be formalized by an adequate concrete syntax language. For instance, the Diagraph approach can be used to formalize these representational properties.

Representational properties are formally defined as $RP := \langle type, RI, \theta_r \rangle$, where:

- $type \in \{'graphical', 'textual'\}$ defines the representation type.
- $RI := \{ri_i | ri_i \in CL, i \in \mathbb{N}\}$ is the set of representational information that define the concrete representation of domain concepts and relationships. *CL* is a concrete syntax language used to formalize the representational information.
- $\theta_r : RI \rightarrow SP$ associates the representational information to structural properties, i.e., to a domain concept or a relation.

Behavioral properties (*BP*) are defined as follows:

Definition 9: A ***behavioral property*** expresses characteristics about the behavior of domain concepts. The set of *BP* defines the dynamic semantics of a DSML.

There are different techniques to design and formalize behavioral properties (e.g., by using action languages, behavioral modeling languages, formal languages, etc.), as discussed in Section 0. A particular interest is here-given on behavioral modeling languages, or simply behavioral languages. Behavioral languages are based on different functioning hypotheses: discrete-events, continuous or hybrid hypotheses, as proposed in Section 0. Chapter IV for example, introduces the behavioral language extended interpreted sequential machine (eISM) and demonstrates the design of discrete-events behavioral models using eISM. As illustration, Figure 18 shows an example of a discrete-events behavioral model (a finite stat machine model) that specifies the behavior of the concept Function as follows.

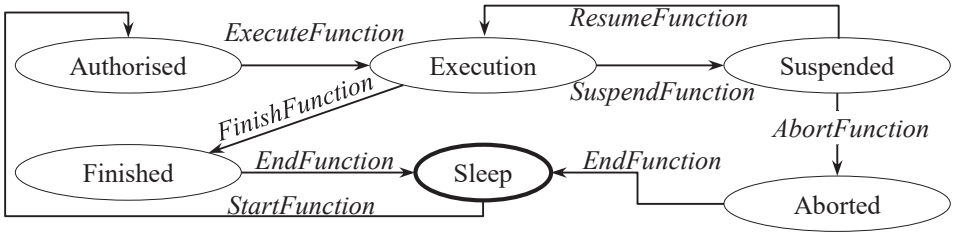


Figure 18. The BP for the concept *Function* of eFFBD.

A function defines an input/output transformation. The transformation is first possible (*Authorized)*, i.e., the function can start but waits for Items (and eventually Resources). The real transformation of energy, material and / or data (*Execution*) starts when the requested Items and Resources are provided. As a result, several output Items and Resources are provided (*Finished*). Due to external events (i.e., in case of dysfunction of the component on which a function has been allocated) a function can suspend or even abort execution (*Suspended*, *Aborted*). This example is furthermore detailed and formalized as an eISM model in Chapter IV.

Note that for the purpose of simulation (i.e., model execution) the behavioral models of different domain concepts must be coordinately used. This leverages the need for a synchronization mechanism allowing data and event exchanges between different behavioral models. Chapter IV introduces such mechanism for coordinated simulation based on the blackboard design pattern.

Behavioral properties are formally defined as $BP := \langle type, BM, \theta_b \rangle$, where:

- $type \in \{'discrete-events','continous','hybrid',\emptyset\}$ defines the behavior type.
- $BM := \{br_i | br_i \in BL, i \in \mathbb{N}\}$ is a behavioral model formalized through a set of rules br_i by using a behavioral language BL.
- $\theta_b : BM \rightarrow CPT$ associates a behavioral model to a domain concept.

Constraint properties (*CP*) are defined as follows:

> ***Definition 10:*** A ***constraint property*** expresses complementary characteristics that cannot be implicitly defined by a DSML. The set of *CP* defines the static semantics of a DSML.

For instance: "all persons (instances of a class Person) that have less than 18 years are minors, whereas the others are majors" is a classical constraint property that cannot be implicitly defined by a class Person.

Depending on which part of a DSML is concerned, constraint properties are classified into:

- structural constraint properties (SCP),
- representational constraint properties (RCP) and
- behavioral constraint properties (BCP)

Structural constraint properties (*SCP*) are defined as follows:

> ***Definition 11****: A* ***structural constraint property*** expresses complementary characteristics that cannot be implicitly defined by the domain structure (see *Definition 7*) of a DSML.

In this sense, **representational constraint properties** (*RCP*) are defined as follows:

> ***Definition 12****: A* ***representational constraint property*** expresses complementary characteristics that cannot be implicitly defined by the representation (see *Definition 8*) of a DSML.

Similarly, **behavioral constraint properties** (*BCP*) are defined as follows:

> ***Definition 13****: A* ***behavioral constraint property*** expresses complementary characteristics that cannot be implicitly defined by the behavior (see *Definition 9*) of a DSML (i.e., the behavior of concepts that form the structure of a DSML).

Illustartions for structural, representational and behavioral properties are proposed hereafter (see CP_1-CP_7).

Constraint properties are formalized by a constraint language. Different constraint languages can be used for the design of different constraint properties, i.e., structural, representational or behavioral. Some constraint languages such as the UPSL-SE (Chapurlat 2013) can be used for the design of multiple types of constraint properties (e.g., structural and behavioral).

The different types of constraint properties must be specified by using an adequate constraint language that is compatible with the DSML's structure, representation or behavior. For instance, if the behavior of a DSML concept is designed by a finite state machine model as shown in Figure 18, constraint languages such as the object constraint language (OCL) (OMG 2014) are not compatible and cannot be used. In contrary, if the behavior is designed by an action language as a set of operations for domain concepts, then the OCL can be used for the specification of behavioral constraint properties such as pre-condition, post-condition, body, etc.

In addition, a formal proof mechanism is requested to verify different types of constraints. Chapter V introduces such mechanism.

Language constraint properties (structural, representational and behavioral) are moreover classified into *a-temporal* and *temporal*. An overview is shown in Figure 19.

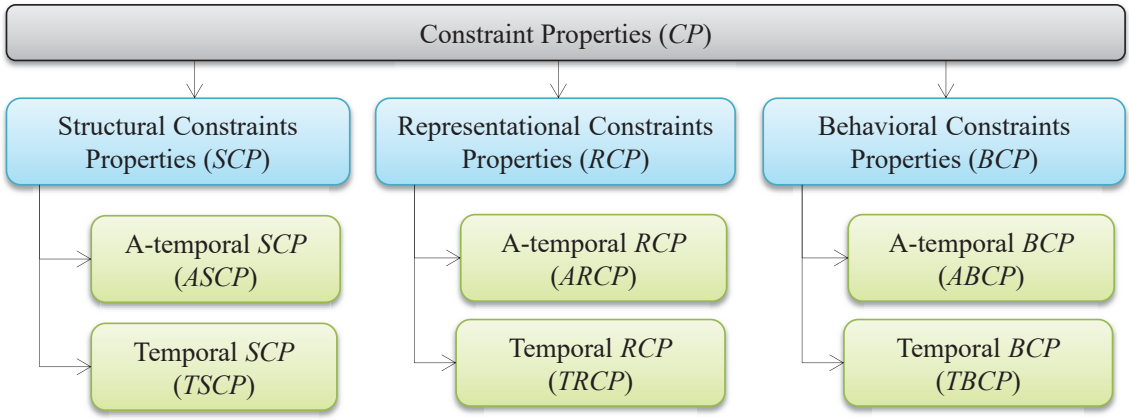


Figure 19. A classification of constraint properties *CP*.

A-temporal structural constraint properties (*ASCP*) are defined as follows:

> ***Definition 14:*** *An* ***a-temporal structural constraint property*** is a structural constraint property (see *Definition 11*) that does not take into account a temporal

dimension (is not time or execution related). *ASCP* are specified based on a domain structure (see *Definition 7*) and are verified based on the structure of a conforming model (see *Definition 20*).

To illustrate, we specify the following *A-temporal SCP* based on the metamodel shown in Figure 16:

CP_1: "If a Function has at least one input resource flow
then it must also have at least one output resource flow"

The above quoted property must be furthermore formalized by an adequate constraint language before being verified. The verification process takes place as soon as a model is designed. The feedback of the verification process is either positive (i.e., the model respects the property) or negative (i.e., the model violates the property and thus must be revisited for corrections).

Temporal structural constraint properties (*TSCP*) are defined as follows:

Definition 15: *A* ***temporal structural constraint property*** is a structural constraint property (see *Definition 11*) that takes into account a temporal dimension (is time or execution related). *TSCP* are specified based on a domain structure (see *Definition 7*) and are verified based on the structure of a conforming model (see *Definition 20*) during model execution (in contrary to *ASCP* that are verified before model execution).

To demonstrate, we specify the following *Temporal SCP* based on the metamodel illustrated in Figure 16:

CP_2: "The quantity of Resources must always
(i.e., each execution step) be positive or nul"

The above quoted property must be formalized by an adequate constraint language before being verified. The verification process takes place as soon as a model is designed and executed. A feedback is provided after or during the model execution. For this type of properties, a model-checker must be integrated with a simulator, as for instance proposed by UPPAL (Larsen et al. 1997).

A-temporal representational constraint properties (*ARCP*) are defined as follows:

Definition 16: *An* ***a-temporal representational constraint property*** is a representational constraint property (see *Definition 12*) that does not take into

account a temporal dimension (is not time or execution related). *ARCP* are specified based on the representation of domain concepts and relations (see *Definition 8*) and are verified based on the representation of a conforming model (see *Definition 21*).

To illustrate, we specify the following *A-temporal RCP* based on the concrete syntax illustrated in Figure 17:

CP_3: "Functions connected with at least three Resources must be graphically represented in a red color"

The above quoted property must be formalized by an adequate constraint language before being verified. The verification process takes place as soon as a model is designed and represented. The feedback of the verification process is either positive (i.e., the model respects the property) or negative (i.e., the model violates the property and thus must be revisited for corrections).

Temporal representational constraint properties (*TRCP*) are defined as follows:

> ***Definition 17:*** *A* ***temporal representational constraint property*** is a representational constraint property (see *Definition 12*) that takes into account a temporal dimension (is time or execution related). *TRCP* are specified based on the representation of domain concepts and relations (see *Definition 8*) and are verified based on the representation of a conforming model (see *Definition 21*) during model execution (in contrary to *ARCP* that are verified before model execution).

To illustrate, we specify the following *Temporal RCP* based on the concrete syntax illustrated in Figure 17:

CP_4: "Items must change color each three execution steps"

The above quoted property must be formalized by an adequate constraint language before being verified. The verification process takes place as soon as a designed model is executed. A feedback is provided after or during the model execution. For this type of properties, a model-checker must be integrated with a simulator.

A-temporal behavioral constraint properties (*ABCP*) are defined as follows:

> ***Definition 18:*** An ***a-temporal behavioral constraint property*** is a behavioral constraint property (see *Definition 13*) that does not take into account a temporal

dimension (is not time or execution related). *ABCP* are specified and checked based on the behavior of domain concepts (see *Definition 9*) of a DSML, before creating or simulating models.

ABCP are used to verify the well-formedness of the behavior. In order to do so, the behavior must respect:

- *The hypotheses of the used behavioral language*: the behavioral language imposes several hypotheses that designed behavioral models must respect. For instance:

  CP_5: "A finite state machine model must have an initial state (otherwise the model is false)"

  A behavioral model must verify all hypotheses imposed by the used behavioral modeling language before being used for the purpose of simulation.
- *Alternative or Stakeholders' hypotheses*: sometimes stakeholders impose, in addition to the hypotheses of a behavioral language, several other hypotheses. For instance:

  CP_6: "A finite state machine model is invalid if it possesses a state without an outgoing transition"

  The above quoted constraint can locally be applied on preselected finite state machine models. The verification process takes place as soon as a behavioral model is designed, before being used for the purpose of simulation.

Temporal behavioral constraint properties (*TBCP*) are defined as follows:

Definition 19: A ***temporal behavioral constraint property*** is a behavioral constraint property (see *Definition 13*) that takes into account a temporal dimension (is time or execution related). *TBCP* are specified based on the behavior of domain concepts (see *Definition 9*) of a DSML and checked based on the model behavior (see *Definition 22*) during model execution (in contrast to *ABCP* that are checked before execution).

For instance:

CP_7: "A finite state machine model must enter in a specific state (i.e., state n) after 10 execution steps"

The above quoted property must be formalized by an adequate constraint language before being verified. The verification process takes place as soon as a designed model is executed. A feedback is provided after or during the model execution. For this type of properties, a model-checker must be integrated with a simulator.

Constraint properties are formally defined as $CP := \langle SC, RC, BC \rangle$, where *SC* is the set of structural constraint properties, *RC* is the set of representational constraint properties and *BC* is the set of behavioral constraint properties.

Structural constraint properties are formally defined as $SC := \langle type, SCP, \theta_{scp} \rangle$, where:

- $type \in \{'temporal', 'a-temporal'\}$ defines the type of the structural constraint property.
- $SCP := \{scp_i | scp_i \in CL, i \in \mathbb{N}\}$ is the structural constraint property formalized through formal rules scp_i by using a constraint language *CL*.
- $\theta_{scp}: SCP \rightarrow SP$ associates a structural constraint property to a domain concept or relationship.

Representational constraint properties are formally defined as $RC := \langle type, RCP, \theta_{rcp} \rangle$, where:

- $type \in \{'temporal', 'a-temporal'\}$ defines the type of the structural constraint property.
- $RCP := \{rcp_i | rcp_i \in CL, i \in \mathbb{N}\}$ is the representational constraint property formalized through formal rules rcp_i by using a constraint language *CL*.
- $\theta_{rcp}: SCP \rightarrow RP$ associates a representational constraint property to the representation of a domain concept or relationship.

Behavioral constraint properties are formally defined as $BC := \langle type, BCP, \theta_{bcp} \rangle$, where:

- $type \in \{'temporal', 'a-temporal'\}$ defines the type of the behavioral constraint property.
- $BCP := \{bcp_i | bcp_i \in CL, i \in \mathbb{N}\}$ is the behavioral constraint property specified as formal rules bcp_i by using a behavioral constraint language *CL*.
- $\theta_{bcp}: BCP \rightarrow BP$ associates a behavioral constraint property to a behavioral model.

A DSML is formalized as a 4-uplet composed of modeling properties that conceptualize domain knowledge. Among modeling properties that concretize domain knowledge are: structural properties (*SP*), representational properties (*RP*), behavioral properties (*BP*) and constraint properties (*CP*). A DSML is then formally defined as follows:

$$DSML := \langle SP, RP, BP, CP \rangle$$

The modeling properties used to concretize domain knowledge through a model are classified into:

- Model structural properties (MSP)
- Model representational properties (MRP) and
- Model behavioral properties (MBP)

Modem structural properties (*MSP*) are defined as follows:

> ***Definition 20**: A **model structural property** expresses characteristics about the structure of a domain. It concretizes domain knowledge through a set of domain objects and links that are instances of domain concept and relations (see *Definition 7*). Objects and links concretize by an adequate value the characteristics of concepts and relations.*

Model structural properties are formalized by using a DSML, i.e., more specifically, the metamodel (i.e., the abstract syntax defined through the *SP*) of a DSML. The resulting structure must conform to the metamodel of the used DSML (see Section 2.1.2 for more details).

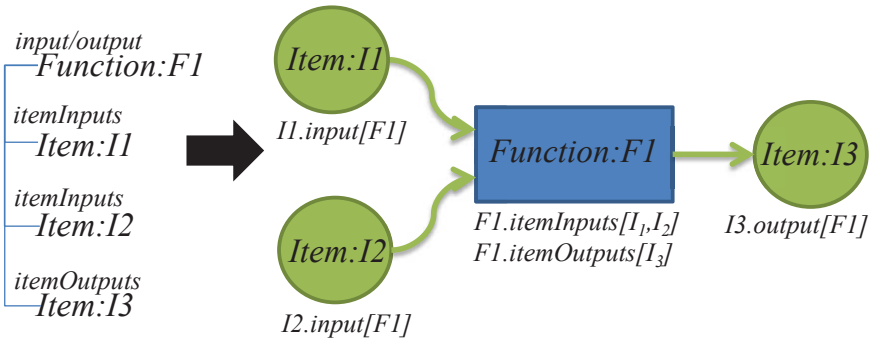


Figure 20. The structure (MSP) - left and the representation (MRP) - right of an eFFBD model.

To illustrate, the left side of Figure 20 shows model structural properties forming the structure of an eFFBD model. The model is composed of four objects, i.e., F1 instance

of the concept Function and I1, I2 and I3, instances of the concept Item, and six links between these concepts, instances of the references: input, output, itemInput and itemOutput.

Model structural properties are formally defined as $MSP := \langle O, L \rangle$, where:

- $O := \{obj_i | obj_i \; instanceOf \; CPT, i \in \mathbb{N}\}$ is the set of domain objects i.e. instances of domain concepts defined by the DSML's *SP*. *InstanceOf* is the relation of instantiation discussed in Section 2.1.2.
- $L := \left\{ link_i \middle| \begin{matrix} link_i \; instanceOf \; REL \wedge REL.type \in \\ \{'reference', 'composition'\}, i \in \mathbb{N} \end{matrix} \right\}$ is the set of links between objects that define the organization of objects in a model. Two types of links can be designed:
  - *Reference links* that are instances of the relation Reference. They are used to connect objects.
  - *Composition links* that are instances of the relation Composition used to embed objects (one object can contain other objects).

Model representational properties (*MRP*) are defined as follows:

> ***Definition 21**: A **model representational property** expresses characteristics about the representation of domain objects and links. *MRP* are used to parametrize the concrete syntax information (see *Definition 8*) for a given object or link, specifying the representation of this object or relation in an editor.*

Model representational properties are formalized by using a DSML, i.e., more specifically, the concrete syntax (i.e., the *RP*) of a DSML. Depending on the nature of a concrete syntax (graphical or textual), *MRP* provide a graphical or a textual representation of the structure of a model, forming a graphical or textual image inside an editor (see Section 2.3.2 for more details).

To illustrate, Figure 20 shows the graphical representation of the previously discussed eFFBD model. Note that the representational information *RI* that defines the concrete syntax illustrated in Figure 17, is parametrized for each object shown in Figure 20, i.e., the function F1 is graphically represented by a blue rectangular form with a given position and size, all items (I1, I2 and I3) are graphically represented by green circular forms, each one having different position but the same size and the links between objects are represented by green arcs.

Model representational properties are formally defined as $MRP := \langle ri, PRI, \theta_{pri} \rangle$ where:

- $ri \in RI$ is a representational information about domain concepts or relations formalized by a concrete syntax language *CL* (the formal definition of *RI* is provided above).
- $PRI := \{pri_i | i \in \mathbb{N}, i = |instances(\theta_r(ri))|\}$ is the set of different parametrizations pri_i for a given representational information *ri* based on different domain objects or links. The total number of parametrizations is equal to the number of objects or relations, instances of the domain concept or relations for which *ri* defines the representation. For instance, the representational information about the concept Item (shown in Figure 17) is parametrized three times for each object instance of Item (I1, I2 and I3) shown in Figure 20.
- $\theta_{pmo}: RI \times O \rightarrow PRI$ is the function that parametrizes the representational information *ri* by associating it with a domain object. Note that: $\forall o \in O, o\ instanceOf(\theta_r(ri))$, a representational information can be parametrized only by an object that is an instance of the domain concept for which *ri* defines the representation.
- $\theta_{pml}: RI \times L \rightarrow PRI$ is the function that parametrizes the representational information *ri* by associating it with links. Note that: a representational information can be parametrized only by a link that is an instance of the domain relation for which *ri* defines the representation.

Model behavioral properties (*MBP*) are defined as follows:

> ***Definition 22****: A* ***model behavioral property*** expresses characteristics about the behavior of domain objects. *MBP* are used to parametrize a behavioral model (that define the behavior of a domain concept *c*, see *Definition 9*) for an object (this object must be an instance of the domain concept *c*). The set of *MBP* defines the necessary information to execute the structure of a model.

Model behavioral properties are formalized by using a DSML, i.e., more specifically, the dynamic semantics (i.e., the *BP*) of a DSML. Before illustrating model behavioral properties, let's first introduce the behavior of the eFFBD *Item* concept shown in Figure 21. *Items* are transformed by functions during functions' execution. They are initially

not ready for transformation (*State: Not ready*). To precede transformation, items must be prepared, eventually reaching the requested quality and quantity, becoming ready (*State: Ready*) and the transformation can begin. During transformation, the quality and quantity of items changes and consequently items' state changes to the initial (*Not ready*) state. The behavior of the objects forming the model illustrated in Figure 20 is formalized by the *MBP* shown in Figure 22.

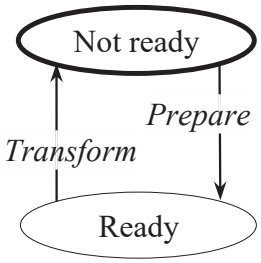


Figure 21. The *BP* of the concept *Item* of eFFBD.

The corresponding behavioral models (i.e., the state machine illustrate in Figure 18 for the concept Function and the state machine shown in Figure 21 for the concept Item) are parameterized for each object (i.e., for function F1, item I1, item I2 and item I3) as shown in Figure 22.

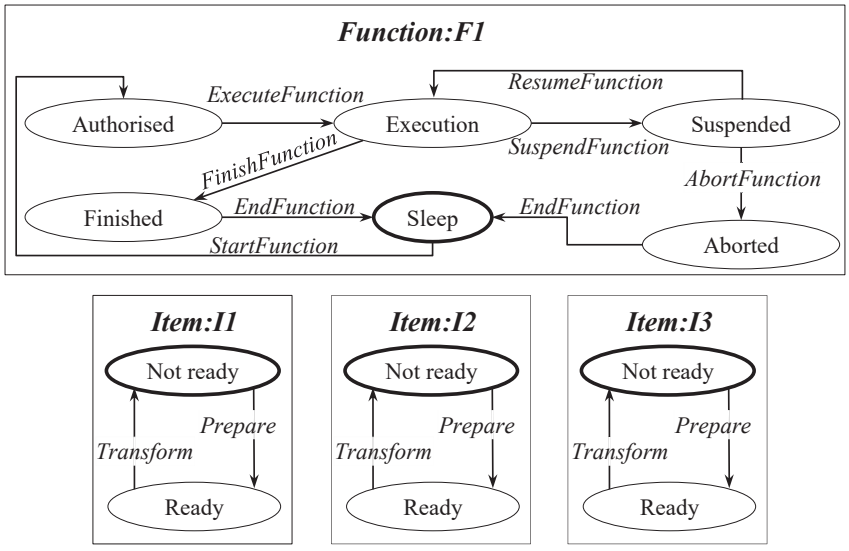


Figure 22. *MBP* for the model illustrated in Figure 20.

For the purpose of simulation (i.e., model execution) behavioral models must be coordinately executed (i.e., the parametrized behavioral model of F1 must be coordinated with the parametrized behavioral models of I1, I2 and I3), as discussed previously in Chapter II. This leverages the need for a synchronization mechanism

allowing data and event exchanges between different behavioral models. Chapter IV introduces such mechanism for coordinated simulation based on the blackboard design pattern.

Model behavioral properties are formally defined as $MBP := \langle bm, PBM, \theta_{pm} \rangle$ where:

- $bm \in BM$ is a behavioral model formalized through a set of rules by using a behavioral language *BL* (the formal definition of *BM* is provided above).
- $PBM := \{pbm_i | i \in \mathbb{N}, i = |instance(\theta_b(bm))|\}$ is the set of different parametrizations pbm_i of *bm* based on domain objects. The total number of parametrizations is equal to the number of objects, instances of the domain concept for which *bm* defines the behavior.
- $\theta_{pm}: BM \times O \rightarrow PBM$ is the functions that parametrizes the behavioral model *bm* by associating it with a domain object. Note that: $\forall o \in O, o\ instanceOf\ (\theta_b(bm))$, a behavioral model can be parametrized only by an object that is an instance of the domain concept for which *bm* defines the behavior.

3.1.2 System properties

Second, system properties that express parts of system or stakeholders requirements (see *Definition 3*) are used to concretize domain knowledge through a set of constraint properties focusing on a SoI model then respecting DSML properties defined above. The verification of system properties tends towards a certain level of model validity (see Section 0 for model validation).

The constraint properties that concretize system properties are structured into:

- Model constraint properties (MCP) and
- Object constraint properties (OCP)

Model constraint properties (*MCP*) are defined as follows:

> ***Definition 23:*** A ***model constraint property*** is a constraint property (see *Definition 10*) that is particularly tailored for and verified for one or more models that are selected by stakeholders.

Let's remind that a *CP* is defined for a DSML (e.g., the eFFBD DSML) and should be verified by any model created by this DSML (e.g., any eFFBD model). *CP* can thus be considered as "general" constraints. In contrary, a *MCP* is also defined for a DSML

(e.g., the eFFBD DSML), but it should be verified only by several preselected models created by this DSML (e.g., several preselected eFFBD models). *MCP* can thus be considered as more "specific" constraints in comparison to *CP*. For instance, the functional architectures (eFFBD models) used in the automotive industry might have some common requirements. These requirements apply only to the functional architectures of different automobiles and do not apply to the functional architectures of other systems.

Similarly to *CP*, *MCP* are classified into:

- model structural constraint properties (*MSCP*),
- model representational constraint properties (*MRCP*) and
- model behavioral constraint properties (*MBCP*)

Model structural constraint properties (*MSCP*) are defined as follows:

> ***Definition 24**: A **model structural constraint property*** is a *SCP* (see *Definition 11*) that is particularly tailored for and is verified based on the structure of selected models (see *Definition 20*).

In this sense, **model representational constraint properties** (*MRCP*) are defined as follows:

> ***Definition 25**: A **model representational constraint property*** is a *RCP* (see *Definition 12*) that is particularly tailored for and is verified based on the representation of selected models (see *Definition 21*).

Similarly, **model behavioral constraint properties** (*MBCP*) are defined as follows:

> ***Definition 26**: A **model behavioral constraint property*** is a *BCP* (see *Definition 13*) that is particularly tailored for and is verified based on the behavior used to execute selected models (see *Definition 22*).

Model constraint properties are also formalized by a constraint language. Different constraint languages can be used for the design of different model constraint properties, i.e., structural, representational or behavioral.

MSCP, *MRCP* and *MBCP* are moreover classified into *a-temporal* and *temporal*:

- A-temporal model structural constraint properties (AMSCP)
- Temporal model structural constraint properties (TMSCP)
- A-temporal model representational constraint properties (AMRCP)

- Temporal model representational constraint properties (TMRCP)
- A-temporal model behavioral constraint properties (AMBCP) and
- Temporal model behavioral constraint properties (AMBCP)

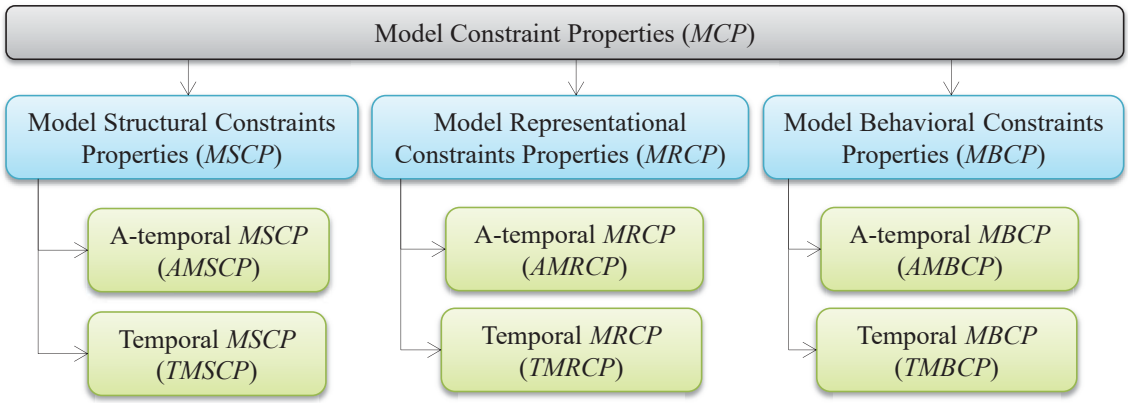


Figure 23. A classification of model constraint properties *MCP*.

An overview is shown in Figure 23. Definitions about the above quoted types of *MCP* are not provided since they correspond to the definitions of a-temporal and temporal *SCP*, *BCP* and *BCP* (see *Definition 14 – Definition 19*).

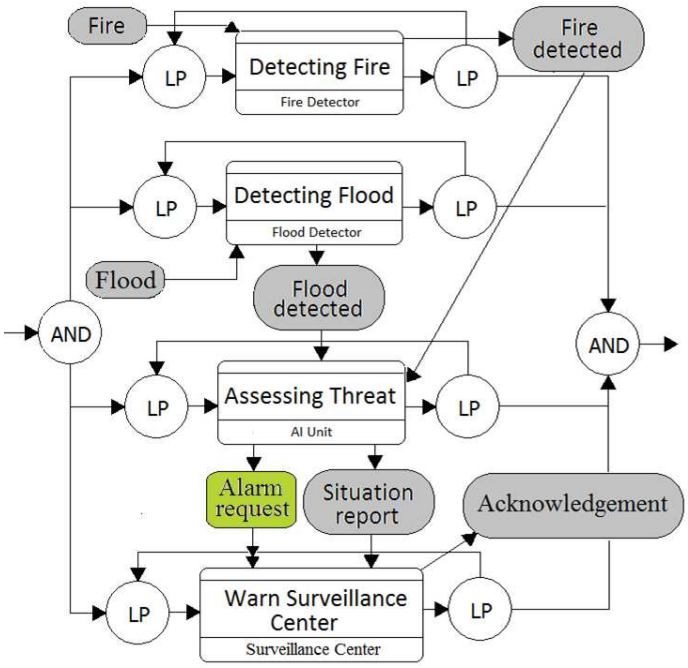


Figure 24. An eFFBD model for the functional architecture of a fire and flood detection system.

To illustrate several of the above quoted types of model constraint properties, let's first introduce the functional architecture of a fire and flood detection system through an

eFFBD model shown in Figure 24. For the rest of *MCP* that are not here illustrated, readers are encouraged to revisit CP_1 – CP_7 to get a general idea about the purpose of each type of property.

The functional architecture of our fire and flood detection system is composed of four main functions that operate non-stop (in an infinite Loop construct: LP) and in parallel (in a parallel construct: AND). Chapter IV provides details on the constructions in a functional architecture (AND, OR, Loop, Iterative, etc.). The Detecting Fire and the Detecting Flood functions provide information about a possible fire or flood threat to the Assessing Thread function. The latter, based on the received information, sends a report of the situation or triggers an alarm request, to the Warn Surveillance Center function that finally acknowledges the situations for further actions.

For the functional architecture shown in Figure 24, the following *AMSCP* can be specified considering the domain structure of the eFFBD DSML:

CP_8: "All functions must be performed infinitely (without an end) and in parallel"
(i.e., any Function must be included in an eFFBD construct named
Loop (LP) without loop exit condition and these LP constructs
must be placed in a parallelism construct AND)

The above quoted property must be formalized by an adequate constraint language before being verified. The verification process takes place locally (i.e., only for the functional architecture of a fire and flood detection system) and do not apply to other eFFBD models. The feedback of the verification process is either positive (i.e., all functions are performed infinitely (without an end) and in parallel) or negative (i.e., the model violates the property and thus must be revisited for corrections).

CP_8 can be complemented by the following *TMBCP* considering the behavior of the concept function shown in Figure 18:

CP_9: "After starting normal functioning (i.e., after the behavioral models of all functions
are in an execution state) functions must never (each execution step) finish execution
(i.e., the behavioral models of all functions must never enter finished state)"

The verification process takes place as soon as the eFFBD model shown in Figure 24 is executed. A feedback is provided after or during the execution. For this type of properties, a model-checker must be integrated with a simulator.

Model constraint properties are formally defined as $MCP := \langle MSC, MRC, MBC \rangle$, where:

- MSC is the set of model structural constraint properties,
- MRC is the set of model representational constraint properties, and
- MBC is the set of model behavioral constraint properties.

The formal specification of *MSC* is the very similar to *SC* with exception to the θ_m function. $MSC := \langle type, SCP, \theta_{scp}, \theta_m \rangle$, where (see the formal specification of *SC* for $type, SCP, \theta_{scp}$):

- $\theta_m : MSC \rightarrow Models$ is the function that allow stakeholders to preselect the models that must check the model constraint property where *Models = {Model$_i$ / $i \in \mathbb{N}$}* is the set of all models designed by using a DSML.

In this sense, the formal specification of *MRC* is the very similar to *RC* with exception to the θ_m function (defined above): $MRC := \langle type, RCP, \theta_{rcp}, \theta_m \rangle$.

The formal specification of *MBC* is the very similar to *BC* with exception to the θ_m function (defined above): $MBC := \langle type, BCP, \theta_{bcp}, \theta_m \rangle$.

Object constraint properties (*OCP*) are defined as follows:

> ***Definition 27:*** An ***object constraint property*** is a constraint property (see *Definition 10*) that is particularly tailored for selected objects (in contrast to *MCP* that are tailored for a selected model).

Let's reconsider the above discussed example of the functional architectures for the automotive industry. Within such context, an *OCP* can be used to specify a requirement about the engines of a specific car brand, or about the engine of one of the cars of that brand. So, *OCP* can be considered as more "specific" constraints in comparison to the *MCP*.

Similarly to *MCP*, object constraint properties are classified into:

- object structural constraint properties (*OSCP*),
- object representational constraint properties (*ORCP*) and
- object behavioral constraint properties (*OBCP*)

Object structural constraint properties (*OSCP*) are defined as follows:

***Definition 28**: An **object structural constraint property** is a SCP (see Definition 11) that is particularly tailored for and is verified based on the structure of preselected objects in a model.*

In this sense, **object representational constraint properties** (*ORCP*) are defined as follows:

***Definition 29**: An **object representational constraint property** is a RCP (see Definition 12) that is particularly tailored for and is verified by the representation of selected objects in a model.*

Similarly, **object behavioral constraint properties** (*OBCP*) are defined as follows:

***Definition 30**: An **object behavioral constraint property** is a BCP (see Definition 13) that is particularly tailored for and is verified by the parametrized behavior model used to describe and simulate object behavior in a model.*

Object constraint properties are also formalized by a constraint language. Different constraint languages can be used for the design of different object constraint properties, i.e., structural, representational or behavioral.

OSCP, ORCP and *PBCP* are moreover classified into *a-temporal* and *temporal*:

- A-temporal object structural constraint properties (AOSCP)
- Temporal object structural constraint properties (TOSCP)
- A-temporal object representational constraint properties (AORCP)
- Temporal object representational constraint properties (TORCP)
- A-temporal object behavioral constraint properties (AOBCP) and
- Temporal object behavioral constraint properties (AOBCP)

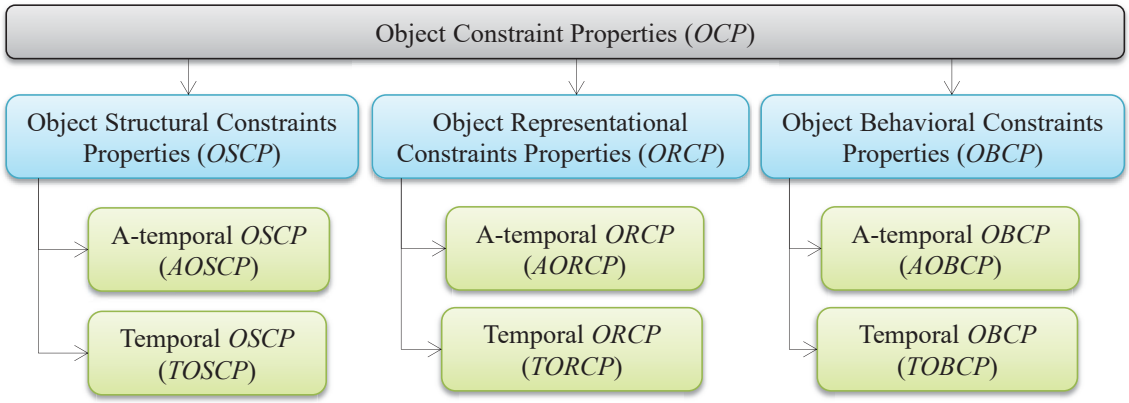


Figure 25. A classification of object constraint properties *OCP*.

An overview is shown in Figure 25. Definitions about the above quoted types of *OCP* are not provided since they correspond to the definitions of a-temporal and temporal *SCP*, *BCP* and *BCP* (see *Definition 14 – Definition 19*).

Several of the above quoted types of object constraint properties are illustrated based on the eFFBD model shown in Figure 24. For the rest of *MCP* that are not here illustrated, readers are encouraged to revisit CP_1 – CP_7 to get a general idea about the purpose of each type of property. First, the following *TOSCP* can be specified, considering the domain structure of the eFFBD DSML:

CP_{10}: "After detecting fire of flood, an acknowledgment about the situation must be provided within 1second"

The above quoted property must be formalized by an adequate constraint language before being verified. This property concerns the following objects shown in Figure 24: Detecting Fire, Detecting Flood, Warn Surveillance Center, Fire Detected, Flood Detected and Acknowledgement. The verification process takes place locally considering the above quoted objects and do not apply to other objects of the same model. Second, the following *TOBCP* can be specified, considering the behavioral model for the concept function shown in Figure 18:

CP_{11}: "The execution frequency of the fire detecting and flood detecting functions must be less that 100ms"

The verification process takes place as soon as the eFFBD model shown in Figure 24 is executed, based on the parametrizations of the behavioral model of functions (shown in Figure 18) for the objects fire detecting and flood detecting. A feedback is provided after or during the execution. For this type of properties, a model-checker must be integrated with a simulator.

Object constraint properties are formally defined as $OCP := \langle OSC, ORC, OBC \rangle$, where:

- *OSC* is the set of object structural constraint properties,
- *ORC* is the set of object representational constraint properties, and,
- *OBC* is the set of object behavioral constraint properties.

The formal specification of *OSC* is the very similar to *SC* with exception to the θ_{os} function. $OSC := \langle type, SCP, \theta_{scp}, \theta_{os} \rangle$, where (see the formal specification of *SC* for $type, SCP, \theta_{scp}$):

- $\theta_{os}: OSC \rightarrow O$ is the function that associates an object structural constraint property to objects that belong to the structures of a model.

In the same sense, the formal specification of *ORC* is the very similar to *RC* with exception to the θ_{or} function. $ORC := \langle type, RCP, \theta_{rcp}, \theta_{or} \rangle$, where (see the formal specification of *RC* for $type, RCP, \theta_{rcp}$):

- $\theta_{or}: ORC \rightarrow MRP$ is the function that associates an object representational constraint property to the representation of objects that belong to the representation of a model.

The formal specification of *OBC* is the very similar to *BC* with exception to the θ_{ob} function. $OBC := \langle type, BCP, \theta_{bcp}, \theta_{ob} \rangle$, where (see the formal specification of *BC* for $type, BCP, \theta_{bcp}$):

- $\theta_{mb}: MBC \rightarrow PBM$ is the function that associates an object behavioral constraint property to the parametrizations of a behavioral model based on objects that belong to a model .

A *Model* is formalized as a 5-uplet composed of modeling and system properties that concretize domain knowledge. Among the modeling properties that concretize domain knowledge are:

- model structural properties (*MSP*),
- model representational properties (*MRP*) and
- model behavioral properties (*MBP*)

The system properties that concretize domain knowledge are formalized through constraint properties that are verified based on preselected models, denoted

- model constraint properties (*MCP*)

or preselected objects in a model, denoted

- object constraint properties (*OCP*).

A *Model* is then formally defined as follows:

$$Model := \langle MSP, MRP, MBP, MCP, OCP \rangle$$

3.1.3 Synthesis

A synthesis of the typology of modeling and system properties is given in Table 5 recalling these properties are structured into two categories 1) properties that conceptualize domain knowledge, and 2) properties that concretize domain knowledge.

The first category of properties concerns the conceptual language (DSML) level, involving solely modeling properties. At DSML level the following modeling properties can be specified: structural (SP) that conceptualize the domain structure through concepts and relations, representational (RP) that conceptualize the representation of domain concepts and relations, behavioral (BP) that conceptualize the behavior of domain concepts and constraint properties (CP) that define additional information that cannot be implicitly defined by SP, RP or BP. Depending on whether CP are defined for the SP, RP or BP, we define structural constraints (SCP), behavioral constraints (BCP) and representational constraints (RCP). In addition, all types of CP are time or execution dependent or independent, restructuring them furthermore into temporal and a-temporal SCP, RCP and BCP, i.e., ASCP, TSCP, ARCP, TRCP, ABCP and TBCP.

The second category of properties (i.e., properties that concretize domain knowledge) concerns the model and object levels. At model level, both modeling and system properties are specified. At this stage modeling properties are structured into: model structural properties (MSP) that define the structure of a model, model representational properties (MRP) that define the representation of a model and model behavioral properties (MBP) that are used to parametrize behavioral models for the objects in a model for the purpose of model execution. System properties (i.e., the systems' requirements and the stakeholders' requirements) are specified through constraint properties at model level as model constraint properties (MCP) and at object level as object constraint properties (OCP). MCP and OCP are defined based on the structure (SP), representation (RP) or behavior (BP) of a DSML. However in constraint to CP that are verified based on any model, MCP and OCP are verified locally based on preselected models and preselected objects in a model. Depending on whether they are defined for the SP, RP or BP, we define model and object structural constraints (MSCP and OSCP), model and object representational constraints (MRCP and ORCP) and model and object behavioral constraints (MBCP and OBCP). In addition, all types of MCP and OCP are time or execution dependent or independent, restructuring them furthermore into temporal and a-temporal.

Table 5. A synthesis of modeling and system properties. (A – a-temporal, T – temporal, ML – modeling language, Ist. – illustration)

	Purpose	**Level**	**Classification**			**Def.**	**A**	**T**	**ML**	**Ist.**
Modeling properties	*Conceptualizing domain knowledge*	*DSML level*	*SP*			*Definition 7*	*x*		*Metamodeling language*	*Figure 16*
			RP			*Definition 8*	*x*		*Concrete syntax language*	*Figure 17*
			BP			*Definition 9*		*x*	*Behavioral language*	*Figure 18*
			CP	*SCP*	*ASCP*	*Definition 10 – Definition 19*	*x*	*x*	*Constraint modeling language*	*CP1 – CP7*
					TSCP					
				RCP	*ARCP*					
					TRCP					
				BCP	*ABCP*					
					TBCP					
	Concretizing domain knowledge	*Model level*	*MSP*			*Definition 20*	*x*		*Abstract syntax (SP)*	*Figure 20*
			MRP			*Definition 21*	*x*		*Concrete syntax (RP)*	*Figure 20*
			MBP			*Definition 22*		*x*	*Dynamic semantics (BP)*	*Figure 21*
System properties			*MCP*	*MSCP*	*AMSCP*	*Definition 23 - Definition 26*	*x*	*x*	*Constraint modeling language*	*CP8, CP9*
					TMSCP					
				MRCP	*AMRCP*					
					TMRCP					
				MBCP	*AMBCP*					
					TMBCP					
		Object level	*OCP*	*OSCP*	*AOSCP*	*Definition 27 - Definition 30*	*x*	*x*	*Constraint modeling language*	*CP10, CP11*
					TOSCP					
				ORCP	*AORCP*					
					TORCP					
				OBCP	*AOBCP*					
					TOBCP					

3.2 Property management: a DSML and Model lifecycle

The management of different type of properties is defined through a formalized lifecycle denoted “DSML and model lifecycle”. The lifecycle is composed of several phases and sub-phases. Each phase highlights the types of properties that need to be designed, the languages that need to be used and the V&V analyses that need to be performed.

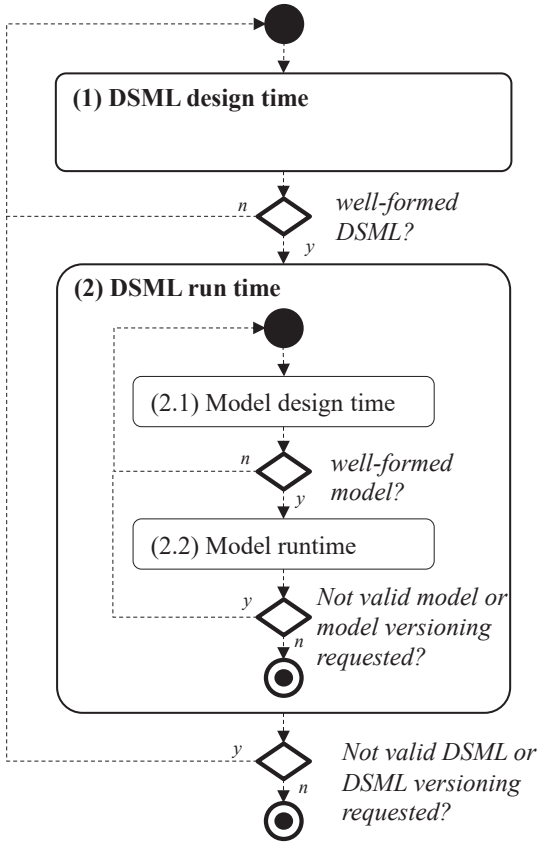


Figure 26. DSML and Model lifecycle phases.

DSML and model lifecycle is illustrated in Figure 26 composed of two major phases: *(1) “DSML design time”* and *(2) “DSML run time”*. In phase (1) DSML designers conceptualize domain knowledge by creating and verifying a DSML before putting it in use for the concretization of domain knowledge by model designers and users in phase (2). The phase (2) is decomposed into two sub-phases: (2.1) “Model design time”, for the design and verification of models and (2.2) “Model run time”, for model simulation, animation, and verification.

3.2.1 DSML design time

During the phase of DSML design time, a DSML is formalizes as a 4-uplet of modeling properties that conceptualize domain knowledge, denoted:

$$DSML := \langle SP, RP, BP, CP \rangle$$

Each set of modeling properties formalize different part of a DSML (see Section 0 for more details on the parts of a DSML):

- Structural properties (*SP*) formalize the DSML's abstract syntax
- Representational properties (*RP*) formalize the DSML's concrete syntax
- Behavioral properties (*BP*) formalize the DSML's dynamic semantics
- Constraint properties (*CP*) formalize the DSML's static semantics

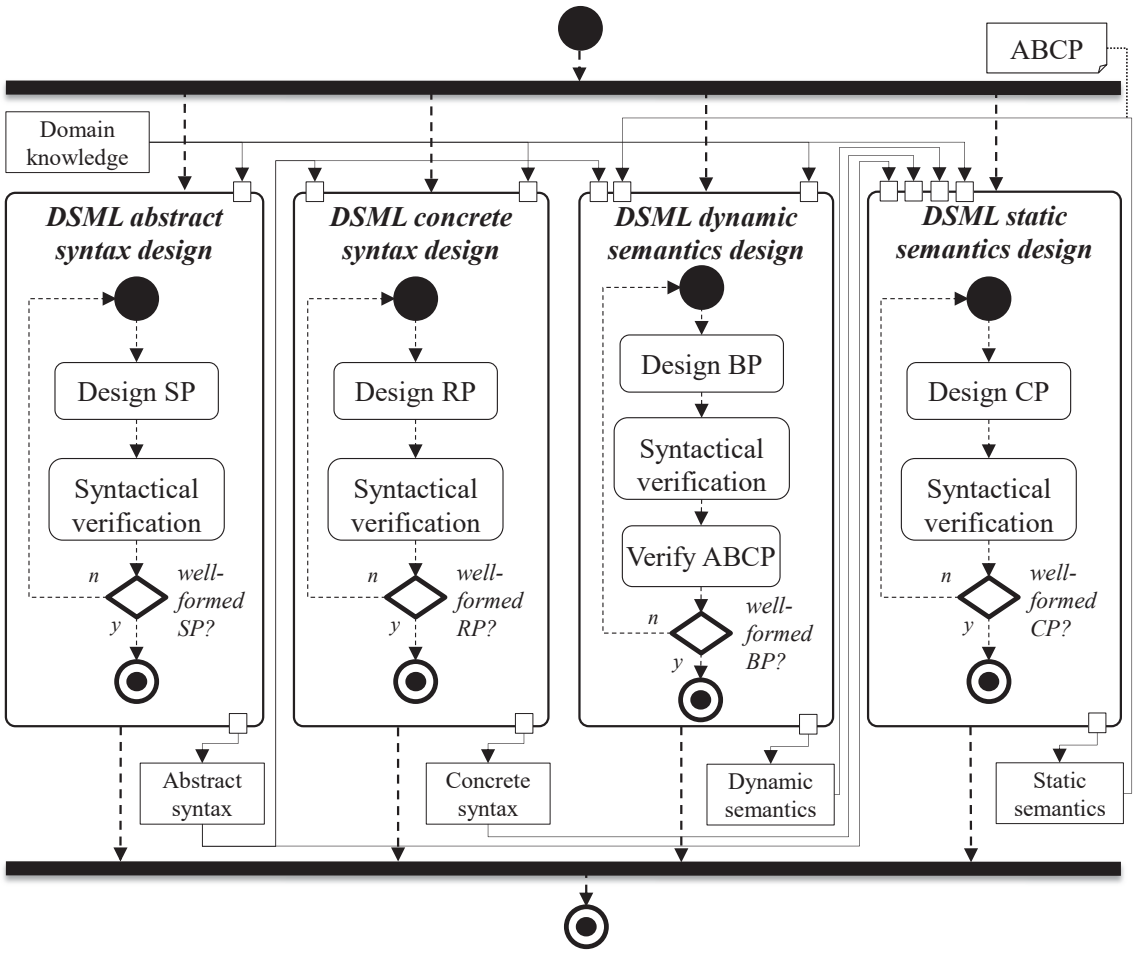


Figure 27. The DSML design time phase.

Figure 27 shows the DSML design time phase composed of four phases that are performed in parallel:

- DSML abstract syntax design

- DSML concrete syntax design
- DSML dynamic semantics design
- DSML static semantics design

During the phase of *DSML abstract syntax design*, a metamodeling language is used to formalize an abstract syntax based on the stakeholders' domain knowledge. The goal is to conceptualize the concepts of a domain and the relations that bound together the concepts, forming the set of structural properties *SP* through a metamodel. This phase is finalized by a syntactical verification to ensure the well-formedness of the designed abstract syntax. A well-formed abstract syntax respects the rules imposed by the used metamodeling language, i.e., conforms to the meta-metamodel of the used metamodeling language.

In parallel, the phases of *DSML concrete syntax design* and *DSML dynamic semantics design* take place based on the stakeholders' domain knowledge and on the abstract syntax that is in phase of design and thus partially provided (considering only the concepts and relations that are already defined). The goal is to formalize, on the one hand, the representations of domain concepts and relations, forming the set of representational properties *RP* by using a concrete syntax language, and on the other hand, the behavior of domain concepts, forming the set of behavioral properties *BP* by using a behavioral language. Finally, it must be verified that:

- *The concrete syntax is well-formed*: a concrete syntax must be syntactically verified to ensure that it respects the rules imposed by the used concrete syntax language. For instance, if the concrete syntax is defined by a model (i.e., a concrete syntax model), then this model should conform to the metamodel of the used concrete syntax language as proposed by the Diagraph approach (Pfister et al. 2014).
- *The dynamic semantics is well-formed*: the behavior is defined as a set of behavioral models. Each behavioral model must be:
    1. Syntactically verified to ensure the respect of syntax imposed by the used behavioral language, i.e., behavioral models must conform to the metamodel of the behavioral language. For instance, eISM behavioral models must conform to the eISM metamodel (discussed in Chapter IV).

2. Verify the behavioral language hypotheses (discussed above) defined through the *ABCP*.
3. Verify the alternative "stakeholders' hypotheses" (discussed above) also defined through the *ABCP*.

Finally, during the phase of DSML static semantics design, a constraint modeling language is used to formalize a static semantics based on the stakeholders' domain knowledge, and on the abstract syntax, concrete syntax and dynamic semantics that are in phase of design and thus partially provided. The goal is to formalize additional information that cannot be implicitly defined by the abstract syntax, concrete syntax or dynamic semantics, forming the set of constraint properties *CP*. This phase is finalized by a syntactical verification to ensure the well-formedness of the designed static semantics. A well-formed static semantics is composed of constraint properties that respect the rules imposed by the used constraint modeling language.

The result of the DSML design time phase is a well-formed DSML composed of:

- Well-formed abstract syntax
- Well-formed concrete syntax
- Well-formed dynamic semantics
- Well-formed static semantics

Such DSML is then provided as input to the second phase of "DSML rune time".

3.2.2 DSML run time / Model design time

During the first sub-phase of "Model design time", model designers use DSMLs to create models. A model is formalized as a 5-uplet of modeling and system properties that concretize domain knowledge, denoted:

$$Model := \langle MSP, MRP, MBP, MCP, OCP \rangle$$

The modeling properties formalize different part of a Model through the following sets of properties:

- Model structural properties (*MSP*) formalize the model's structure
- Model representational properties (*MRP*) formalize the model's representation
- Model behavioral properties (*MBP*) formalize the model's behavior

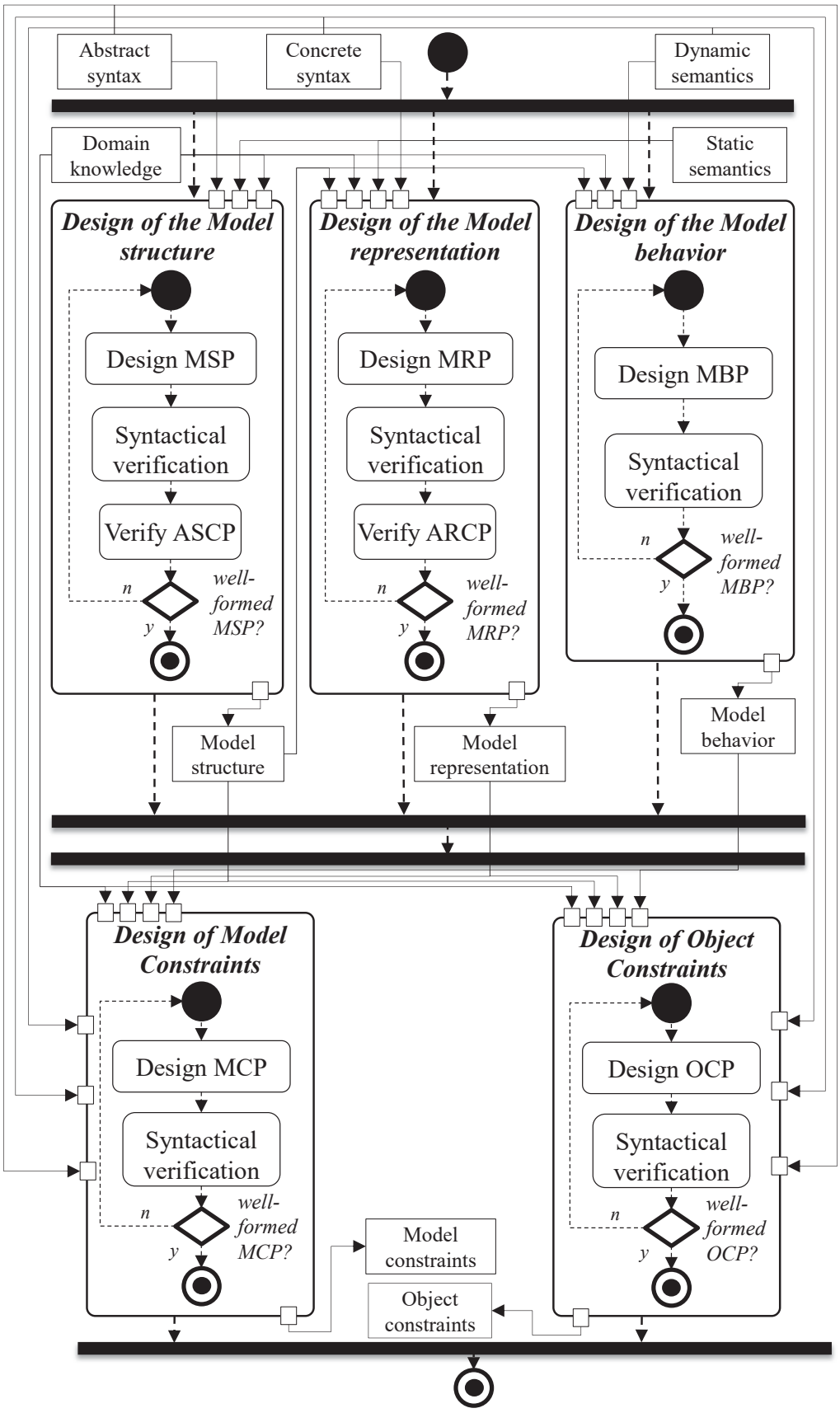


Figure 28. The DSML run time / Model design time phase.

The system properties formalize stakeholders' and systems' requirements through two sets of constraint properties:

- Model constraint properties (*MCP*), verified locally based on selecting models
- Object constraint properties (*OCP*), verified locally based on selecting objects

Figure 28 shows the DSML run time / Model design time phase composed of five phases, among which the following three are performed in parallel:

- Design of model structure
- Design of model representation
- Design of model behavior

Followed by two phases that are also performed in parallel:

- Design of model constraints
- Design of object constraints

During the first three parallel phases, the previously designed and well-formed DSML is used to design a model composed of a structure, a representation and behavior based on the stakeholders' domain knowledge, and on the DSML's abstract syntax, concrete syntax and dynamic semantics. The resulting model must furthermore be verified for well-formedness. The verification process consists in verifying the well-formedness of the model's structure, representation and behavior.

A model is well-formed if:

- *The model's structure is well-formed*. For this purpose two verification processes are performed:
    1. A syntactical verification: the model's structure must conform to the metamodel, i.e., to respect the rules imposed by the structural properties *SP*. Section 2.1.2 provides more details on the conformity relation.
    2. Verification of *ASCP*: the model's structure must be checked by a model checker or interpreter to determine if it respects the *ASCP*.
- The model's representation is well-formed:
    1. A syntactical verification: the model's representation must respect the rules imposed by the representational information (*RI*) of a concrete syntax.

2. Verification of *ARCP*: the model's structure must be checked by a model checker or interpreter to determine if it respects the *ARCP*.

- The model's behavior is well-formed:
    1. A syntactical verification: the behavior of a model is correctly parametrized considering the *BP*.

During the following two parallel phases: design of model / object constraint properties, system properties are formalized using a constraint modeling language based on the stakeholders' knowledge, the designed model (its structure, representation and behavior) and the used DSML (its abstract syntax, concrete syntax and dynamic semantics). *MCP* and *OCP* complement the static semantics by constraint properties and are verified locally based on preselected models or objects (see Section 3.1.2 for more details). This phase is finalized by a syntactical verification to ensure the well-formedness of the designed *MCP* and *OCP*, i.e., to ensure that they respect the rules imposed by the used constraint modeling language.

The result of the Model design time sub-phase is a well-formed model composed of:

- Well-formed structure
- Well-formed representation
- Well-formed behavior
- Well-formed model constraints
- Well-formed object constraints

Such model is then provided as input to the second sub-phase of "Model rune time" illustrated in Figure 29.

3.2.3 DSML run time / Model run time

During this sub-phase, models are executed, animated and used as a base for formal proof, to assure that they represent as accurately as possible a SoI.

Before preceding any V&V analyses, models must verify the *AMCP* as well as the *AOCP* (of course, if any of these properties are applied to the considered model or the objects contained in this model). For instance, the *AMSCP* CP_8: "*All functions must be performed infinitely (without an end) and in parallel*" must be verified by the functional architecture of the fire and flood detection system illustrated in Figure 24. A verification mechanism is furthermore proposed in Chapter V.

The V&V analyses consist here of simulation (i.e., model execution), model animation and formal properties proof based on models (i.e., verification of the requirements):

- The model execution is based on a gradual computation of the execution rules specified by the behavioral properties (i.e., the parameterized behavioral models *MBP*).
- The model animation is a result to the systematic visualization of changes (i.e., systematic modification of *MRP*) driven by the model execution according to the DSML's *RP*.
- Formal proof consists of formal verification of the temporal constraint properties (*TSCP, TMSCP, TOSCP, TRCP, TMRCP, TORCP, TBCP, TMBCP* and *TMBCP*).

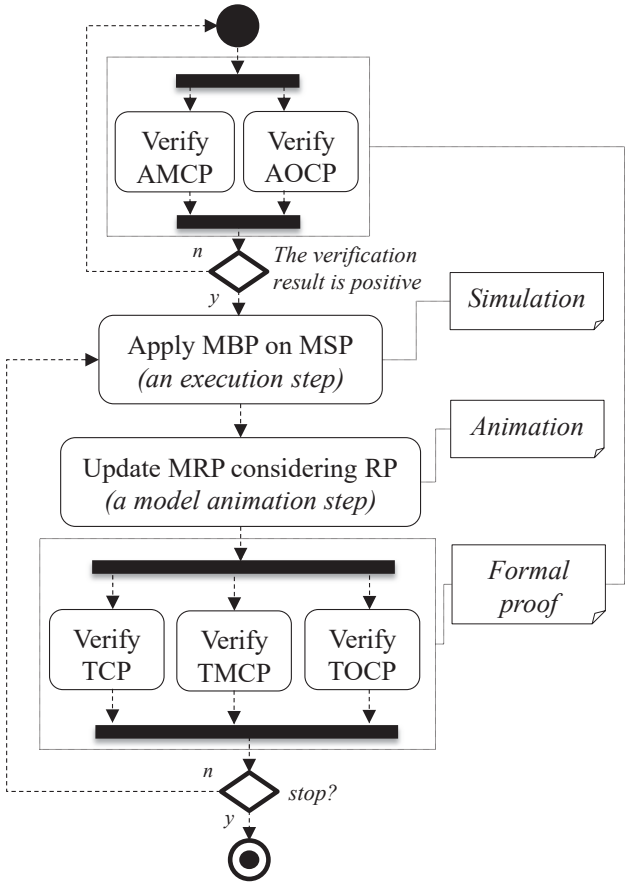


Figure 29. The DSML run time / Model run time phase

3.2.4 Synthesis

The whole DSML and model lifecycle is synthetized in Table 6. The figure highlights for each phases and sub-phased of the lifecycle the properties that need to be modeled,

the expected V&V analyses considering the designed properties and the expected results of these V&V analyses.

Table 6. Synthesis of the DSML and model lifecycle.

Phase		***Properties***	***V&V analyses***	***V&V result***	
DSML design time		*SP*	*Syntactical analysis of SP*	*Well-formed abstract syntax*	*Well-formed DSML*
		RP	*Syntactical analysis of RP*	*Well-formed concrete syntax*	
		BP	*Syntactical analysis of BP: Verification of ABCP*	*Well-formed dynamic semantics*	
		CP	*Syntactical analysis of CP*	*Well-formed static semantics*	
DSML run time	*Model design time*	*MSP*	*Syntactical analysis of MSP: Verification of ASCP*	*Well-formed model structure*	*Well-formed model*
		MRP	*Syntactical analysis of MRP; Verification of ARCP*	*Well-formed model representation*	
		MBP	*Syntactical analysis of MBP;*	*Well-formed model behavior*	
		MCP	*Syntactical analysis of MCP*	*Well-formed model constraints*	
		OCP	*Syntactical analysis of OCP*	*Well-formed object constraints*	
	Model run time		*Verification of AMCP and AOCP; Simulation; Animation; Verification of TCP, TMCP and TOCP*	*Valid (as much as possible) model*	

During the phase of DSML design time a DSML is designed based on four different types of properties: structural properties (SP), representational properties (RP), behavioral properties (BP) and constraint properties (CP). Before proceeding to the next phase, each of the properties must be verified for well-formedness, mainly based on a syntactical analysis, except for the BP that must also verify the ABCP (i.e., the behavioral language hypotheses and stakeholders' hypotheses).

As a result, a well-formed DSML is used to design model during the DSML run time / Model design time (sub) phase. A model is designed based on the model structural properties (MSP), model representation properties (MRP) and model behavioral properties (MBP), along with model and object constraint properties (MCP and OCP) that aim at formalizing stakeholders' and systems' requirements. Before proceeding to the next phase, each of the model properties must be verified for well-formedness. This includes syntactical verification of all model's properties, but also a verification of the previously specified a-temporal structural and representational constraint properties based on the model's structure and representation. As a result, a well-formed model is provided and used in the next sub-phase of Model run time. During the Model run time, a model is first verified considering the AMCP and AOCP, before being used for simulation and animation. Temporal properties (TCP, TMCP and TOCP) are verified during simulation. The goal of this phase is to validate models as much as possible, allowing stakeholders to detect and eliminate design errors and mistakes early during the phase of design and to support them to make decisions with confidence.

3.3 A multi-viewpoint modeling based on properties

This section presents a multi-viewpoint modeling based on the typology of properties introduced in Section 3 and the DSML and model lifecycle introduced in 3.2. The key concepts of the multi-viewpoint modeling are *composite DSML* (discussed in Section 3.3.1) and a *composite Model* (discussed in Section 3.3.2). Section 3.3.3 introduces a modified version of the DSML and model lifecycle for the design and management of *composite DSML* and *composite model*, denoted *composite DSML and Model lifecycle*.

This work does not take into consideration entirely the model interoperability problematic as difined for instance in (Tolk & Muguira 2003). The latter involves problems related for instance with the syntaxical or semantical compatibility between domain concepts and relations from different DSMLs. For the purpose of our work, we consider first that the modeling needs of each viewpoint are covered by using a unique DSML. Second, semantic ambiguities (e.g. domain concepts with same name, different names but same meaning, or different cardinamlity constraints) between these DSML are out of the scope of our work and are considered as clarified (e.g., there is no possible confusion between domain concepts from eFFBD and PBD).

So, a **composite DSML** is defined as follows:

Definition 31: A ***composite DSML*** is composed of several heterogeneous DSMLs, each one covering the modeling of a viewpoint (e.g., requirements, functional, logical, physical, etc.) of a system of interest (SoI).

Composite DSMLs are used to design multi-viewpoint SoI models called **composite models**.

Definition 32: A ***composite model*** is composed of several models that are conform to different DSMLs that take part in a composite DSML, allowing a more expressive, realistic and complete representation of a SoI.

3.3.1 Composite DSML

A composite DSML is formally defined as:

$$compDSML := \langle DS, DR, DB, CP, DCP \rangle$$

Where:

- *DS* is a composite structure, here-denoted dependent structure, putting in relation different abstract syntaxes from each composing DSML;
- *DR* is a composite representation, here-denoted dependent representation, putting in relation different concrete syntaxes from each composing DSML;
- *DB* is a composite behavior, here-denoted dependent behavior, putting in relation different dynamic semantics from each composing DSML;
- *CP* is the set of constraints properties (see *Definition 10*) that are specified based on one of the composing abstract syntaxes, concrete syntaxes or dynamic semantics.
- *DCP* is the set of dependent constraint properties that are specified based on the dependencies of the abstract syntaxes, concrete syntaxes or dynamic semantics.

A **dependent structure** (*DS*) is defined as follows:

Definition 33: A ***dependent structure*** of a composite DSML (see *Definition 31*) is the composition of several abstract syntaxes (see *Definition 7*) that belong to the composing DSMLs.

The composition process aims at relating structurally several abstract syntaxes, creating an overall composite abstract syntax, i.e., a dependent structure. The abstract syntaxes are defined by the DSMLs' structural properties (*SP*), as proposed above. So, the

composition process consists in defining structural dependency properties (*SDP*, formalized in the next) between Structural Properties (*SP*) from each of the composing DSMLs.

Structural properties (*SP*) and structural dependency properties (*SDP*) can be formalized by a metamodeling language such as for example the OMG's standard MOF (OMG 2015a) (see Section 2.1.2 for more details).

Figure 30 illustrates a metamodel that contains:

- the *SP* of the eFFBD language (note that, for the sake of simplicity, only the concept Function of the eFFBD language is shown in the figure)
- the *SP* of the PBD language: Component, Interface, Link and Context
- the *SDP* that bound together the PBD and the eFFBD: the relations *functions* and *performs* between the concepts Function and Component.

The physical components of a system perform one or more functions and functions are allocated to a component. The input, output, and triggers flows of allocated functions are themselves allocated to a Link devoted then to carry out these flows from external source (Context) or from an existing Component.

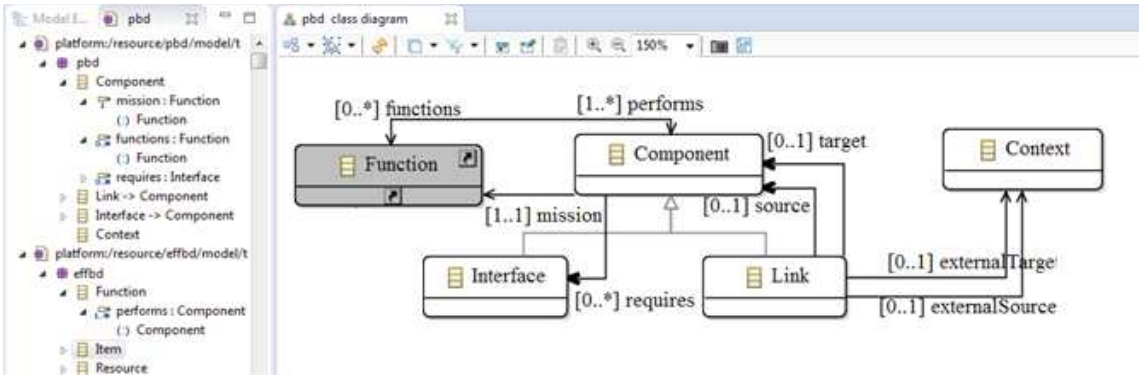


Figure 30. A dependent structure (*DS*) composing SP of the eFFBD and the PBD (designed by the EMF, the eFFBD metamodel is "loaded" into the PBD metamodel).

A dependent structure (*DS*) of a composite DSML is then formally defined as:

$DS := \langle CSP, CSD \rangle$, where:

- *CSP* = {SP_i / SP_i is the set of structural properties from i^{th} composing DSML that take part in a composite DSML}. The formal definition of SP_i is shown in Section 3.1.1.

- $CSD = \{(SP_i, SP_j, SDP) / \forall i \in |CSP|, \forall j \in |CSP|, i <> j \}$ defines the structural dependencies between two structural properties of each DSML composing the composite DSML, where:
    i. SP_i is the first set of structural properties
    ii. SP_j is the second set of structural properties
    iii. SDP is the set of structural properties that define the dependencies between SP_i and SP_j, denoted "structural dependency properties". The formal definition of a structural property is shown in Section 3.1.1.

A **dependent representation** (*DR*) is defined as follows:

> ***Definition 34:*** A ***dependent representation*** of a composite DSML (see *Definition 31*) is the composition of several concrete syntaxes (see *Definition 8*) that belong to the composing DSMLs.

The composition process aims at relating several concrete syntaxes creating an overall navigable composite concrete syntax, i.e., a dependent representation. The concrete syntaxes are defined by the DSMLs' representational properties (*RP*), as proposed above. So, the composition process consists in defining representational dependency properties (*RDP*, formalized in the next) between Representational Properties (*RP*) from each of the composing DSMLs.

To illustrate, Figure 31 shows the graphical representation of the PBD language and the eFFBD language (see also Figure 17 for the RP of the eFFBD) as well as the dependency between these two representations. Note that the representations are only schematized and must furthermore be formalized by an adequate concrete syntax language that supports multi-view representation, such as Diagraph (Pfister et al. 2014) or Obeo Designer (Juliot & Benois 2010).

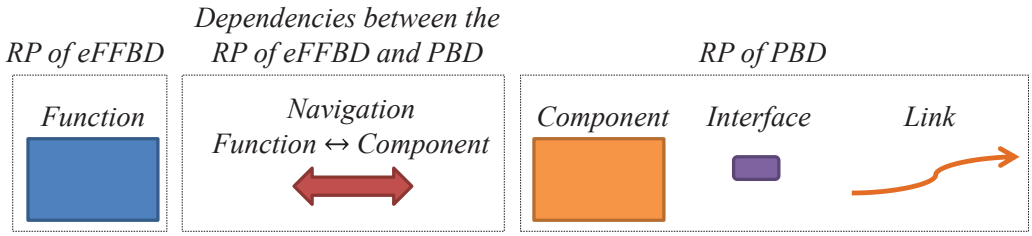


Figure 31. Combining two graphical RP (for the PBD and for the eFFBD) into a DR.

A dependent representation (*DR*) of a composite DSML is formally defined as:

$DR := \langle CRP, CRD \rangle$, where:

- CRP = {RP_i / RP_i is the set of representational properties from i^{th} composing DSML that take part in a composite DSML}. The formal definition of RP_i is shown in Section 3.1.1.
- CRD = {(RP_i, RP_j, RDP) / $\forall i \in |CRP|$, $\forall j \in |CRP|$, $i <> j$ } defines the representational dependencies between two representational properties of each DSML composing the composite DSML, where:
    i. RP_i is the first set of representational properties
    ii. RP_j is the second set of representational properties
    iii. RDP is the set of representational properties that define the dependencies between RP_i and RP_j, denoted "representational dependency properties". The formal definition of a representational property is shown in Section 3.1.1.

A dependent behavior (*DB*) is defined as follows:

> ***Definition 35:*** A ***dependent behavior*** of a composite DSML (see *Definition 31*) is a composition of several dynamic semantics (see *Definition 9*) that belong to the composing DSMLs.

The composition process aims at relating several dynamic semantics (e.g., by including the specification of data and event exchanges) creating an overall centralized composite dynamic semantics, i.e., a dependent behavior. The dynamic semantics are defined by the DSMLs' Behavioral Properties (*BP*), as proposed above. So, the composition process consists in defining behavioral dependency properties (*BDP*, formalized in the next) between Behavioral Properties (*RP*) from each of the composing DSMLs.

The illustration consists in integrating the behavior of the concept Function of the eFFBD shown in Figure 18 and the behavior of the concept Component of the PBD described hereafter and shown in Figure 32 as a 5-state discrete-events model.

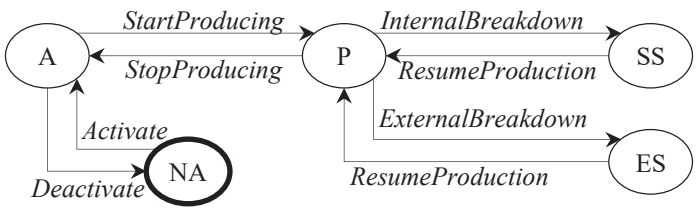


Figure 32. The *BP* for the concept Component of PBD.

Components are initially non-active ($NA:S_1$) waiting for the initial signal to prepare for production ($A:S_2$). When the signal arrives they start producing ($P:S_3$) by performing their functions (i.e., functions from the functional viewpoint should start execution), until they receive, either a stop signal, which put them in the previous state, or an internal or an external breakdown signal, which immediately makes them stop producing and puts them in maintenance states ($SS:S_4$ *or* $ES:S_5$), suspending also the execution of their functions (i.e., functions from the functional viewpoint enter suspended state).

For the purpose of simulation (i.e., model execution) the behavioral models of the concepts Function and Component must be coordinately used. This leverages the need for a synchronization mechanism allowing data and event exchanges between different behavioral models. Chapter IV introduces such mechanism for coordinated simulation based on the blackboard design pattern.

A dependent behavior (*DB*) of a composite DSML is formally defined as:

$$DB := \langle CBP, CBD \rangle, \text{ where:}$$

- CBP = {BP_i / BP_i is the set of behavioral properties from ith composing DSML that take part in a composite DSML}. The formal definition of BP_i is shown in Section 3.1.1.
- CBD = {(BP_i, BP_j, BDP) / $\forall i \in |CBP|$, $\forall j \in |CBP|$, i<>j } defines the behavioral dependencies between two behavioral properties of each DSML composing the composite DSML, where:
    iv. BP_i is the first set of behavioral properties
    v. BP_j is the second set of behavioral properties
    vi. BDP is the set of behavioral properties that define the dependencies between BP_i and BP_j, denoted “behavioral dependency properties”. The formal definition of a behavioral property is shown in Section 3.1.1.

Constraint properties (CP) are introduced and formally defined in Section 3.1.1.

Dependent constraint properties (*DCP*) are defined as follows:

> ***Definition 36:*** A ***dependent constraint property*** is a constraint property (see *Definition 10*) that expresses complementary characteristics that cannot be

implicitly defined by the dependencies in a composite DSML (see *Definition 31*).

Depending on the concerned type of dependencies, dependent constraint properties are classified into:

- dependent structural constraint properties (DSCP),
- dependent representational constraint properties (DRCP) and
- dependent behavioral constraint properties (DBCP)

Dependent structural constraint properties (*DSCP*) are defined as follows:

> ***Definition 37**: A **dependent structural constraint property*** is a structural constraint property (see *Definition 11*) that expresses complementary characteristics that cannot be implicitly defined by the structural dependencies (SDP) in a dependent structure (see *Definition 33*) of a composite DSML (see *Definition 31*).

In this sense, dependent representational constraint properties (*DRCP*) are defined as follows:

> ***Definition 38**: A **dependent representational constraint property*** is a representational constraint property (see *Definition 12*) that expresses complementary characteristics that cannot be implicitly defined by the representational dependencies (RDP) in a dependent representation (see *Definition 34*) of a composite DSML (see *Definition 31*).

Similarly, **dependent behavioral constraint properties** (*DBCP*) are defined as follows:

> ***Definition 39**: A **dependent behavioral constraint property*** is a behavioral constraint property (see *Definition 13*) that expresses complementary characteristics that cannot be implicitly defined by the behavioral dependencies in a dependent behavior (see *Definition 35*) of a composite DSML (see *Definition 31*).

Dependent constraint properties are formalized by a constraint language. Different constraint languages can be used for the design of different dependent constraint properties, i.e., structural, representational or behavioral.

DSCP, *DRCP* and *DBCP* are moreover classified into *a-temporal* and *temporal*:

- A-temporal dependent structural constraint properties (ADSCP)
- Temporal dependent structural constraint properties (TDSCP)
- A-temporal dependent representational constraint properties (ADRCP)
- Temporal dependent representational constraint properties (TDRCP)
- A-temporal dependent behavioral constraint properties (ADBCP) and
- Temporal dependent behavioral constraint properties (ADBCP)

An overview is shown in Figure 33.

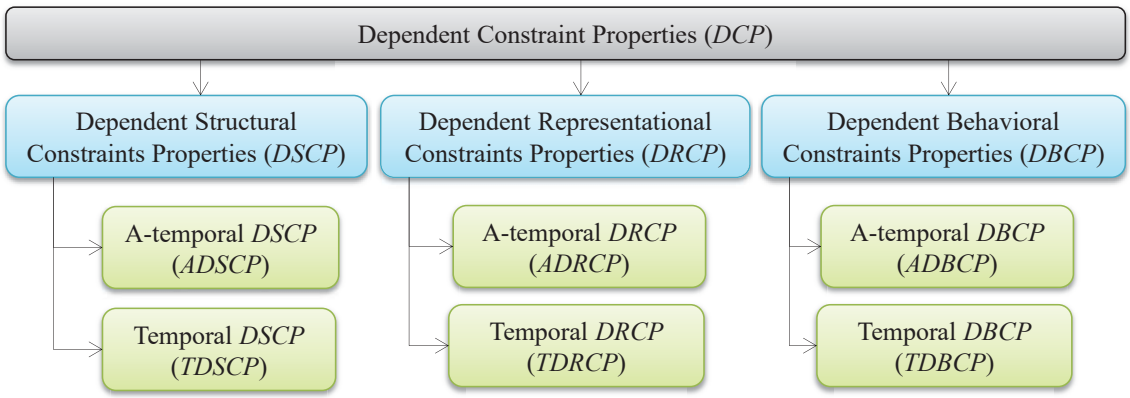


Figure 33. A classification of dependent constraint properties *DCP*.

Definitions about the above quoted types of *DCP* are not provided since they correspond to the definitions of a-temporal and temporal *SCP*, *BCP* and *BCP* (see *Definition 14 – Definition 19*).

To illustrate the DCP, a ADSCP and a TDBCP are proposed hereafter.

The ADSCP is based on the structural dependencies between the SP of the eFFBD and the PBD shown in Figure 30:

CP_{12}: "If a component C has a mission function,
then this function is allocated and performed by C"

The above quoted property must be furthermore formalized by an adequate constraint language before being verified. The verification process takes place as soon as a composite model is designed. The feedback of the verification process is either positive (i.e., the model respects the property) or negative (i.e., the model violates the property and thus must be revisited for corrections).

The TDBCP is based on the behavioral dependencies between the BP of the eFFBD and the PBD:

> CP_{13}: "It is always true (each execution step) that if a component is in a breakdown state, its functions must suspend execution (i.e., must enter suspended state the next execution step)"

The above quoted property must be formalized by an adequate constraint language before being verified. The verification process takes place as soon as a composite model is executed. A feedback is provided after or during the model execution. For this type of properties, a model-checker must be integrated with a simulator.

Dependent constraint properties are formally defined as:

$$DCP := \langle DSC, DRC, DBC \rangle, \text{ where}$$

- *DSC* is the set of dependent structural constraint properties,
- *DRC* is the set of dependent representational constraint properties and,
- *DBC* is the set of dependent behavioral constraint properties.

The formal specification of *DSC, DRC* and *DBC* is the very similar respectively to *SC, RC* and *BC* (see Section 3.1.1) with exception to the θ_{scp}, θ_{rcp} and θ_{bcp} functions:

- $\theta_{scp}: SCP \rightarrow DSP$ associates a dependent structural constraint property to a structural dependency between different viewpoint structures.
- $\theta_{rcp}: SCP \rightarrow DRP$ associates a dependent representational constraint property to a representational dependency between different viewpoint representations.
- $\theta_{bcp}: BCP \rightarrow DBP$ associates a dependent behavioral constraint property to a behavioral dependency between dependent behaviors.

3.3.2 Composite Model

A composite Model (see *Definition 32*) is formally defined as:

$$compModel := \langle DMS, DMR, DMB, MCP, OCP, DMCP, DOCP \rangle$$

Where:

- *DMS* is a composite model structure, here-denoted dependent model structure putting in relation different structures from each composing model;

- *DMR* is a composite model representation, here-denoted dependent model representation putting in relation different parametrized representations from each composing model;
- DMB is a composite model behavior, here-denoted dependent model behavior putting in relation different parametrized behavioral models from each composing model;
- *MCP* and *OCP* are sets of Model Constraint Properties (see *Definition 23*) and Object Constraint Properties (see *Definition 27*) that are specified based on one of the composing models.
- *DMCP* and *DOCP* are sets of Dependent Model Constraint Properties (see *Definition 23*) and Dependent Object Constraint Properties that are specified based on the dependencies between model structures, representations and behaviors.

A **dependent model structure** (*DMS*) is defined as follows:

> ***Definition 40:*** A ***dependent model structure*** of a composite model (see *Definition 32*) is the composition of several model structures (see *Definition 20*) that belong to the composing models.

The composition process aims at relating structurally several model structures, creating an overall composite model structure, i.e., a dependent model structure. The model structures are defined by the model structural properties (*MSP*), as proposed above. So, the composition process consists in defining model structural dependency properties (*MSDP*, formalized in the next) between Model Structural Properties (*MSP*) from each of the composing models.

For illustrative purpose, a dependent model structure is shown in Figure 34, integrating the *MSP* of the eFFBD model shown in Figure 20 with the MSP of a PBD model. The *MSP* of the eFFBD model is shown on the left side of the figure, the *MSP* of the PBD is shown on the right side of the figure and dependencies are shown in the middle.

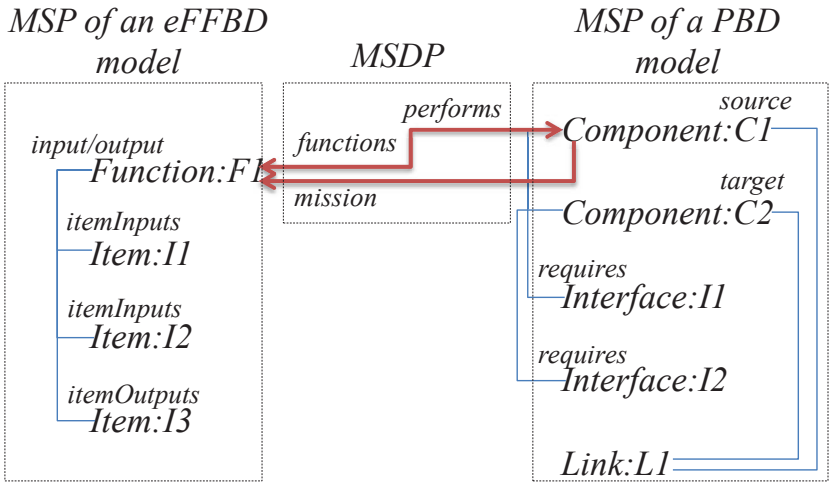


Figure 34. A dependent model structure (*DMS*) integrating the *MSP* of an eFFBD model, and the MSP of a PBD model.

A dependent model structure (*DMS*) of a composite model is formally defined as:

$$DMS := \langle CMSP, CMSD \rangle, \text{ where:}$$

- $CMSP$ = {MSP_i / MSP_i is the set of model structural properties from i^{th} composing model that take part in a composite model}. The formal definition of MSP_i is shown in Section 3.1.1.
- $CMSD$ = {(MSP_i, MSP_j, $MSDP$) / $\forall i \in |CMSP|$, $\forall j \in |CMSP|$, $i <> j$ } defines the model structural dependencies between two model structural properties of each composing model that take part in the composite model, where:
    i. MSP_i is the first set of model structural properties
    ii. MSP_j is the second set of model structural properties
    iii. $MSDP$ is the set of model structural properties that define the dependencies between MSP_i and MSP_j, denoted "model structural dependency properties". The formal definition of a model structural property is shown in Section 3.1.1.

A **dependent model representation** (*DMR*) is defined as follows:

> ***Definition 41:*** A ***dependent model representation*** of a composite model (see *Definition 32*) is the composition of several model representations (see *Definition 21*) that belong to the composing models.

The composition process aims at relating representationally several model representations, creating an overall navigable composite model rerepsentation, i.e., a dependent model rerepsentation. The model representations are defined by the model representational properties (*MRP*), as proposed above. So, the composition process consists in defining model representational dependency properties (*MRDP*, formalized in the next) between Model Representatinal Properties (*MRP*) from each of the composing models.

An example is shown in Figure 35 representing graphically the *DMS* shown in Figure 34, respecting the *DRP* shown in Figure 31. Navigation is possible from one viewpoint to another as shown in the figure.

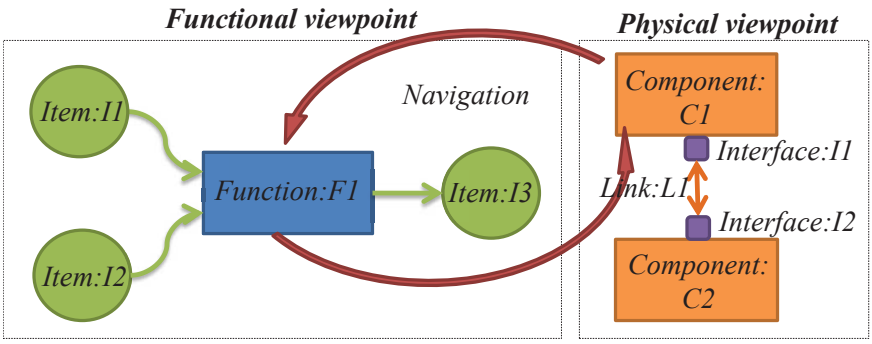


Figure 35. A DMR of the dependent model structure (*DMS*) shown in Figure 34.

A **dependent model representation** (*DMR*) is formally defined as:

$DMR := \langle CMRP, CMRD \rangle$, where:

- $CMRP$ = {MRP_i / MRP_i is the set of model representational properties from i[th] composing model that takes part in a composite model}. The formal definition of MRP_i is shown in Section 3.1.1.
- $CMRD$ = {(MRP_i, MRP_j, $MRDP$) / $\forall i \in |CMRP|$, $\forall j \in |CMRP|$, $i <> j$ } defines the model representational dependencies between two model representational properties of each composing model that takes part in the composite model, where:
    i. MRP_i is the first set of model representational properties
    ii. MSP_j is the second set of model representational properties
    iii. $MRDP$ is the set of model representational properties that define the dependencies between MRP_i and MRP_j, denoted "model

representational dependency properties". The formal definition of a model representional property is shown in Section 3.1.1.

A **dependent model behavior** (*DMB*) is defined as follows:

> ***Definition 42:*** A ***dependent model behavior*** of a composite model (see *Definition 32*) is the composition of several model behaviors (see *Definition 22*) that belong to the composing models.

The composition process aims at relating behaviorally several model behaviors, creating an overall composite model behavior, i.e., a dependent model behavior. The model behaviors are defined by the model behavioral properties (*MBP*), as proposed above. So, the composition process consists in defining model behavioral dependency properties (*MBDP*, formalized in the next) between Model Behavioral Properties (*MBP*) from each of the composing models.

As illustration, Figure 36 shows the DMB of the model shown in Figure 34, integrating the MBP of an eFFBD model (shown in Figure 22) and the MBP of a PBD model. The corresponding behavioral models:

- the state machine illustrate in Figure 18 for the concept Function,
- the state machine shown in Figure 21 for the concept Item and
- the state machine illustrated in Figure 32 for the concept Component

are parameterized for each object (i.e., for the function F1, the item I1, the item I2, the item I3, the component C1 and the component C2) as shown in Figure 36.

As previously discussed, for the purpose of simulation (i.e., model execution) behavioral models must be coordinately executed (i.e., the parametrized behavioral model of F1 must be coordinated with the parametrized behavioral models of I1, I2 and I3 from the same model, but also with the parametrized behavioral model of C1 from the PBD model). This leverages the need for a synchronization mechanism allowing data and event exchanges between different behavioral models. Chapter IV introduces such mechanism for coordinated simulation based on the blackboard design pattern.

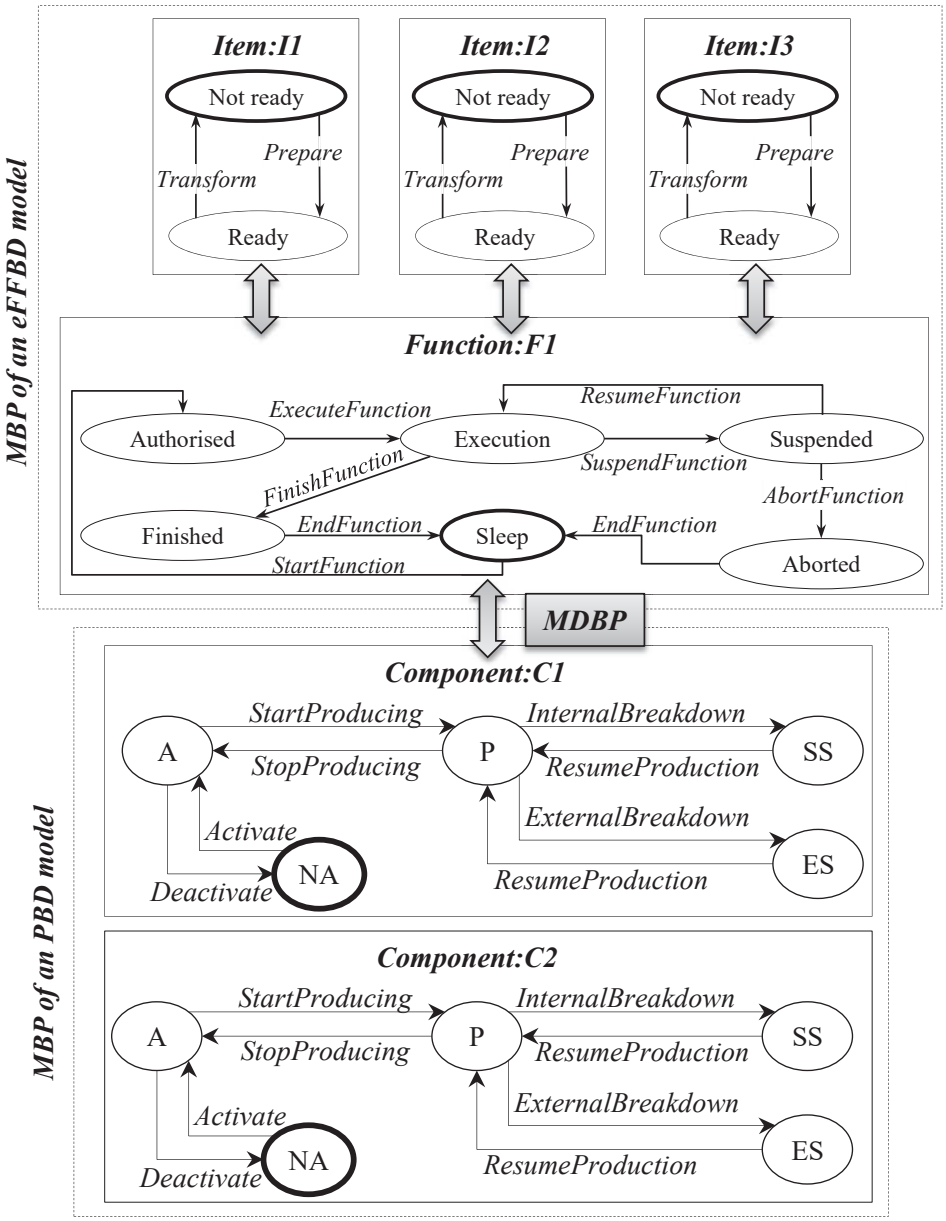


Figure 36. A DMB of the dependent model structure (DMS) shown in Figure 34.

A dependent model behavior (*DMB*) is formally defined as:

$DMB := \langle CMBP, CMBD \rangle$, where:

- *CMBP* = {MBP_i / MBP_i is the set of model behavioral properties from i^{th} composing model that take part in a composite model}. The formal definition of MBP_i is shown in Section 3.1.1.
- *CMBD* = {(MBP_i, MBP_j, *MBDP*) / ∀i∈|*CMBP*|, ∀j∈|*CMBP*|, i<>j } defines the model behavioral dependencies between two model behavioral properties of each composing model that take part in the composite model, where:
    iv. MBP_i is the first set of model behavioral properties

v. MBP_j is the second set of model behavioral properties

vi. $MBDP$ is the set of model behavioral properties that define the dependencies between MBP_i and MBP_j, denoted "model behavioral dependency properties". The formal definition of a model behavioral property is shown in Section 3.1.1.

Model constraint properties (*MCP*) and object constraint properties (*OCP*) are introduced and formally defined in Section 3.1.2.

Dependent model constraint properties (*DMCP*) are defined as follows:

> ***Definition 43:*** A ***dependent model constraint property*** is a model constraint property (see *Definition 23*) that expresses complementary characteristics that are verified locally based on the dependencies in a composite model (see *Definition 32*).

Dependent model constraint properties are classified into:

- dependent model structural constraint properties (DMSCP),
- dependent model representational constraint properties (DMRCP) and
- dependent model behavioral constraint properties (DMBCP)

Dependent model structural constraint properties (*DMSCP*) are defined as follows:

> ***Definition 44****: A* ***dependent model structural constraint property*** is a model structural constraint property (see *Definition 24*) that expresses complementary characteristics that are verified locally based on the model structural dependencies (MSDP) in a dependent model structure (see *Definition 40*).

In this sense, dependent model representational constraint properties (*DMRCP*) are defined as follows:

> ***Definition 45****: A* ***dependent model representational constraint property*** is a model representational constraint property (see *Definition 25*) that expresses complementary characteristics that are verified locally based on the model representational dependencies (MRDP) in a dependent model representation (see *Definition 41*).

Similarly, dependent model behavioral constraint properties (*DMBCP*) are defined as follows:

Definition 46**: A **dependent model behavioral constraint property is a model behavioral constraint property (see *Definition 26*) that expresses complementary characteristics that are verified locally based on the model behavioral dependencies (MBDP) in a dependent model behavior (see *Definition 42*).

Dependent model constraint properties are formalized by a constraint language. Different constraint languages can be used for the design of different dependent model constraint properties, i.e., structural, representational or behavioral.

DMSCP, *DMRCP* and *DMBCP* are moreover classified into *a-temporal* and *temporal*:

- A-temporal dependent model structural constraint properties (ADMSCP)
- Temporal dependent model structural constraint properties (TDMSCP)
- A-temporal dependent model representational constraint properties (ADMRCP)
- Temporal dependent model representational constraint properties (TDMRCP)
- A-temporal dependent model behavioral constraint properties (ADMBCP) and
- Temporal dependent model behavioral constraint properties (ADMBCP)

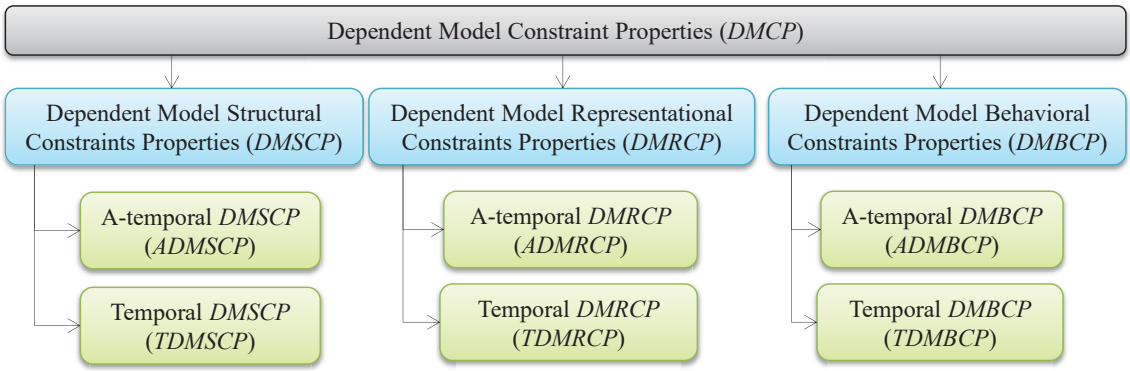


Figure 37. A classification of dependent model constraint properties *DMCP*

An overview is shown in Figure 37. Definitions about the above quoted types of *DMCP* are not provided since they correspond to the definitions of a-temporal and temporal *SCP*, *BCP* and *BCP* (see *Definition 14 – Definition 19*).

For illustrative purpose, the functional architecture of a fire and flood detection system shown in Figure 24 is integrated with a physical architecture specified through a PBD model. The integrated architecture is shown in Figure 38.

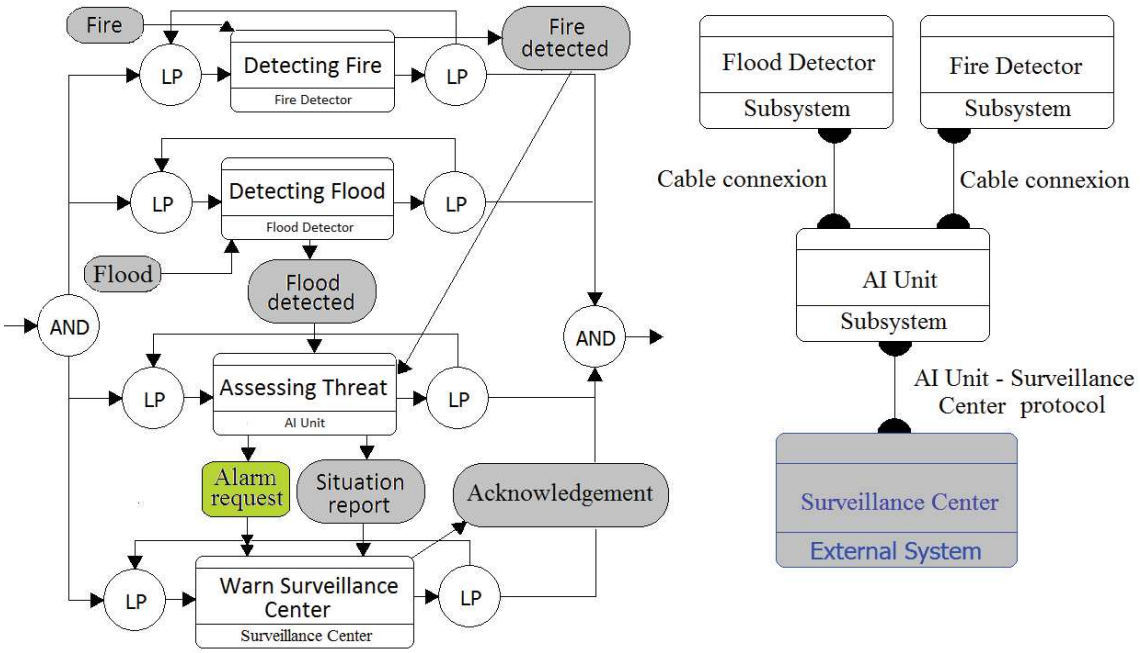


Figure 38. The architecture of a fire and flood security system, combining functional (left) and physical (right) models.

The physical architecture is composed of four main components: Flood Detector, Fire Detector, AI Unit and a Surveillance Center. Each of these components performs one function from the eFFBD model which is its mission function (e.g., the Flood Detector component performs the Flood Detecting function which is its mission function).

Based on the architecture, the following ADMSCP can be specified, considering the metamodel shown in Figure 30:

CP_{14}: "Each component performs one function which is its mission function"

The above quoted property must be furthermore formalized by a constraint modeling language. The verification process takes place locally for the functional and physical architecture shown in Figure 38. CP_{14} is not verified on any other model.

Dependent object constraint properties (*DOCP*) are defined as follows:

> ***Definition 47:*** A ***dependent object constraint property*** is an object constraint property (see *Definition 27*) that expresses complementary characteristics verified locally based on particular objects that take part in the dependencies of a composite model (see *Definition 32*).

Similarly to *DMCP*, dependent object constraint properties are classified into:

- dependent object structural constraint properties (DOSCP),
- dependent object representational constraint properties (DORCP) and
- dependent object behavioral constraint properties (DOBCP)

Dependent object structural constraint properties (*DOSCP*) are defined as follows:

> ***Definition 48**: A **dependent object structural constraint property*** is an object structural constraint property (see *Definition 28*) that expresses complementary characteristics verified locally based on particular objects that take part in the structural dependencies (MSDP) in a dependent model structure (see *Definition 40*).

In this sense, dependent object representational constraint properties (*DORCP*) are defined as follows:

> ***Definition 49**: A **dependent object representational constraint property*** is an object representational constraint property (see *Definition 29*) that expresses complementary characteristics verified locally based on the representation of particular objects that take part in the representational dependencies (MRDP) in a dependent model representation (see *Definition 41*).

Similarly, dependent object behavioral constraint properties (*DOBCP*) are defined as follows:

> ***Definition 50**: A **dependent object behavioral constraint property*** is an object behavioral constraint property (see *Definition 30*) that expresses complementary characteristics verified locally based on the behavior of particular objects that take part in the behavioral dependencies (MBDP) in a dependent model behavior (see *Definition 42*).

Dependent object constraint properties are formalized by a constraint language. Different constraint languages can be used for the design of different dependent object constraint properties, i.e., structural, representational or behavioral.

DOSCP, *DORCP* and *DOBCP* are moreover classified into *a-temporal* and *temporal*:

- A-temporal dependent object structural constraint properties (ADOSCP)
- Temporal dependent object structural constraint properties (TDOSCP)
- A-temporal dependent object representational constraint properties (ADORCP)
- Temporal dependent object representational constraint properties (TDORCP)

- A-temporal dependent object behavioral constraint properties (ADOBCP) and
- Temporal dependent object behavioral constraint properties (ADOBCP)

An overview is shown in Figure 39. Definitions about the above quoted types of *DOCP* are not provided since they correspond to the definitions of a-temporal and temporal *SCP*, *BCP* and *BCP* (see *Definition 14 – Definition 19*).

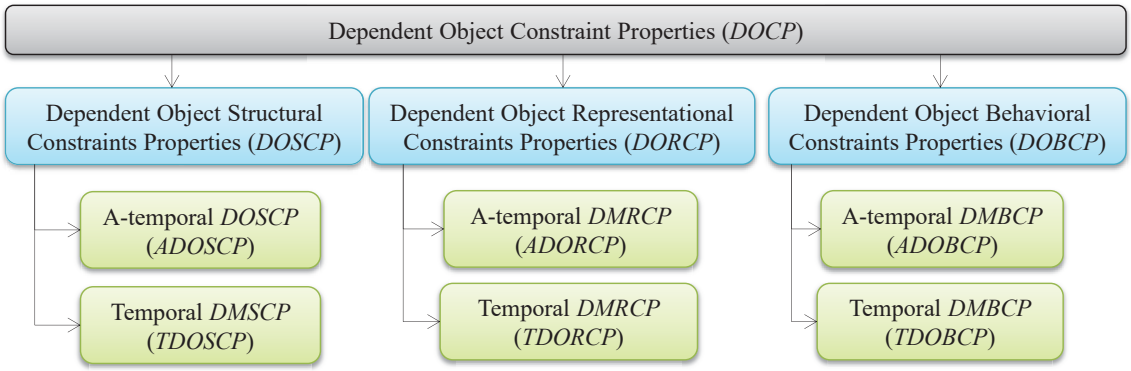


Figure 39. A classification of dependent object constraint properties *DOCP*

As illustration, the following TDOSCP is proposed, considering the metamodel shown in Figure 30 and the model shown in Figure 34:

CP_{15}: "It is always true (every execution step) that if the AI unit is alerted of an ongoing threat, it must send a report to the surveillance center, even if this threat appears not to be an incident"

The verification process takes place as soon as the functional and physical architecture shown in Figure 38 is executed. A feedback is provided after or during the execution. It is important to note that this property can be locally applied to selected AI Unit and Surveillance Center components and to the Detecting Fire and Detecting Flood functions and do not apply to other components or functions of the same model. For this type of properties, a model-checker must be integrated with a simulator.

3.3.3 Composite DSML and Model lifecycle

The composite DSML and model lifecycle is a modified version of the DSML and model lifecycle (see Section 3.2) for the design and management of composite DSMLs and models based on properties (see Section 3). It is illustrated in Figure 40 composed of two major phases: *(1) "Composite DSML design time"* and *(2) "Composite DSML run time"*.

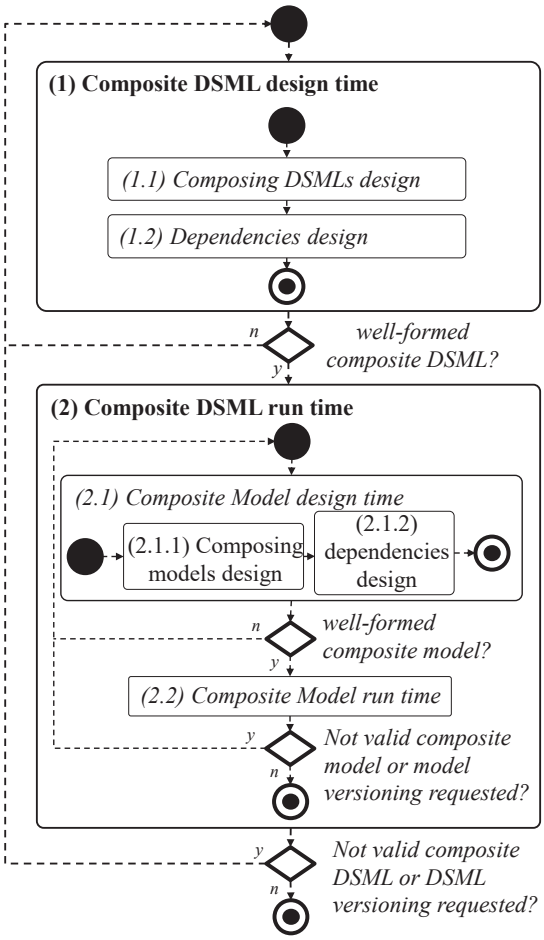


Figure 40. Composite DSML and model lifecycle.

3.3.3.1 Composite DSML design time

The phase of Composite DSML design time splits into two sub-phases: (1.1) Composing DSMLs design and (1.2) Dependencies design.

During the first sub-phase, DSML designers conceptualize their domain knowledge by creating and verifying various DSMLs for different viewpoints denoted “composing DSMLs”. This process is schematized in Figure 27 and detailed in Section 3.2.1.

The second sub-phase consists in designing the dependencies between composing DSMLs:

- The structural dependencies between abstract syntaxes formalized as structural dependency properties (SDP) (see *Definition 33*)
- The representational dependencies between concrete syntaxes formalized as representational dependency properties (RDP) (see *Definition 34*)

- The behavioral dependencies between dynamic semantics formalized as behavioral dependency properties (BDP) (see *Definition 35*)
- The dependent constraint properties (DCP) (see *Definition 36*)

Figure 41 shows the Dependency design phase composed of four phases that are performed in parallel:

- Structural dependency design
- Representational dependency design
- Behavioral dependency design
- Dependent constraints design

During the phase of *Structural dependency design*, a metamodeling language is used to formalize the structural dependencies between the abstract syntaxes that belong to the composing DSMLs in a composite DSML. The goal is to conceptualize the concepts and the relations that define the dependencies between two abstract syntaxes, forming the set of structural dependency properties *SDP* through a metamodel. This phase is finalized by a syntactical verification to ensure the well-formedness of the designed *SDP*, i.e., the respect to the rules imposed by the used metamodeling language (conformity to the meta-metamodel of the used metamodeling language).

In parallel, the phases of Representational dependency design and Behavioral dependency design take place based on:

- the stakeholders' domain knowledge,
- the structural dependencies that are in phase of design and thus partially provided (considering only the concepts and relations that are already defined),
- the concrete syntaxes and the dynamic semantics that belong to the composing DSMLs in a composite DSML.

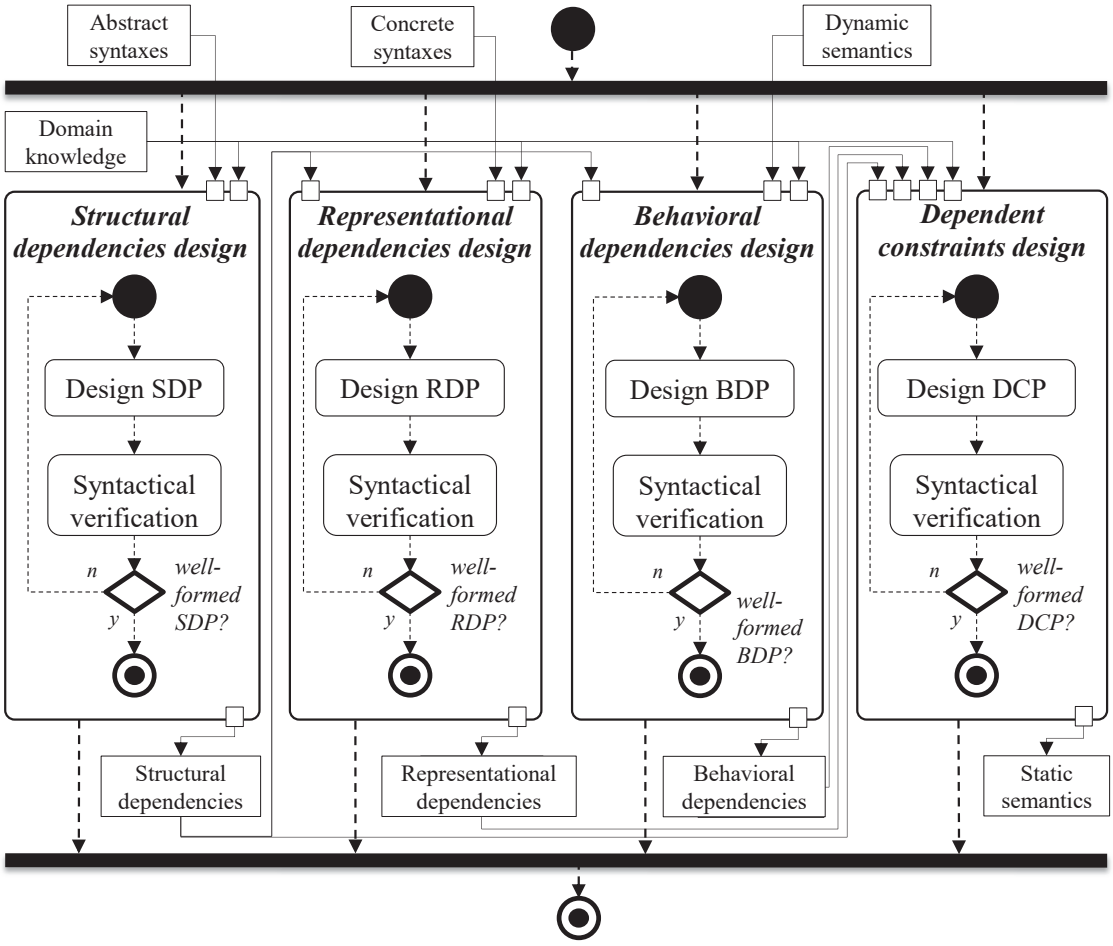


Figure 41. The dependencies design phase for a composite DSML.

The goal is to formalize, on the one hand, the representational dependencies of different concrete syntaxes, forming the set of representational dependency properties *RDP* by using a concrete syntax language, and on the other hand, the behavioral dependencies of different dynamic semantics, forming the set of behavioral dependency properties *BDP* by using a behavioral language. The designed dependencies must be syntactically verified to ensure:

- the well-formedness of the RDP, i.e., the respect to the rules imposed by the used concrete syntax language.
- the well-formedness of the BDP, i.e., the respect to the rules imposed by the used behavioral language.:

Finally, during the phase of *Dependent constraints design*, a constraint modeling language is used to formalize the dependent constraint properties (see *Definition 36*) that complement the static semantics. The goal is to formalize additional information that cannot be implicitly defined by the structural, representational and behavioral

dependencies, forming the set of *DCP*. This phase is finalized by a syntactical verification to ensure the well-formedness of the designed *DCP*, i.e., the respect to the rules imposed by the used constraint modeling language.

The results of the *Dependency design* phase are:

- Well-formed structural dependencies
- Well-formed representational dependencies
- Well-formed behavioral dependencies
- Well-formed dependent constraint properties

Thanks to the dependencies, the composing DSML can be integrated into a composite DSML that is furthermore provided as input to the second phase of "Composite DSML run time" for the design of composite models.

3.3.3.2 Composite DSML run time

The Composite DSML run time phase is decomposed into two sub-phases: (2.1) "Composite Model design time", for the design and verification of composite models and (2.2) "Composite Model run time", for V&V analyses based on models.

The *Composite Model design time* splits into (2.1.1) Composing models design and (2.1.2) Dependencies design.

During the phase (2.1.1), various models are created and verified by using the composing DSMLs of the composite DSML. The models, here-denoted "composing models", represent different viewpoints of a SoI. The process of designing and verifying models is schematized in Figure 28 and detailed in Section 3.2.2.

The phase (2.1.2) "Dependency design" consists in designing the dependencies between composing models:

- The structural dependencies between the structures of different models are formalized as model structural dependency properties (MSDP) (see *Definition 40*)
- The representational dependencies between the representations of different models are formalized as model representational dependency properties (MRDP) (see *Definition 41*)

- The behavioral dependencies between the behaviors of different models are formalized as model behavioral dependency properties (MBDP) (see *Definition 42*)

But also at designing the dependent model constraint properties (DMCP) (see *Definition 43*) and the dependent object constraint properties (DOCP) (see *Definition 47*)

Figure 42 shows the Dependency design phase composed of five phases, among which the following three are performed in parallel:

- Model structural dependencies design
- Model representational dependencies design
- Model behavioral dependencies design

Followed by two phases that are also performed in parallel:

- Design of dependent model constraints
- Design of dependent object constraints

During the first three parallel phases, the previously designed and well-formed dependencies between the composing DSML (i.e., the structural dependencies (SDP), the representational dependencies (RDP) and the behavioral dependencies (BDP)) are used to design and parametrize the dependencies between composing models in a composite model, i.e., model's structural dependencies (MSDP), model's representational dependencies (MRDP) and model's behavioral dependencies (MBDP).

The designed dependencies must furthermore be verified for well-formedness. The *well-formedness verification of model's structural dependencies* consists in:

- A syntactical verification: conformity to the metamodel, i.e., to the SDP.
- Verification of *ADSCP*: the model's structural dependencies must be checked by a model checker or interpreter to determine if they respect the *ADSCP*.

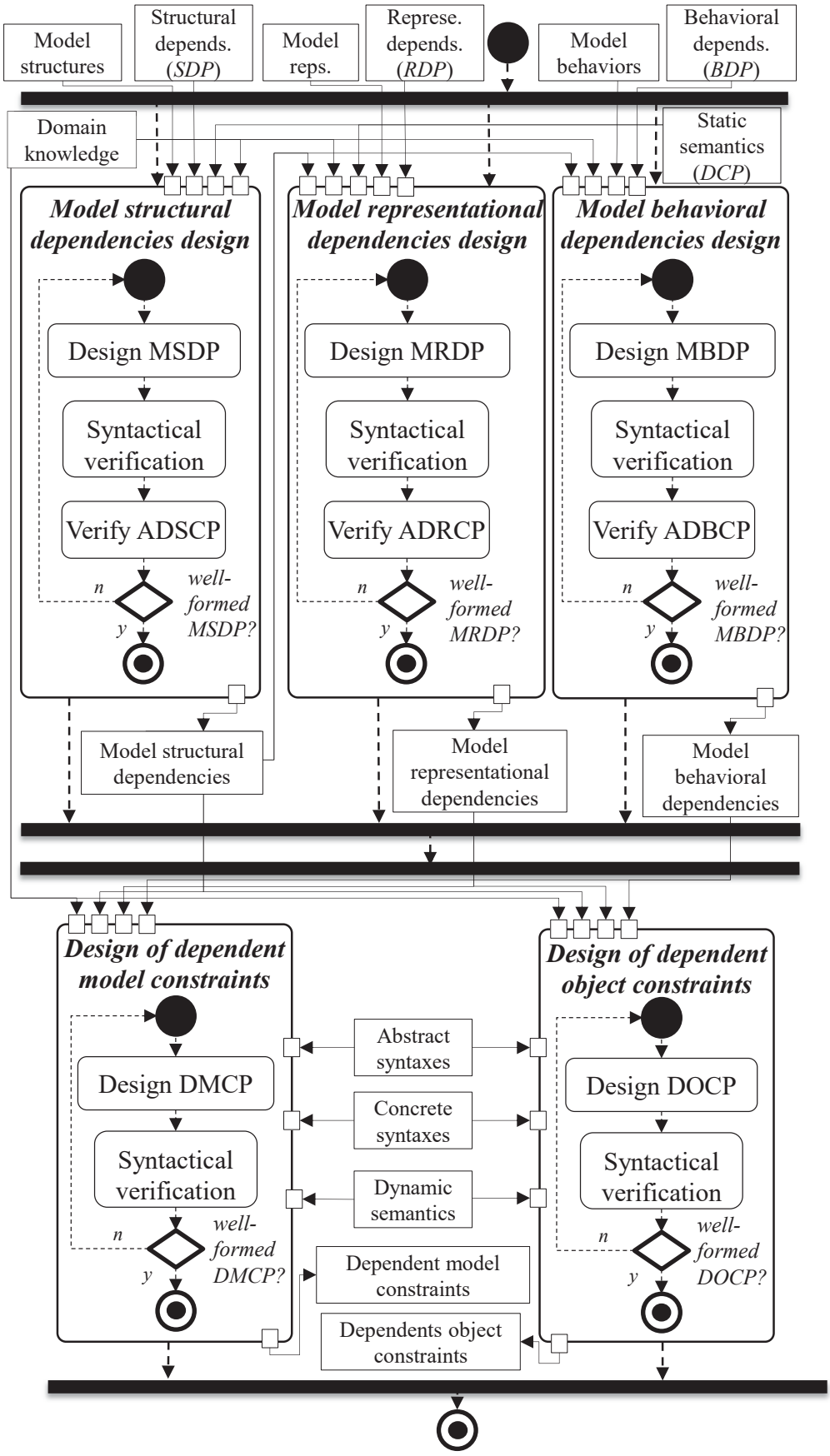


Figure 42. The dependencies design phase for a composite model.

The well-formedness verification of model's representational dependencies consists in:

- A syntactical verification: the model's representation dependencies must respect the rules imposed by the representational dependencies (*RDP*).
- Verification of *ADRCP*: the model's representational dependencies must be checked by a model checker or interpreter to determine if they respect the *ADRCP*.

The well-formedness verification of model's behavioral dependencies consists in:

- A syntactical verification: the model's behavioral dependencies must respect the rules imposed by the behavioral dependencies (*BDP*) (i.e., must be correctly parametrized).
- Verification of *ADBCP*: the model's behavioral dependencies must be checked by a model checker or interpreter to determine if they respect the *ADBCP*.

Thanks to the dependencies, the composing models can be integrated into a composite more that is first used for the design of "dependent model / object constraint properties" and then is provided as input to the last phase of "Composite Model run time" for V&V analyses.

As illustrated in Figure 42, during the following two parallel phases: design of dependent model / object constraint properties, system properties are formalized using a constraint modeling language based on the stakeholders' knowledge, the designed composite model (its structure, representation and behavior) and the used composite DSML. *DMCP* and *DOCP* complement the static semantics by constraint properties and are verified locally based on preselected models or objects (see Section 3.1.2 for more details). This phase is finalized by a syntactical verification to ensure the well-formedness of the designed *DMCP* and *DOCP*, i.e., to ensure that they respect the rules imposed by the used constraint modeling language.

The result of the sub-phase (2.1) 'Composite Model design time" is a well-formed composite model composed of:

- *Well-formed structure* that integrates the structures of several composing models
- *Well-formed representation* that integrates the representations of several composing models
- *Well-formed behavior* that integrates the behaviors of several composing models

- Well-formed model constraints
- Well-formed object constraints

Such composite model is then provided as input to the sub-phase (2.2) “Composite Model rune time” illustrated in Figure 43.

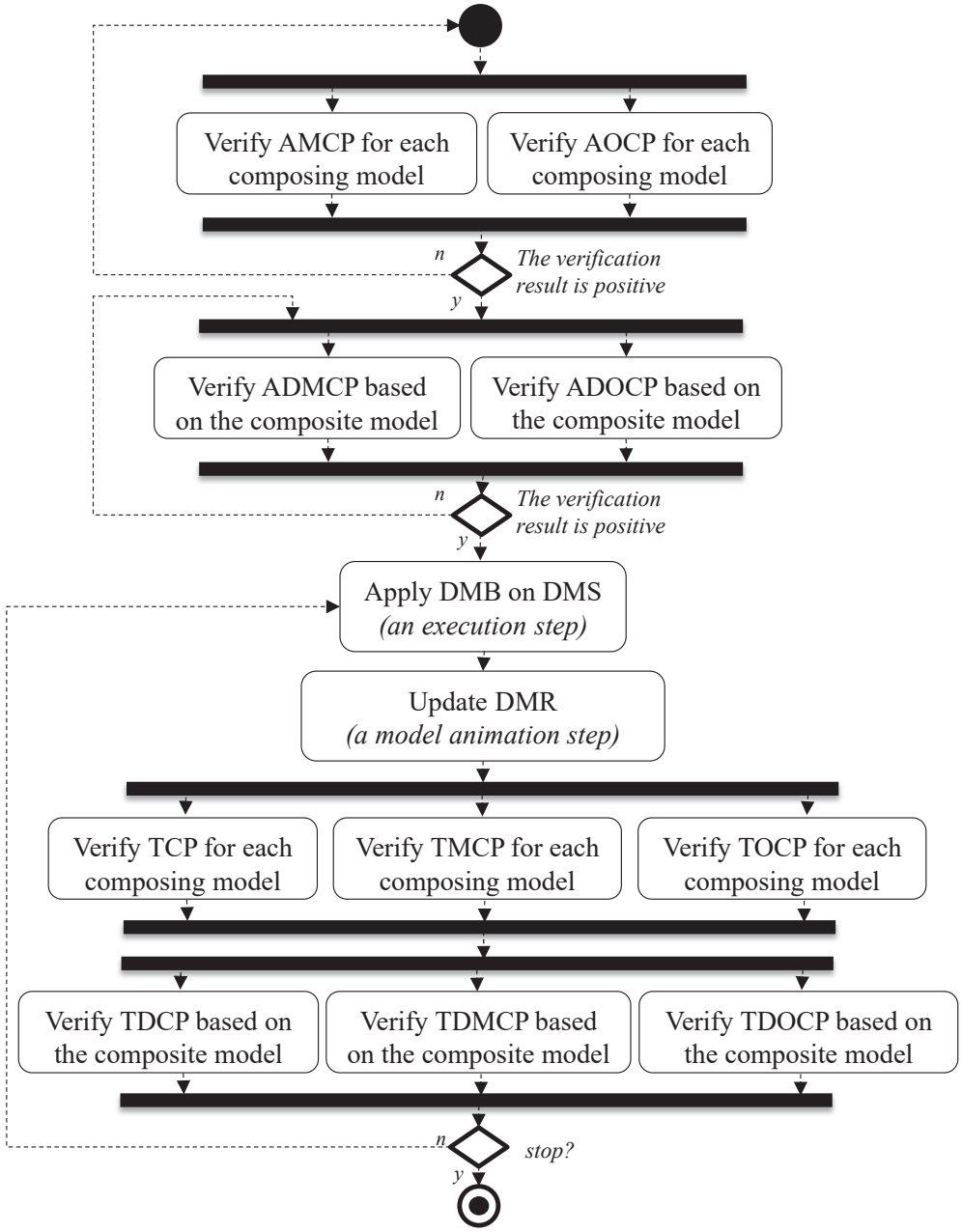


Figure 43. The Composite Model run time phase.

During this sub-phase, composite models are executed, animated and used as a base for formal proof, to assure that they represent as accurately as possible a SoI, i.e., to validate them.

First, each composing model must verify the *AMCP* as well as the *AOCP* (of course, if any of these properties are applied to the considered model or the objects contained in this model). For instance, the *AMSCP* CP8: "All functions must be performed infinitely (without an end) and in parallel" must be verified by the functional architecture of the fire and flood detection system illustrated in Figure 38. Next, the composite model is used to verify the *ADMCP* and the *ADOCP*. For instance, the ADMSCP CP14: "Each component performs one function which is its mission function" must be verified based on the dependencies between the functional and physical architecture of the fire and flood detection system illustrated in Figure 38

The following V&V analyses consist of simulation (i.e., model execution), model animation and verification of temporal constraint properties:

- The model execution is based on a gradual computation of the execution rules specified by the dependent model behavior (Chapter IV provides details on the simulation mechanism).
- The model animation is a result to the systematic visualization of changes (i.e., systematic modification of *MRP*) driven by the model execution according to the *DMR* (see *Definition 41*).
- Formal proof consists in verifying all temporal constraint properties for all composing models first separately (i.e., verifying the *TSCP, TMSCP, TOSCP, TRCP, TMRCP, TORCP, TBCP, TMBCP* and *TMBCP*) and regrouped together (i.e., the *TDSCP, TDMSCP, TDOSCP, TDRCP, TDMRCP, TDORCP, TDBCP, TDMBCP* and *TDMBCP*).

3.3.4 Synthesis

The complete modeling of a SoI (i.e., modeling that covers every aspect of that SoI) can be achieved by integrating various heterogeneous DSML into a composite DSML. Composite DSMLs can then be used for the design of multi-viewpoint SoI models called composite models, i.e., an integration of several heterogeneous models that conform to different DSMLs from a composite DSML, allowing a more expressive, realistic and complete representation of a SoI.

The design and management of composite DSMLs and models is based on a lifecycle, denoted "composite DSML and model lifecycle". Table 7 synthetizes the phases and sub-phases of the composite DSML and model lifecycle. It highlight the properties that

need to be modeled, the expected V&V analyses considering the designed properties, the languages used to model properties and the expected results of these V&V analyses for each phase and sub-phased.

Table 7. Synthesis of the composite DSML and model lifecycle. (MML–metamodeling language, CSL–concrete syntax language, BML–behavioral modeling language, CML–constraint modeling language).

Phase			**Properties**	**V&V analyses**	**Languages**	**V&V result**
Composite DSML design time		*Composing DSMLs design*	*CSP*	*For each SP: syntactical analysis*	*MML*	*Well-formed composite DSML*
			CRP	*For each RP: syntactical analysis*	*CSL*	
			CBP	*For each BP: syntactical analysis, verification of ABCP*	*BML*	
			CP	*Syntactical analysis of CP*	*CML*	
		Dependencies design	*SDP*	*Syntactical analysis of SDP*	*MML*	
			RDP	*Syntactical analysis of RPP*	*CSL*	
			BDP	*Syntactical analysis of BDP*	*BML*	
			DCP	*Syntactical analysis of DCP*	*CML*	
Composite DSML run time	*Composite Model design time*	*Composing Models design*	*DMS*	*For each MSP: syntactical analysis, verification of ASCP*	*CSP*	*Well-formed composite model*
			DMR	*For each MRP: syntactical analysis, verification of ARCP*	*CRP*	
			DMB	*For each MBP: syntactical analysis*	*CBP*	
			MCP	*Syntactical analysis of MCP*	*CML*	
			OCP	*Syntactical analysis of OCP*	*CML*	
		Dependencies design	*MSDP*	*Syntactical analysis of MSDP, Verification of ADSCP*	*SDP*	
			MRDP	*Syntactical analysis of MRDP, Verification of ADRCP*	*RDP*	
			MBDP	*Syntactical analysis of MBDP, Verification of ADBCP*	*MDP*	
			DMCP	*Syntactical analysis of DMCP*	*CML*	
			DOCP	*Syntactical analysis of DOCP*	*CML*	
	Composite Model run time			*Verification of AMCP and AOCP, Verification of ADMCP and ADOCP Simulation, Animation, Verification of TCP, TMCP and TOCP, Verification of TDCP, TDMCP and TDOCP*		*Valid (as much as possible) composite model*

3.4 Conclusion

The modeling of complex systems is divided into the modeling of different viewpoints, based on the stakeholders' domain knowledge. For this purpose, stakeholders must first conceptualize their domain knowledge in a form of modeling language (i.e., DSML) through different types of modeling properties, a design process that involves different type of language. We distinguish:

- Structural properties (SP) and dependencies between structural properties (DSP) designed by a metamodeling language;
- Representational properties (RP) and dependencies between representational properties (DRP) designed by a concrete syntax language;
- Behavioral properties (BP) and dependencies between behavioral properties (DBP) designed by a behavioral modeling language;
- Constraint properties (CP) and dependency constraint properties (DCP) designed by a constraint modeling language;

Stakeholders can then use such DSMLs to concretize their domain knowledge. More specifically, they use:

- The SP and the DSP to design the structure of a model as model structural properties (MSP) and the model structural dependencies (MSDP)
- The RP and the DRP to design the representation of a model as model representational properties (MRP) and the model representational dependencies (MRDP)
- The BP and the DBP to parametrize the behavior for a model as model behavioral properties (MBP) and the model behavioral dependencies (MBDP)

Furthermore, system properties express the requirements of systems or stakeholders based on a modeling artefact that is defined by modeling properties. We distinguish two types of system properties: model constraint properties (MCP), object constraint properties (OCP), dependency model constraint properties (DMCP) and dependency object constraint properties (DOCP).

The design process is illustrated in Figure 44.

The management of different type of properties is defined through a formalized lifecycle denoted “composite DSML and model lifecycle”. The lifecycle is composed of several phases and sub-phases. Each phase highlights which of the above quoted properties need to be designed and the V&V analyses that need to be performed.

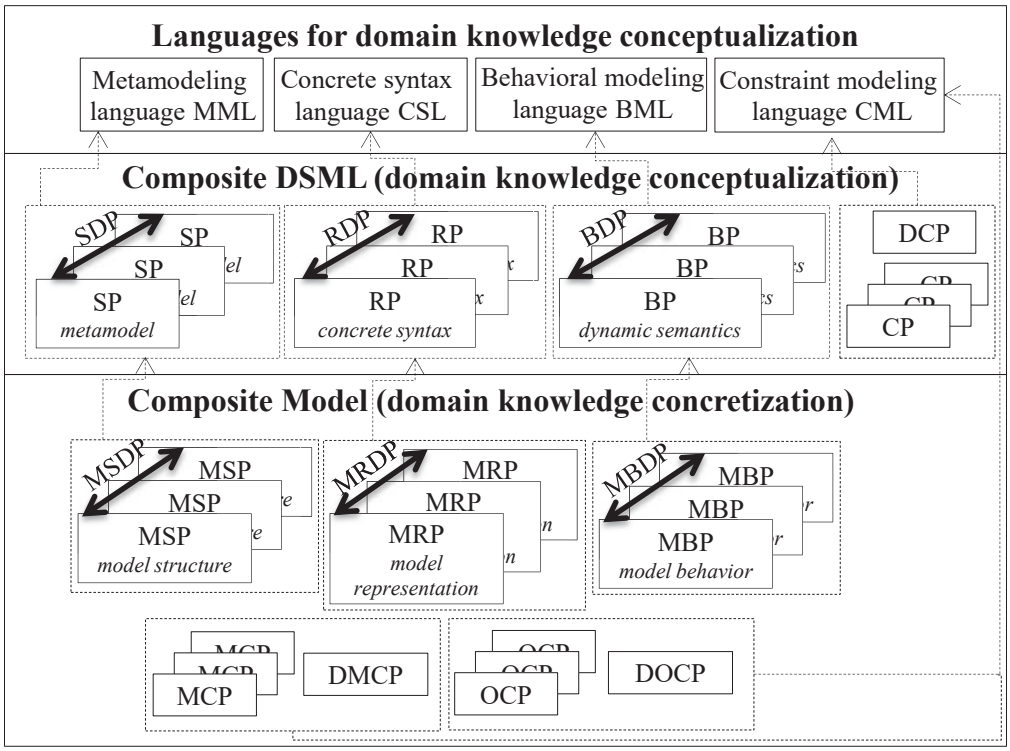


Figure 44. Conceptualization and concretization of domain knowledge.

CHAPTER IV

MODELING BEHAVIOR FOR MBSE

This chapter presents a part of the conceptual, methodological and technical contributions of this work. It is focused on the design of dynamic semantics (i.e., the Behavioral properties (*BP*) introduced in Chapter III) for executable DSML for MBSE. A map of Chapter's outline with respect to the type of contributions is shown in Figure 45.

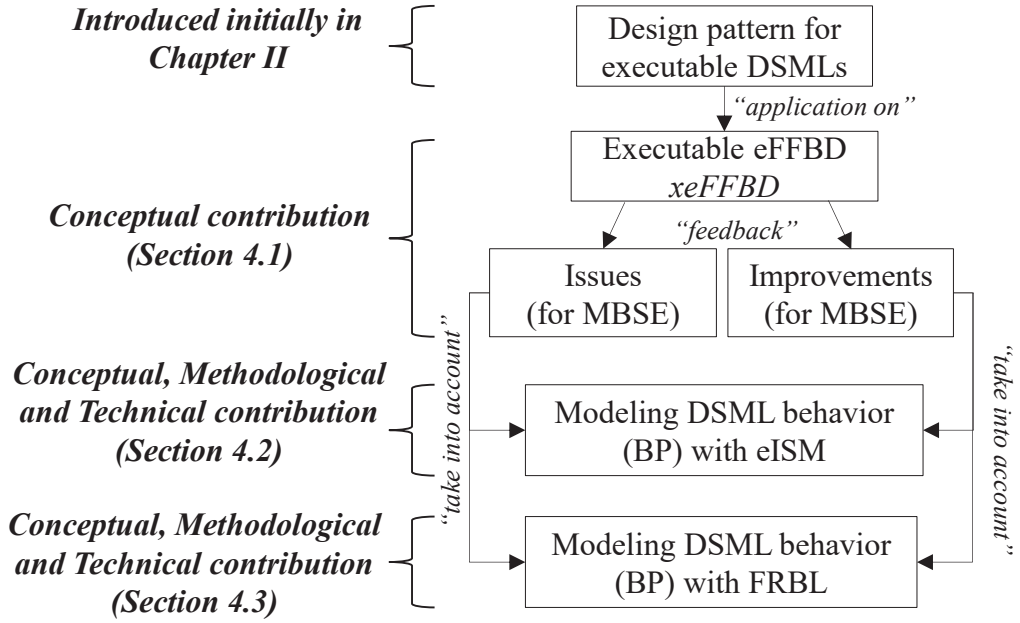


Figure 45. Map of conceptual, methodological and technical contributions of Chapter IV.

Chapter IV is structured as described in the next. Section 4 presents an evaluation of an intuitive approach for the design of executable DSMLs, based on the eFFBD language. The goal is to highlight issues and possible improvements of the selected approach for the context of MBSE. As a result to the feedback of the evaluation, we propose two approaches for the design of executable DSMLs discussed in Section 4.2 and 4.3.

4.1 Evaluating a design pattern for executable DSMLs

This section focuses on the design of dynamic semantics for a DSML by MBSE experts. The accent is placed particularly on assisting and automating the process as much as possible, allowing stakeholders to design dynamic semantics with minimal effort.

Similar claims have been made in (Combemale et al. 2012). They propose a design pattern that guides experts for the design of executable DSML, denoted xDSML. Briefly, an xDSML is a DSML that integrates dynamic semantics (i.e., referred as

executable semantics by the authors). For more details, the approach is illustrated and detailed in Chapter II.

The goal of this section is to evaluate the application of the approach in the field of MBSE. Therefore, Section 4.1.1 discussed the application of the approach on the eFFBD (see Section 2.4.3) in attempt to design an executable version, denoted xeFFBD. A discussion is then raised in Section 4.1.2 emphasizing the applicability within the field of MBSE, highlighting current problems, and Section 4.1.3 proposed possible improvements for the MBSE context.

4.1.1 Application: executable eFFBD - xeFFBD

The expected result is an executable eFFBD with integrated operational semantics that can be used to directly execute eFFBD models, without transforming them into a third-party approach as initially proposed by (Seidner 2009).

Let's first, recall that the design pattern for xDSML proposed in (Combemale et al. 2012) promotes two major phases:

- *Phase 1:* The design of a metamodel that contains the domain concepts and relations (DDMM), but also execution related information for concepts in a form of state model scattered across the SDMM that defines the concepts' states and the EDMM that defines the concepts' transitions between states.
- *Phase 2:* The design of execution-related information that describe when do state models evolve from one state to another and the results of their evolution of terms of changes of data in the model.

So first, during phase 1 domain concepts and relationships of the eFFBD are defined into the domain metamodel, denoted xeFFBD DDMM illustrated in Figure 46.

To reduce complexity and to ease understanding we propose to split the eFFBD DDMM into three packages *xeFFBD Diagram*, *xeFFBD Construct* and *xeFFBD Flow*. The xeFFBD DDMM package is obtained by merging them using the "merge" package operator defined by the MOF (MOF, 2014). Before presenting other concepts, let us first precise the core elements of eFFBD which are *Function, Resource* and *Item*. *Functions* describe what a system must do. They transforms one or more input *Items* in one or more output *Items* respecting transformation rules, possibly under control of triggers. *Resource* is something (data, material or energy e.g. human operator,

consumable, plans, etc.) that is requested and utilized or consumed during an inputs/outputs transformation. Requested resources are considered as independent from transformation goal and they are requested for function execution that modifies them. *Item* is something (data, material or energy) that is requested and transformed by function in order to provide another(s) distinct *Item(s)*. Taking into account its type, an *Item* can be consumed or can remain available during certain time duration after which its value becomes obsolete and unusable. These core elements are characterized by temporal attributes e.g. minimal and maximal time of execution, life time, etc.

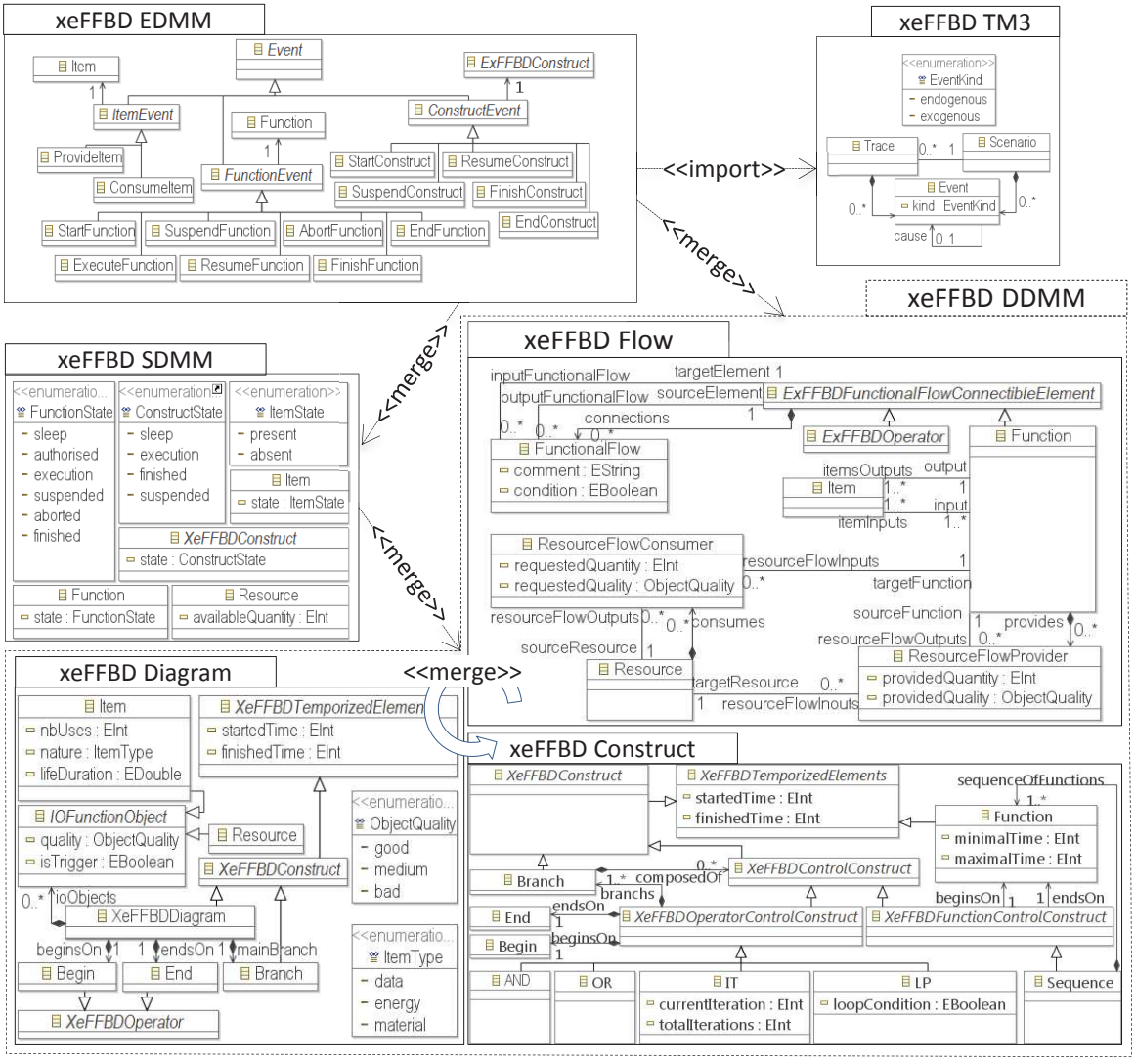


Figure 46. xeFFBD Phase 1 – design stages.

xeFFBD Diagram illustrated in Figure 46 is the core package describing a eFFBD diagram as a quadruplet of begin and end operators, main branch and set of input/output objects carried by flows. Begin and end describe starting and finishing

points in a diagram. The branch is composed of various control constructions named eFFBD Constructs described in the next. Two sorts of input/output objects are then available: items and resources respectively carried out by item flows and resource flows as detailed below. Last a diagram is temporized element, having started and finished execution time.

xeFFBD Construct package illustrated in Figure 46 represents different constructions recurring into a eFFBD Diagram. These constructions allows engineer to describe how functions are chained and how their execution is controlled in different manners introducing possibility to describe function parallelism, sequence, exclusion, and choices. A construct can either be 1) a function control construct composed of a set of functions (eventually one unique function) put in sequence, or 2) an operator control construction containing minimum one branch beginning on a begin operator and ending on an end operator, Four types of operator control construction are introduced: AND, OR, Iteration and Loop. A fifth one, named replication construction, is not considered at this moment. AND and OR constructions contain minimum two branches and they represent respectively parallel and exclusive execution of branches. Iteration and Loop constructions represent two possibilities of repetitive execution of one branch differing in the stop condition. Iteration fixes a number of iterations, while loop stops on a Boolean condition. Constructions are temporized elements having started and finished execution time.

xeFFBD Flow package illustrated in Figure 46 describes what are the three types of flows that can be handled in an eFFBD: functional flow, item flow and resource flow. A functional flow describes the order in which functions are executed (related to the primitive relation successor/predecessor between two functions). It is represented by the functional flow class connecting functional flow connectable elements which are either operators or functions. A Resource Flow describes requested Resources of a function that consumes them and restores them after execution, modifying eventually some of resource characteristics such as its quality and quantity levels. For this a Resource Flow is characterized by two attributes: quantity and quality. Quantity attribute indicates the requested amount of resource, consumed as an input by a function in order to execute it (requested quantity), and provided as an output after execution of related functions (provided quantity). Quality attribute indicates the level of resource quality, requested as an input in order to execute related functions (requested quality), and restituted after

function execution as an output altering then eventually the level of quality of the resource (provided quality) i.e. mixing for instance its availability and its efficiency. Item flow relates Item with function by input or output relationships. These relationships describe items that are needed and consumed as inputs for function execution and items that are provided as output after execution. Provided items are a result from transformation of inputs flows and eventually under the help or the control of resource flows. Note that there is a special kind of triggering items and resources that can trigger function execution, controlling then function start and/or stop conditions. Functional and resource flow have attributes (comment, condition and quantity, etc.), so they are represented in the metamodel using the class-association pattern, while item flow is represented using associations.

Once a DDMM is defined, the second design stage consists in defining the SDMM package, here-denoted xeFFBD SDMM. This package contains the possible states of selected domain concepts, denoted *evolving concepts* because instances of these concepts will become able to evolve during model execution. In the case of the eFFBD language, we have chosen the following concepts: Construct, Function, Item and Resource. The eFFBD SDMM package is illustrated in Figure 46. For instance, the concept Function contains six states: Sleep, Authorized, Execution, Finished, Suspended and Aborted. We interpret the states as follows. The input/output transformation described by a Function, is first possible (*Authorized)* i.e. the function can start but wait for Items (and eventually Resources) before being able to make the real transformation of energy, material and / or data (*Execution*) providing then the outputs items and resources (*Finished*). Due to external events, a function can be suspended and even aborted (*Suspended*, *Aborted*) in case of dysfunction of the component on which the function has been allocated. Note that, this is our interpretation of the functions' possible states. Depending on the level of detail that need to be captured by the states of a concept, it is plausible to specify them differently, adding details by adding additional states, removing details by removing states or even redefining them completely by new states. In some cases, it is event impossible to capture all state of a concept. For instance, Items and Resources are continuously transformed during the execution of a Function and the number of requested states to describe these evolutions can increase considerably, becoming sometimes infinite. For

such cases, we stress the need of a continuous behavioral model instead of a discrete-event model. For more details on this discussion see the next section.

The third design stage consists in defining the requested events for transition firing in the xeFFBD EDMM package. We defined three types of events: construct event, function event and item event as illustrated in Figure 46. Each of these events provokes a transition firing, consequently changing the state of an instance of an evolving concept.

The last design phase consists in defining a monitoring mechanism into a package denoted TM3. The design pattern proposes a generic trace mechanism that is here-reused and illustrated in Figure 46.

The phase 2 consists in specifying the execution semantics (into the package Semantics) for the previously defined xeFFBD metamodel that despite the execution-related information (states defined in the SDMM and transitions defined in the EDMM) is yet unexcitable. The goal is to define how and when transitions are fired, provoking state changes, and the consequent result of the state changes. For the design of this package, we use in a first stage a property-driven approach proposed in (Combemale et al. 2008). This approach describes how to define formally execution rules as formal properties, and how to formally verify these rules. The properties can be of three types: structural properties, temporal properties and quantitative properties. They can either be applied once during an execution, denoted existential properties, or all the time, denoted universal properties.

To sum up, the model execution relies on state models spread across the metamodel of a DSML (DDMM, EDMM and SDMM) and on rules defined as formal properties in the Semantics package. For instance, based on the state model of a Function, if the event *StartFunction* is applied on an instance of *Function* that is in the state *Sleep*, a transition is fired changing its state into *Authorized*.

As illustration, Figure 47 shows the execution of a simple eFFBD model. The functioning of lower level embedded constructs is controlled (i.e., *started* and *finished*) by higher level embedding constructs, taking also into account the connections between functions defined as functional flows. This model is composed of a starting point (entering arrow), an ending point (exiting arrow) and a main branch. A sequence is placed inside the main branch, containing three functions: F1, F2 and F3. Note that, for

the sake of simplicity, input and output object flows are neglected. The execution occurs as detailed hereafter. Each Construct controls the execution of Branches and Constructs it contains. So, the diagram starts the main branch which starts the sequence. Since this sequence contains functions, it must control their execution as follows. First, it starts the beginning function (F1), and awaits F1 to end execution, to start the following F2 function. This process repeats until the ending function, in this case F3, ends execution, which marks that the sequence has finish execution. The main branch then ends the execution of the sequence, before finishing its own execution. The diagram finally ends the execution of the main branch, which marks the end of the execution of the diagram and the eFFBD model.

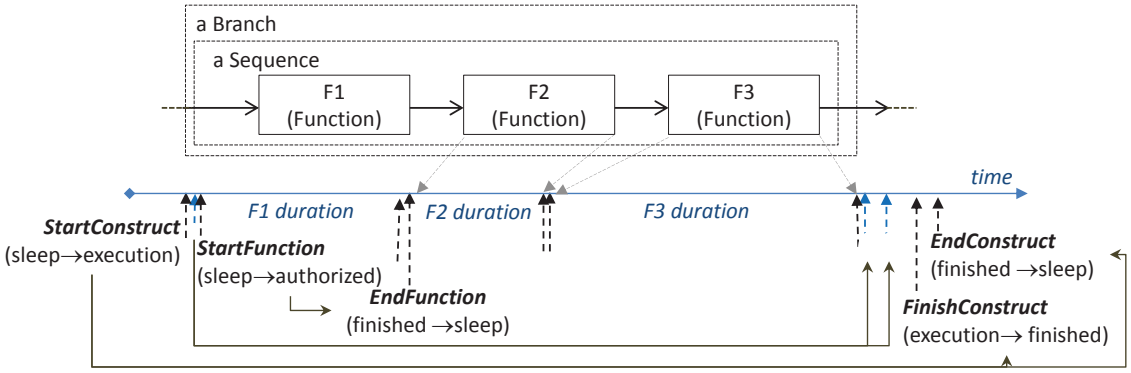


Figure 47. An execution of an eFFBD model

The execution rules of the concept Function are here-after formally defined using the previously property-driven approach proposed in (Combemale et al. 2008). An input/output transformation described by the Function is first possible, i.e., the function can start but has to wait for Items and eventually Resources (Figure 48, Eq.1) before being able to make the real transformation of energy, material and / or data (Figure 48, Eq.2) providing then the outputs items and resources and finishing its execution respecting minimal and maximal execution time (Figure 48, Eq.3). Note that, as previously discussed the execution of Functions is controlled by their containing Sequence. Therefore, the execution rules that are used to start and end the execution of functions take part in the execution rules of the Sequence construct. In addition, due to external events, a function can become temporarily suspended, can resume its execution or can abort execution (*Suspended*, *Aborted*). These external events can be then shared with other domain concepts from other modeling languages. For instance, the function behavior can depend from the component behavior that performs this function. So, the

event Suspended can be a common event shared between eFFBD and a PBD (executable Physical Block Diagram).

For f ∈ Function

(Eq. 1)	{ (f.state==***authorised***) **AND** (∀ i ∈ f.itemInputs,(i.state==***present***)) **AND** (∀ j ∈ f. resourceFlowInputs, ((j.requestedQuantity >= j.sourceResource.availableQuantity) **AND** (j.requestedQuality == j.sourceResource.quality)))) **implies *executeFunction***(f) }
(Eq. 2)	{ (f.state==***execution***) **implies** ((∀ i ∈ f. itemInputs, (***consumeItem***(i))) **AND** (∀ j ∈ f. resourceFlowInputs, (j.sourceResource.availableQuantity **-=** j.requestedQuantity)) }
(Eq. 3)	{ ((f.state==***execution***) **AND** ((internalTime - f.startedTime) >= minimalTime) **AND** ((internalTime - f.startedTime) <= maximalTime)) **implies** (***finishFunction***(f)) }
(Eq. 4)	{ (f.state==***finished***) **implies** ((∀ i ∈ f. itemOutputs, (***provideItem***(i))) **AND** (∀ j ∈ f. resourceFlowOutputs, (j.targetResource.availableQuantity **+=** j.providedQuantity)))}

Figure 48. The semantics of a Function as execution rules.

4.1.2 Discussion: current problems and causes

The discussed design pattern for xDSML proposes an effective and relevant solution that guides and assists experts for the specification of dynamic semantics for a DSML. The application of this design pattern to the field of the MBSE rises however several issues that seem crucial and remain partially or completely uncovered. They are discussed in the next, highlighting possible conceptual, methodological and technical improvements that might aid to complement this approach for the needs of the MBSE context.

Issue 1: state notion and formalization. After all domain concepts and relationships are identified and defined inside a DDMM, first, a sequence of states for all evolving concepts has to be defined inside a SDMM following Discrete Events Systems theory where a concept may evolve into one of a number of different states. Second, transitions between states and events that trigger transition firing are defined inside the EDMM, together with execution rules and a semantics mapping mechanism into the Semantics package.

However, all behaviors are not based on discrete-event hypothesis, as previously discussed. Namely, some concepts (such as the Item and Resource concepts from the eFFBD) have much more detailed behaviors characterized by a continuum of different

states that they might evolve into. In such case, the behavior should ideally be specified by a *continuous model*. For example, a continuous model for the concept Resource can be specified by a differential equation that describes how the value of the resource changes in function to time.

Issue 2: improved readability. The discrete-events models that describe the behavior of concepts are scattered across the SDMM, the EDMM and the package Semantics. Namely, the SDMM contains the possible states, the EDMM defines the transitions between states and the package Semantics defines when and how transitions are fired provoking state-changes.

Unfortunately, the readability of such behavior is limited for MBSE experts. Indeed, the classical graphical notation of a state-machine model composed of circles for states and links for transitions between states is more accessible and readable.

Issue 3: transient states detection and management. Considered approach defines temporal properties using the temporal OCL (TOCL) (Ziemann & Gogolla 2003). Temporal properties are examined taking into account a unique temporal dimension (discrete or continuous) that is used for event synchronization and transitions firing.

However, when modeling critical, parallel or distributed systems, it is very important to manage the stability of models every time they evolve. A behavioral model is "*stable*" if succeeding an evolution, taking into account the same inputs, the model cannot evolve in another state. Otherwise, the model is "*unstable*" and its current state is named "*transient*" state, as defined in the case of Sequential Function Chart (IEC 1999).

Issue 4: mechanism for formal proof. The question here concerns concepts and techniques to formalize and verify execution rules described as properties. Namely, the execution rules are specified as formal properties using the TOCL. A mechanism for formal verification is then proposed based on the TINA (time petri-net analyzer) model-checker. Unfortunately, this technique requires transforming the concepts' behavioral models (i.e., the states and transitions from the SDMM and the EDMM along with the properties from the Semantics package) into petri-nets models, facing the classical issues related to transformation approaches discussed in Chapter II.

Issue 5: designing dependencies in modeling languages – a way for model interoperability. In the context of MBSE, a SoI is modeled by using various models (relevant for one or more objectives) each one representing a viewpoint of a SoI (e.g.,

requirements, functional, physical, behavioral, etc.) as discussed in Chapter II. These models must be coherent, first separately and then considering the other models of the same SoI. Therefore, the used DSMLs must define their dependencies (as proposed in Chapter III), allowing the interoperation between viewpoint models.

Unfortunately, models interoperability is out of the scope of the studies approach. For this purpose, different DSMLs must integrate structural dependencies between their DDMM, and also behavioral dependencies between their SDMM, EDMM and Semantics.

4.1.3 Proposition: improvements for the MBSE context

The application of the xDSML design pattern in the field of the MBSE raised five issues that seem crucial and remain partially or completely uncovered. We propose in this section, for each of the above discussed issues a possible improvement relevant for the MBSE context.

Improvement 1: state notion and formalization. The specification of a continuous behavior by a finite number of states (i.e., by a discrete-events model) might sometimes become limited for V&V due to lack of details that need to be modeled. For example, a discrete-events model for the eFFBD concept Resource can be specified by a two-state state machine model (*sufficient* and *insufficient*) of which one of the states describes that the resource is sufficient and can be transformed and the other describes that the resource is insufficient and cannot be transformed. In such scenario, details about the Resource's quality or the quantity are neglected.

To address this issue, we adopt the symbolic representation of states by variables introduced initially by the automata theory, as proposed by (Vandermeulen 1996) for the Interpreted Sequential Machine (ISM). This allows increasing the level of details by combining discrete-events models and variables, denoted "symbolic variables" or "state variables". For instance, in the case of the Resource, the behavior can be defined by a two-state model (with states: *sufficient* and *insufficient*) and two additional symbolic variables representing resources' quality and quantity. For this purpose, the discrete-event models, along with the specification of states and transitions must also integrate additional component for the specification of symbolic variables. In the case of the ISM, this component is denoted data part (Vandermeulen 1996).

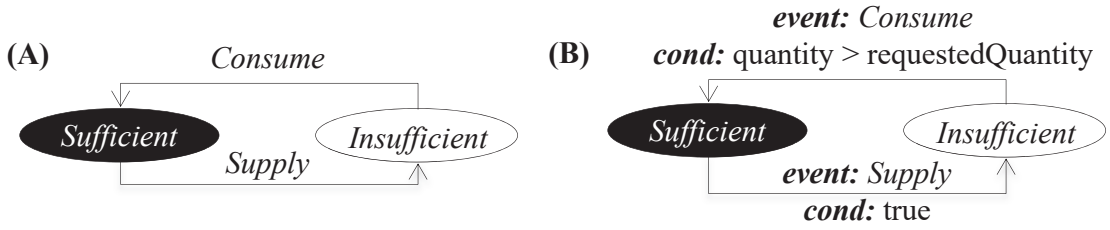


Figure 49. Improving readability by abstraction.

Improvement 2: improved readability. To improve readability of the discrete-events models for the, we propose first to abstract the behavior by using for example the graphical notation of the finite state machine model, as proposed above. For instance, the two-state behavior of the eFFBD concept Resource is illustrated in Figure 49 (A). This allows making the connection between a concept of a DDMM, its states from a SDMM and different events from an EDMM that cause the state change. Nonetheless, the event firing is preconditioned by the execution rules from the machine.

Furthermore, we propose to refine transitions by associating to each one a pair of <*condition,event*> as shown by Figure 49 (B). The *condition* (True by default) is a Boolean function computed on various variables: states variables proposed in *Improvement 1*, attributes of any domain concept from the local DDMM or external variables corresponding to other domain concepts from another DDMM. Moreover, we classify conditions and events into *inter* and *intra*.

- *Intra conditions/events* are based on information from the current model.
- *Inter conditions/events* are based on information from one or several other models from the same SoI whose behavior interacts with the behavior of studied model.

Inter conditions/events are the foundation stone of the behavioral interoperability invoked by above discussed *Issue 5*.

The *event* is similar to the stimuli, proposed in the approach. In addition, we adopt two rules from the discrete event modeling theory:

1) Two events cannot be simultaneous so it is always possible to distinguish them.
2) There exists a default event *e* always occurring.

A Transition can then be fired when receiving an event, if and only if its condition evaluates to true.

Improvement 3: transient states detection and management. Stability management consists in checking the stability of a behavioral model every time the model evolves. Managing models stability involves a transient state detection algorithm that manages two time scales, as proposed by the Ptolemy approach (Lee 2003):

- An external time scale
- An internal time scale

Both time scales are modeled by two independent logical clocks. The internal time scale is reinitialized every time the model evolves and incremented while the model is in transient state, every time calculating its future state, eventually reaching its stability.

As illustration, Figure 50 shows the outcomes of the models' execution with and without stability management.

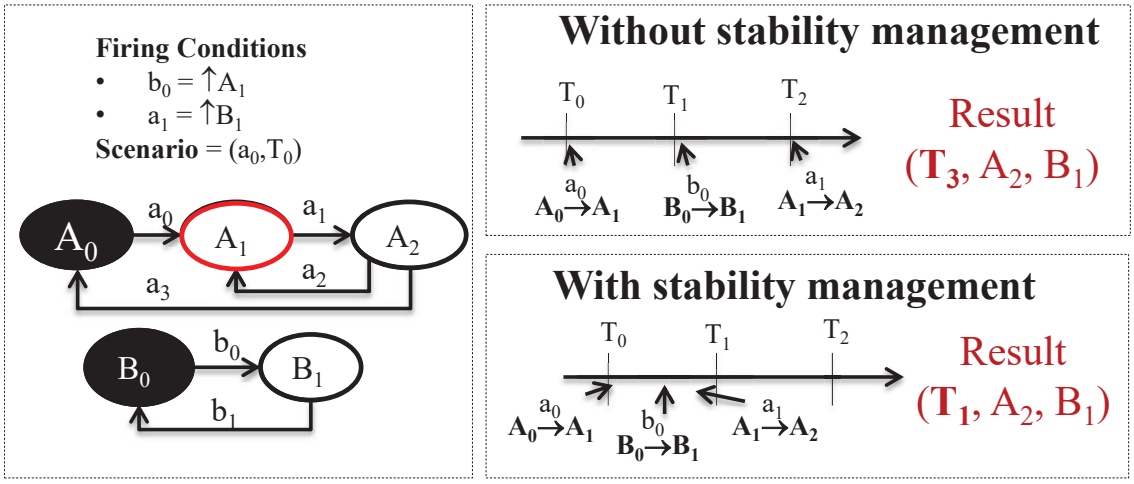


Figure 50. Transient state management.

The figure is interpreted as follows. The initial states of the models are respectively A_0 and B_0. According to the scenario (a_0,T_0), at time T_0, the a_0 event fires the transition between A_0 and A_1, changing the current state of the first model into A_1. During the second time unit (considering the firing condition $b_0 = \uparrow A_1$) the transition b_0 is fired, changing the current state of the second state model into B_1. Since $a_1 = \uparrow B_1$, the transition b_0 is fired during the third time unit, changing the current state of the first state model into A_2. So as a result the state models are in states A_2 and B_1 at the end of the third time unit as shown in the top right side of Figure 50. However, with stability management, this whole evolution is done in one single time unit as shown in the bottom right side of Figure 50.

We propose furthermore in Chapter V an algorithm for transient state detection and management. Briefly, the functioning of this algorithm is described as follows. Values of each variables appearing in a conditions and occurring events are read and then frozen in external time. State models evolve taking into account these values in an internal scale allowing then to detect transient states and to reach the stable state. The external time depends from environment evolution scale and can be seen as a physical scale time defined as a set of moment ordered by taking into account Time Unit duration. It is initialized when a simulation starts. The internal time is however a logical scale time as defined in Discrete Event Simulation theory. It is initialized at each moment defined in external time and there are no common temporal dimensions between internal and external scales.

Improvement 4: mechanism for formal proof. The goal here is to provide a mechanism for formal proof allowing a direct verification of the behavior instead of transforming it into third-party formalism such the TINA model-checker. For this purpose, we stress the need of formalism for the design of behavior that allows formal proof. For example, behavioral models designed by the previously discussed ISM (Vandermeulen 1996) have formal underlying structure that supports symbolic model checking. In (Vandermeulen et al. 1995) the authors describe how can ISM models be formally verified based on the temporal boolean difference.

In addition, it will be interesting to formalize system requirements as properties and to formally verify them. For this purpose, despite the above discussed issue of direct verification, it is equally important to adopt a strategy for requirement formalization. Such strategy must bridge the gap between the informal languages used first to specify requirements and the semi-formal and formal languages that provide verification mechanism, as proposed in (Chapurlat 2013). The goal of this work is to define an appropriate and tooled property modeling and proof approach inspired by the above quoted research results.

Improvement 5: designing dependencies in modeling languages – a way for model interoperability. In the context of MBSE, a SoI is modeled by using various models each one representing a viewpoint of a SoI as discussed in Chapter II. These models must be coherent, first separately and then considering the other models of the same SoI. Therefore, as stated as working hypothesis in Chapter III, stakeholders have

defined dependencies between DSMLs that are used in each viewpoint, allowing then a partial interoperability between viewpoint models. So, this notion of model interoperability is here-considered limited to:

- *Structural interoperability*: models are structurally bound together (see *Definition 33* and *Definition 40*).
- *Behavioral interoperability*: models are behaviorally bound together considering data from other models during model execution (see *Definition 35* and *Definition 42*).

Both structural and behavioral interoperability must furthermore be taken into account by:

- *a simulation mechanism* for a coordinate simulation of all behavioral models from all domain models
- *a proof mechanism* for a formal verification of properties considering all models of a SoI (as opposed to verification that takes into account only one model)

4.2 Modeling the behavior of a DSML with a Discrete-Events Language

This section introduces a discrete-events language in a form of a DSML for the modeling of discrete-events behaviors. The DSML is an extended version of the Interpreted Sequential Machine (Vandermeulen 1996), denoted eISM. The goal is to use it for the design of behavior (dynamic semantics / executional semantics) for a DSML. Indeed, we are inspired by the idea of designing discrete-events models for concepts of the DDMM (denoted evolving concepts) as discussed in Section 4. However, instead of scattering the discrete-events models across several loosely coupled modules (SDMM, EDMM and the package Semantics) we propose to associate them directly to the domain concepts.

Following the discussions of Section 4.1.2 and Section 4.1.3, we argument first the choice of the eISM in Section 4.2.1. Then in Section 4.2.2 we introduce and formally define the eISM. In Section 4.2.3 we illustrate the integration process between eISM and the metamodeling language EMOF. We discuss several technical issues related to the eSIM in Section 4.2.4. In Section 4.2.5 we propose a formal proof mechanism for eISM and in Section 4.2.6 and 4.2.7 we show illustrate based on two examples.

4.2.1 The eISM languages: discussion about the choice

The ISM initially introduced in (Vandermeulen 1996) is a formal language based on discrete-event hypothesis for modeling and verifying the behavior of systems and their interactions with the environment. According to the authors, the ISM has the following advantages in comparison to other discrete events modeling languages:

- First, it operates with typed input/output data (primitive types, e.g., Boolean, Integer, Real, Character or compound type) and complex expressions built using internal typed data.
- Second, it separates classical state/transition specification, here-denoted Control Part (CP), from data specification, here-denoted Data Part (DP).
- Third, ISM has formal underlying structure, based on the Linear Temporal Logic (LTL) abstracted in the form of Elementary Valid Formulas (EVF).

The first advantage makes the ISM applicable in the MBSE context. Namely, concepts from the DDMM of a DSML can be naturally used as a source of data. The separation of the state/transition specification (CP) from the data specification (DP) allows replacing some states that are normally added into the CP, as "symbolic" variables in the DP, limiting the combinatorial explosion of the number of states. This is helpful for continuous behaviors as previously discussed for the eFFBD Resource concept (see *Improvement 1* in Section 4.1.3). The graphical notation of ISM models can address the previously discussed readability issue (see *Improvement 2* in Section 4.1.3). The ISM formal underlying structure allows formal verification based on model checking techniques and tools (e.g. STEP, MEC, TINA or UPPAAL) by reusing the EVFs without any transformation as for instance discussed in *Issue 4* (Section 4.1.2). For example, in (Vandermeulen et al. 1995) the EVF are reused as a source to the Temporal Boolean Difference (TBD) method (discussed here-after). This method calculates the sensitivity of the present to the future evolution of ISM models.

Nevertheless, the initial version of ISM is not suited to address the *Issue 3* (the detection of transient states and stability management) and the *Issue 5* (model interoperability in terms of behavioral dependencies between DSMLs and synchronized execution of multiple ISM models) discussed in Section 4.1.2. Therefore, we propose an extended version of the ISM, denoted eISM, along with synchronization rules and mechanisms, allowing:

- Stability management and transient state detection
- Synchronized execution of multiple ISM models based on the blackboard communication pattern

In addition to the above quoted limitations, the mechanism for formal proof of ISM takes into account one ISM model, even though in a DSML there are multiple behavioral models that should be considered simultaneously by the formal proof mechanism. This problem becomes even more complicated when relating several DSML, because the formal proof mechanism must handle multiple sets of behavioral models, each one specifying he behavior of a composing DSML.

4.2.2 Introduction to the eISM: a formal specification

An eISM is composed of four interconnected parts called: Input Interpreter (*II*), Output Interpreter (*OI*), Control Part (*CP*) and Data Part (*DP*) as illustrated in Figure 51.

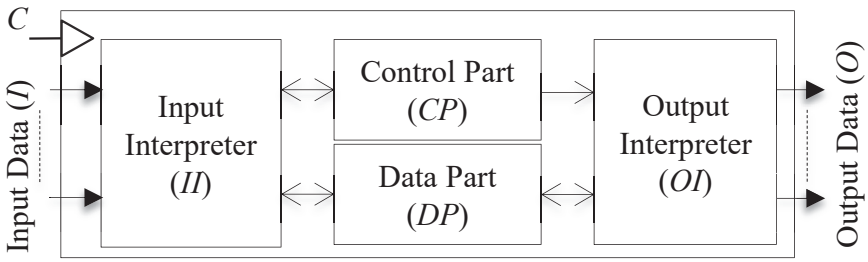


Figure 51. The components (modules) of an eISM model.

The *CP* is a graph of states and transitions. The *DP* holds the model data. The *II* interprets *input data* (gathered into the set *I*) available in the Blackboard (*BB*) and *model data* from the *DP*. Interpreted data takes part in the firing conditions that are associated with each transition of the *CP*, consequentially taking part in the *CP*'s evolution. The *OI* is an interface that interprets the evolution of the *CP* by updating the values of the output data (gathered into the set *O*) and the values of the model data from the *DP*.

An eISM model is formalized as a 6-uplet eISM $\stackrel{\text{def}}{=} \langle I, O, CP, DP, II, OI, \rangle$ where:

a) I is the set of input data available from the *BB*. Each input i_i is defined by a current value $cvalue_i$, a domain definition I_i and a type I_i', such as $I_i \subseteq I_i$'.
b) O is the set of output data that is sent to the *BB* by the *OI*. Each output o_i is defined by a current value $cvalue_i$, a domain definition O_i and a type O_i', such as $O_i \subseteq O_i$'.

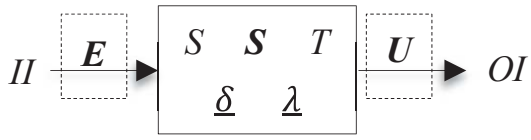


Figure 52. An overview of the Control Part (CP).

c) The *CP* (Control Part) illustrated in Figure 52, is defined as a graph of states related by labeled transitions and formally defined as a 5-uplet $CP \stackrel{\text{def}}{=} \langle S, \boldsymbol{S}, T, \boldsymbol{E}, \boldsymbol{U} \rangle$ where: $S = \{s_1, \ldots, s_v\}$ is a set of states, $\boldsymbol{S} = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_v\}$ is a set of state propositional variables, $T = \{T_1, \ldots, T_q\}$ is a set of transitions, $\boldsymbol{E} = \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_q\}$ is a set of firing condition propositional variables and $\boldsymbol{U} = \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\}$ is a set of update propositional variables. Transitions are given in the following form $T_i = [(\boldsymbol{s}_i, \boldsymbol{e}_j), (\boldsymbol{s}_k, \boldsymbol{u}_l)]$, as illustrated in Figure 53. By hypothesis, there is a unique state s_i that is active each moment of the evolution. When the state s_i is active (otherwise inactive), the propositional variable associated to that state i.e., $\boldsymbol{s}_i$ = *True* (*False* otherwise). In addition, firing condition propositional variables, $\boldsymbol{e}_j \in \boldsymbol{E}$, evaluate to *True* if an only if the corresponding firing condition function e_j computed by *II* returns *True*. A transition T_i can be fired, if and only if, the transition's firing condition propositional variable $\boldsymbol{e}_i$ evaluates to true and the source state of the transition T_i is an active state, by the transition function $\underline{\delta}$ defined as:

$$\underline{\delta}: \boldsymbol{S} \times \boldsymbol{E} \rightarrow \boldsymbol{S}$$
$$(\boldsymbol{s}_i, \boldsymbol{e}_j) \rightarrow \boldsymbol{s}_k$$

Firing a transition activates the output function $\underline{\lambda}$ defined as:

$$\underline{\lambda}: \boldsymbol{S} \times \boldsymbol{E} \rightarrow \boldsymbol{U}$$
$$(\boldsymbol{s}_i, \boldsymbol{e}_j) \rightarrow \boldsymbol{u}_l$$

As a consequence to these two functions, the source state of transition T_i is deactivated, its target state is activated and the corresponding update propositional variable $\boldsymbol{u}_l \in \boldsymbol{U}$ is set to *True*.

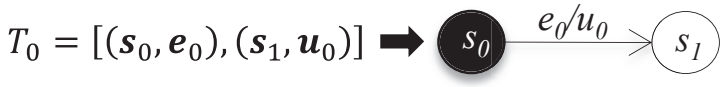


Figure 53. Example of Transition T_0 between initial state (s_0) and s_1.

d) The *DP* (Data Part) holds the model data that is used to specify transitions' firing condition functions *E* and update functions *U*. It is formally defined by a 2-uplet $DP \triangleq^{\text{def}} \langle LD, ID \rangle$ where: $LD$is a set of language data directly derived from the corresponding DSML concept (denoted $c \in CPT$) and $ID = \{id_1, \dots, id_d\}$ is a set of internal (to the eISM model) data, explicitly needed for the description of firing condition and update functions. The variables from the *ID* set are defined by a current value *cvalue*, a domain definition *DP* and a type *ID'* such that *ID* $\subseteq$ *ID'*.

Figure 54. An overview of the Input Interpreter (II).

The *LD* set if derived directly from a domain concept *c*, i.e., from its attributes defined by the set *A* and relations defined by the set *REL*.

$$LD \triangleq^{\text{def}} \langle AV, RV^{nbref}, CV^{nbcom}, IAV, IRV^{nbiref}, ICV^{nbicom} \rangle$$

where:

- *AV* is the set of variables directly derived from the attributes of the concept *c*, formally defined as: $\forall att \in AV, att \in A$.
- RV^{nbref} are *nbref* sets of variables derived from the references (i.e., relationships of type reference) of the concept *c*. *nbref* is the number of references of the concept *c*. This is formally defined as: $nbref := |REL| \; and \; \forall r \in REL, r.type = 'reference'$. Each set $RV_i \in RV^{nbref}, \forall i \in [1..nbref], |RV_i| \in [lb..ub]$ might contain minimum *lb* and maximum *up* number of variables (*lb* is the lower bound multiplicity and *up* is the upper bound multiplicity of the reference).
- CV^{nbcom} are *nbcom* sets of variables derived from the compositions (i.e., relationships of type composition) of the concept *c*. *nbcom* is the number of compositions of the concept c, formally defined as follows: $nbcom := |REL| \; and \; \forall r \in REL, r.type = 'composition'$. Each set might contain minimum *lb* and maximum *up* number of variables (*lb* is the lower bound

multiplicity and *up* is the upper bound multiplicity of the reference), formally defined as follows: $CV_i \in RV^{nbcom}, \forall i \in [1..nbcom], |CV_i| \in [lb..ub]$.

- *IAV* (Inherited Attribute Variables) is a set of variables derived from the attributes of the more generic concepts of *c*, formally defined as: Let *IA* be the set of inherited attributes of c: $\forall att \in IAV, att \in IA$.
- IRV^{nbiref} are *nbiref* sets of variables derived from the references (i.e., relationships of type reference) of the more generic concepts of *c*. Let *IREF* be the set of inherited references: *nbiref* is the number of inherited references of the concept *c* $nbiref := |IREF|$. Each set might contain minimum *lb* and maximum *up* number of variables (*lb* is the lower bound multiplicity and *up* is the upper bound multiplicity of the reference): $IRV_i \in IRV^{nbiref}, \forall i \in [1..nbiref], |RV_i| \in [lb..ub]$.
- ICV^{nbicom} are *nbicom* sets of variables derived from the compositions (i.e., relationships of type composition) of the more generic concepts of *c*. Let *ICOM* be the set of inherited compositions: *nbicom* is the number of inherited compositions of the concept *c* $nbicom := |ICOM|$. Each set $ICV_i \in ICV^{nbicom}, \forall i \in [1..nbicom], |CV_i| \in [lb..ub]$ might contain minimum *lb* and maximum *up* number of variables (*lb* is the lower bound multiplicity and *up* is the upper bound multiplicity of the reference).

e) The *II* (Inputs Interpreter) illustrated in Figure 54, reads data (input data from the *BB* and model data from the *DP*) and based on it, evaluates the firing condition propositional variables that are associated with transitions of the *CP*. It is formally defined as 5-uplet $II \stackrel{\text{def}}{=} \langle I, LD, ID, \underline{E}, \boldsymbol{E} \rangle$ where $\underline{E} = \{\underline{e}_1, \dots, \underline{e}_x\}$ is a set of firing condition functions and $\boldsymbol{E} = \{\boldsymbol{e}_1, \dots, \boldsymbol{e}_x\}$ is a set of firing condition propositional variables. Firing condition functions are composed of a Boolean expression part (evaluated using input and model data) and a requested events part (evaluated using only input data), formally defined as: $\forall \underline{e_i} \in \underline{E}, \underline{e_i} = \{cond_i, event_i\}$. The firing condition function evaluates to *True*, if both parts compute to *True*, *False* if at least one computes to *False*. This is formally defined as:

$$\underline{e_i}: I \cup LD \cup ID \rightarrow \{0,1\}$$
$$\underline{e_i}(id_1, \dots, id_{|I|}, ld_1, \dots, ld_{|LD|}, id_1, \dots, id_{|ID|}) = (0|1)$$

Every firing condition propositional variable is associated with a firing condition function. This is formally defined as:

$$\forall i \in [1,..,x], \underline{e_i}\left(id_1,\dots,id_{|I|},ld_1,\dots,ld_{|LD|},id_1,\dots,id_{|ID|}\right) = 1$$

$$\Leftrightarrow (\boldsymbol{e_i} = True)$$

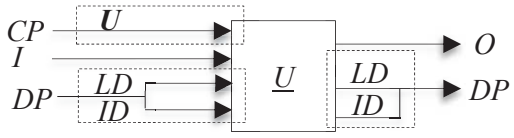


Figure 55. An overview of the Output Interpreter (OI).

f) The *OI* (Outputs Interpreter) associates the update propositional variables with the corresponding update functions and evaluate these update functions. As a result, the model data from the *DP* and on the output data that is send to the *BB,* are both modified (updated). The *OI* is illustrated in Figure 55 and is formally defined as a 6-uplet $OI \stackrel{\text{def}}{=} \langle LD, ID, I, O, \boldsymbol{U}, \underline{U}\rangle$ where $\boldsymbol{U} = \{\boldsymbol{u}_1,\dots,\boldsymbol{u}_q\}$ is a set of update propositional variables and $\underline{U} = \{\underline{u}_1,\dots,\underline{u}_q\}$ is a set of updates. Each update might be associated with three types of update functions:

1) update functions for output data, formally defined as:

$$\underline{u}_{ij}: I \cup LD \cup ID \rightarrow O$$
$$\underline{u}_{ij}\left(id_1,\dots,id_{|I|},ld_1,\dots,ld_{|LD|},id_1,\dots,id_{|ID|}\right) = \left(o_1,\dots,o_{|O|}\right)$$

2) update functions for language data, formally defined as:

$$\underline{u}_{ij}: I \cup LD \cup ID \rightarrow LD$$
$$\underline{u}_{ij}\left(id_1,\dots,id_{|I|},ld_1,\dots,ld_{|LD|},id_1,\dots,id_{|ID|}\right) = \left(ld_1,\dots,ld_{|LD|}\right)$$

3) update functions for internal data, formally defined as:

$$\underline{u}_{ij}: I \cup LD \cup ID \rightarrow ID$$
$$\underline{u}_{ij}\left(id_1,\dots,id_{|I|},ld_1,\dots,ld_{|LD|},id_1,\dots,id_{|ID|}\right) = \left(id_1,\dots,id_{|ID|}\right)$$

When an update propositional variable $\boldsymbol{u}_i$ is set to true, the corresponding update is activated, evaluating simultaneously all associated update functions.

A metamodel of the eISM language that contains all concepts discussed above, is illustrated in Figure 56.

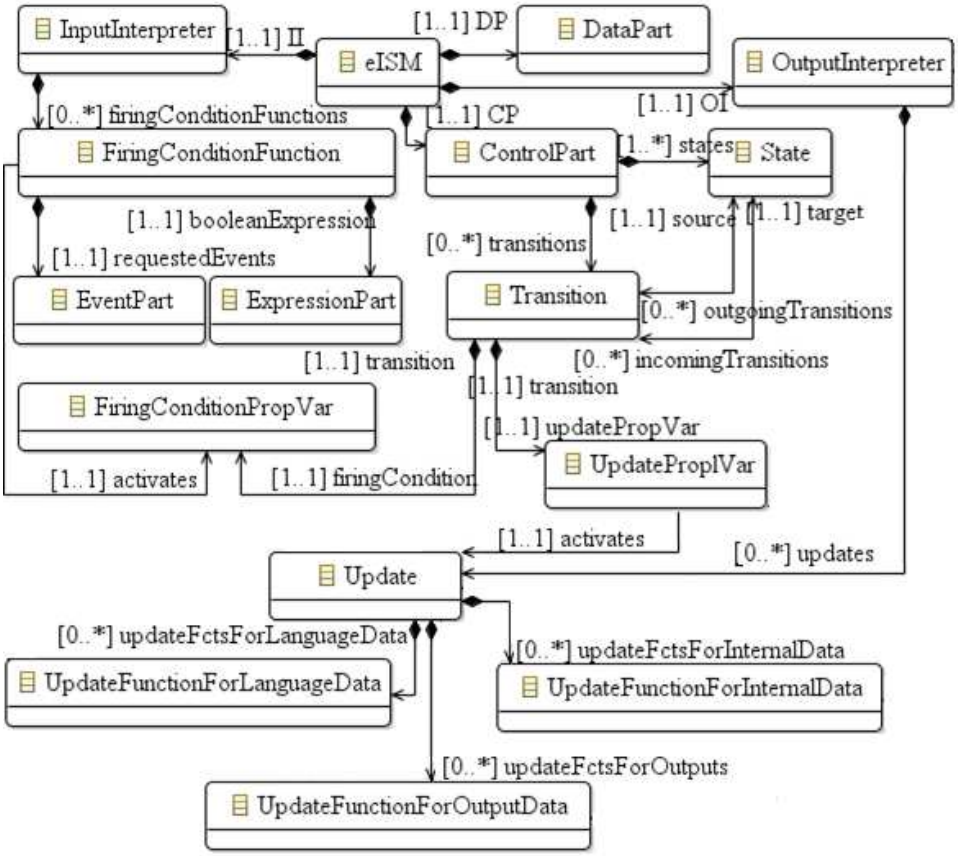


Figure 56. Metamodel of the eISM language.

4.2.3 Integrating the eISM and the metamodeling language EMOF

There are two possible way to relate the domain consents specified by the DDMM and their behavior specified as an eISM behavioral model:

1) By interfaces
2) By integrating eISM with the metamodeling language used to design the DDMM

In the first case, eISM behavioral modes don't have a direct access to the concepts' data. Therefore, the data part of eISM models must either be manually updated or by the means of transformations. In contrary, in the second case, eISM behavioral modes have direct access to the concepts' data. The relations between the concepts and eISM behavioral models are defined at M3 meta-meta layer as described below.

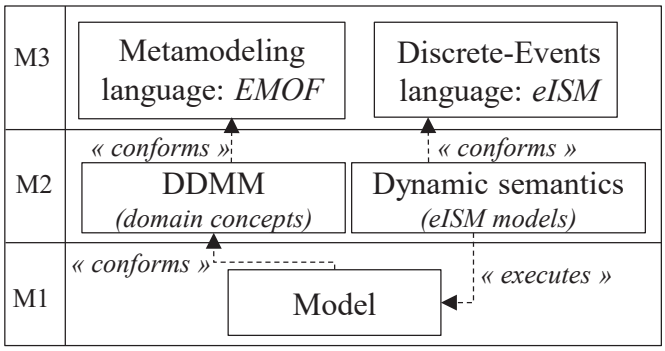


Figure 57. A metamodeling stack for executable DSMLs.

We focus here on the integration of eISM with the metamodeling language EMOF. EMOF is the EMF (Steinberg et al. 2008) version of the initially introduced MOF (OMG 2015a)). The goal is to design a M3 metamodeling layer that can be used for the creation of executable DSMLs as illustrated in Figure 57.

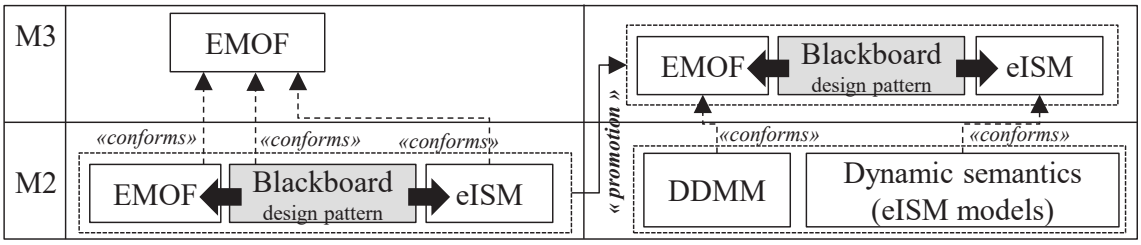


Figure 58. The integration process bounding a EMOF with eISM.

The process that allows the integration between EMOF and eISM (illustrated in Figure 58) is inspired by (Muller et al. 2005). It is composed of four steps:

- Step 1: model the eISM language
- Step 2: download EMOF to M2 layer
- Step 3: specify the dependencies between eISM and EMOF
- Step 4: promote the result at the M3 layer

The first step of modeling the eISM language is discussed above and illustrated in Figure 56. The second step consists in recovering the meta-metamodel of EMOF at M2 layer. This is a technical issue that is solved by the import option of EMF. The third step consists in establishing the relationships between EMOF and eISM. Note that, to address the previously discussed *Issue 5* (model interoperability in terms of behavioral dependencies and synchronized execution of ISM models) the integration between EMOF and eISM is established following the blackboard design pattern, proposed in (Engelmore & Morgan 1988). Chapter V provides more details on the blackboard

design pattern and on the synchronized execution of eISM models. Finally the resulting metamodel is promoted to the M3 layer, replacing the initial EMOF.

The resulting “executable” meta-metamodel is shown in Figure 59, integrating EMOF (in red) with eISM (in white) based on the blackboard design pattern (in gray). Note that already defined DSMLs that conform to the original EMOF remain fully compatible with this new executable version.

So, the communication between different types of behavioral models (among which are eISM behavioral models) is assured by the blackboard communication pattern that establishes the means for data of event exchange (see Chapter V for more details). However, two behavioral models can communicate if they have information about each other (i.e., the sender behavioral model must have information about the behavioral model that receives the message). For this purpose, the corresponding concepts of the behavioral models (defined by the bi-directional reference *behavioralmodel/concept* between *EClass* and *Behavioral Moedl* in Figure 59) must be structural bound together by a reference of a composition.

For example, a simple case scenario is illustrated in Figure 60 representing a telephone communication between two persons. When two persons make a call (1), the behavioral model of the caller should send an event to the behavioral model of the call receiver. If the latter respond (2), an event is send back to the caller and a communication is established (3).

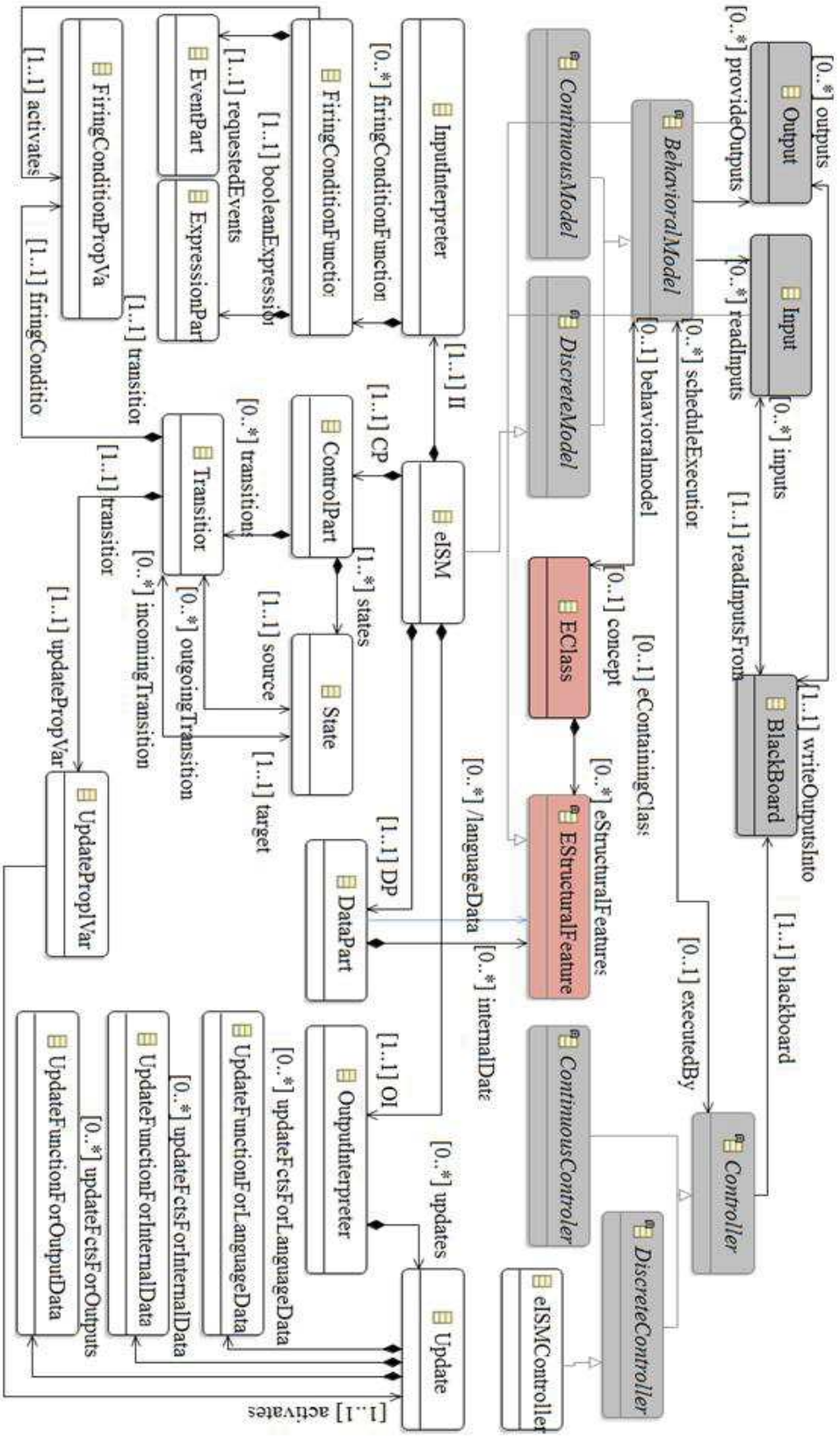


Figure 59. Integrating EMOF (in red) with eISM (in white) based on the blackboard design pattern (in gray).

A simplified domain structure (DDMM) is illustrated in the middle of Figure 60, composed of the *Person* concept and a bidirectional reference that highlights the caller and the call receiver. This reference is crucial for the behavioral communication, in this call a telephone call between two persons. The behavioral model from the right side of the Figure 60, contains information about the caller object and the receiver object. For instance, if the person *a* calls the person *b*, then the behavioral model of the person *a* have information about the receiver of the call in its DP, and as a result sends a message to the behavioral model of the person *b*.

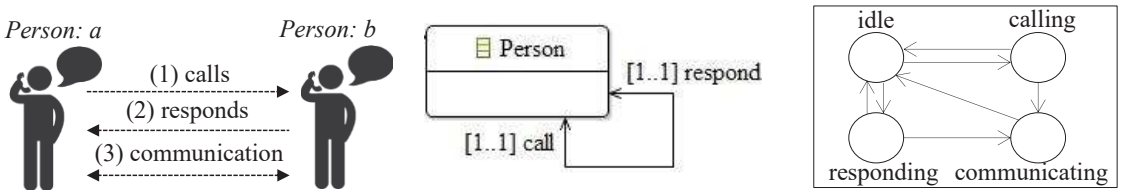


Figure 60. A model (left), a structure (middle) and a behavior (right).

So, the domain structure of a language DDMM (i.e., domain concepts, concepts' attributes and relationships between concepts) is in close relationship with its behavior and might sometimes directly influence the behavioral specification. For instance in the case of eISM, this includes the introduction of new states, transitions, firing conditions and update functions, modifying the control part (CP), input interpreter (II) and output interpreter (OI).

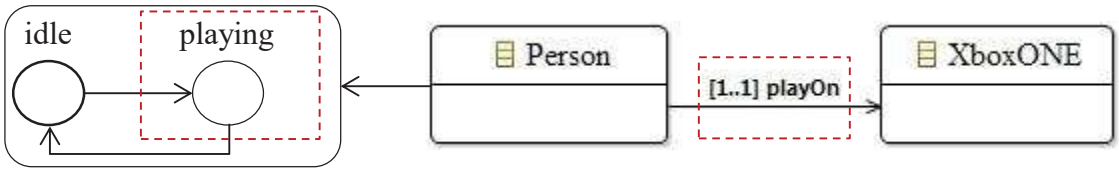


Figure 61. The structure impacts the number of states in a discrete-events behavioral model.

For instance in Figure 61, the presence of the reference *playOn* influence the presence of a new state *playing* into the behavior of a Person. This furthermore impacts on the *II* as a consequence to the need of firing conditions and on the *OI* as a consequence to the need of update functions.

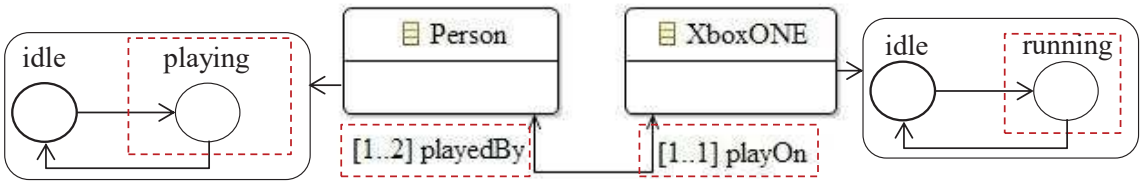


Figure 62. The structure impacts the synchronized functioning of behavioral models.

The structure might also influence the synchronized functioning of different behavioral models. For instance the example illustrated in Figure 62 shows the influence of a bi-directional reference on the behavior of its source concept (Person) and its target concept (XboxONE). In this case, if a person (instance of the class Person) plays on an Xbox One (instance of the class XboxOne), the console should be running (i.e. the person is in playing state and the XboxOne is in running state). Another example is the controlled execution of Functions in the eFFBD language. In this case, the execution of function is controlled by the container sequence, as discussed in Section 4.1.1 (see Figure 47). The container function must start the execution of composing functions and wait until all composing functions finish execution.

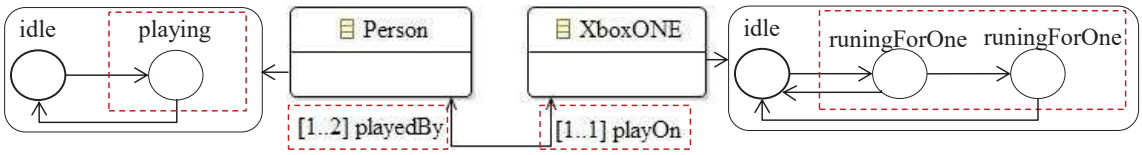


Figure 63. Multiplicity impacts the behavior.

Multiplicities might also impact on the behavior. For instance, in Figure 63 another "console playing" example is illustrated, where, depending on the number of persons that play on the same Xbox One, the console should be respectively in *idle*, *runningForOne* or *runningForTwo* state.

4.2.4 Technical issues related to the eISM

In addition to the integration process discussed in Section 4.2.3, the following technical issues still remain, preventing the design and management of eISM models:

- *Technical issue 1:* an editor for eISM does not exist
- *Technical issue 2:* the EMOF's editor (class/relation diagram) is not suited for eISM models

- *Technical issue 3:* the EMF code generator (*genmodel*) is not suited for eISM model (i.e., does not generated code for eISM models)

To address the first technical issue, we have designed an eISM graphical editor by using the Obeo Designer approach (Juliot & Benois 2010). For instance, this editor is used to design an eISM model for the eFFBD concept Function, as illustrated in Section 4.2.6 by Figure 65. The choice of the Obeo Designer approach is justified by the following points:

1) Obeo Designer is easy to use, not requiring significant tool-related knowledge;
2) supports a multi-viewpoint graphical representation;
3) is integrated into the EMF and is compliant with the EMOF;
4) has a tool-supported release that is open source, currently available for download, maintained and regularly updated.

The second issue is about the management of eISM models (i.e., their design and their association with domain concepts modeled by classes) by using the graphical EMOF editor (i.e., the EMF's class/relation diagram). Namely, first eISM models must be created and graphically represented in the EMOF editor. Second, eISM models must be association with domain concepts and such associations must also be graphically represented.

We have addressed this issue by extending the initial EMOF editor, including the above quoted features. As illustration, the EMOF editor shown in Figure 69, illustrates several classes related to each other by references and compositions, but also, related to red-oval forms that represent eISM models. A double-click on these red-ovals opens the eISM editor, discussed in *Technical issue 1*, and allows designing an eISM model.

The third technical issue is about the EMF's code generation mechanism represented by a so-called *genmodel*. The genmodel allows generating Java interfaces and implementation classes for all the classes shown in an EMOF editor, plus a factory and package implementation class. However, the genmodel is not suited to generate code for the eISM models shown in the EMOF editor. At this point, eISM models are designed and graphically represented in the eISM editor, but they lack the necessary Java code (similarly to the EMOF classes and relations before the code generation). For instance, let's consider the eISM model for the eFFBD concept Function, illustrated in Section

4.2.6 by Figure 65. For this model, we need one instance of the class eISM (see Figure 56), six instances of the class State (see Figure 56) for each of the states and so on.

We have addressed this issue by extending the initial *genmodel* to generate (in addition to the Java interfaces and implementation classes for all the classes shown in an EMOF editor) the necessary Java code for all the eISM models shown in an EMOF editor.

4.2.5 A formal proof mechanism for the eISM

The formal proof mechanism proposed in this section allows formal verification of properties based on eISM behavioral models. The goal is on the one hand to verify the well-formedness of eISM behavioral models before using them for the purpose of simulation, and on the other hand, to verify properties during simulation. The verification must be performed taking eISM models separately but also together with other eISM models (from the same DSML or other DSML from the same modeling environment).

A verification process consists in general of: 1) a formal specification, on which the verification process is conducted, 2) formal properties that are verified on the formal specification during the verification process and 3) a tool for verification, i.e., a model-checking tool.

1) Formal specification

The underlying structure of an eISM behavioral model is based on the Linear Temporal Logic (LTL), defined by a set of *Elementary Valid Formulas (EVF)* that are initially introduced in (Larnac et al. 1995) as follows.

EVF are inferred from the *PC*'s transitions combined with LTL operators. Let $T_i = \left[(\boldsymbol{s}_i, \boldsymbol{e}_i), (\boldsymbol{s}_j, \boldsymbol{u}_i)\right]$ a transition between states $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$, associated to an $\boldsymbol{e}_j$ firing condition propositional variable and to a $\boldsymbol{u}_i$ update propositional variable (see Figure 53). T_i infers as an *EVF* of the following form:

$$EVF(T_i) := \square\left(\boldsymbol{s}_i \wedge \boldsymbol{e}_i \supset \bigcirc \boldsymbol{s}_j \wedge \boldsymbol{u}_i\right)$$

Its interpretation stands as follows: "it is always true ($\square$ operator) that if $\boldsymbol{s}_i$ is the current state (and therefore $\boldsymbol{s}_i$ is *true*) and $\boldsymbol{e}_j$ is *true*, then the next state ($\bigcirc$ operator) will be $\boldsymbol{s}_j$ ($\boldsymbol{s}_j$ will be *true*), and the current output propositional variable $\boldsymbol{u}_i$ becomes *true*". The list of

all the *EVFs* gives a symbolic and equivalent description of the behavior of an eISM model.

Similarly, a *Unified Valid Formula* (*UVF*) is computed by taking *EVFs* into consideration. Briefly, the concept of *Temporal Event (E_t)* describes possible effects of an eISM model evolution. E_t can either be a future state (E_t=○$\boldsymbol{s}_i$), a future state within n-time steps (E_t=○$^n\boldsymbol{s}_i$), a future output propositional variable (E_t=○$\boldsymbol{u}_i$), or a future output propositional variable within n-future steps (E_t=○$^n\boldsymbol{u}_i$). A *Unified Valid Formula (UVF)* defines then conditions that must be satisfied for the occurrence of a temporal event E_t:

$$UVF(E_t) := \bigvee_{(p,q)/\boldsymbol{s}_p \wedge \boldsymbol{e}_q \supset E_t} (\boldsymbol{s}_p \wedge \boldsymbol{e}_q)$$

Its interpretation stands as follows: "next temporal event E_t (respectively state S_j or update function $\underline{u_j}$) is reachable if and only if at least one of the proposed conditions is verified". So the calculation of *UVFs* consists in manipulating the set of EVFs.

$$T_k = [(\boldsymbol{s}_k, \boldsymbol{e}_k), (\boldsymbol{s}_j, \boldsymbol{u}_k)]$$
$$T_l = [(\boldsymbol{s}_l, \boldsymbol{e}_l), (\boldsymbol{s}_j, \boldsymbol{u}_l)]$$

Figure 64. An example of a state model with three states (S_k, S_l and S_j) and two transitions (T_k and T_l).

For instance, let's consider the following *EVF* formulas, derived from the Figure 64 state model:

$$(1)\ EVF(T_k) := \square(\boldsymbol{s}_k \wedge \boldsymbol{e}_k \supset \bigcirc \boldsymbol{s}_j \wedge \boldsymbol{u}_k)$$

$$(2)\ EVF(T_l) := \square(\boldsymbol{s}_l \wedge \boldsymbol{e}_l \supset \bigcirc \boldsymbol{s}_j \wedge \boldsymbol{u}_l)$$

The *UVF(E_t)* when E_t = ○$\mathbf{s}_j$ is then noted:

$$UVF(E_t) := (\boldsymbol{s}_k \wedge \boldsymbol{e}_k) \vee (\boldsymbol{s}_l \wedge \boldsymbol{e}_l)$$

whose interpretation is: "$\mathbf{s}_j$ will be active in the next step (○$\mathbf{s}_j$ is true), either if $(\boldsymbol{s}_k \wedge \boldsymbol{e}_k)$ is true or if $(\boldsymbol{s}_l \wedge \boldsymbol{e}_l)$ is true".

2) Formal properties

Chapter III introduces different types of properties, among which are the constraint properties (CP, see *Definition 10*). This section focuses particularly on the following constraint properties:

- A-temporal behavioral constraint properties (ABCP), see *Definition 18*
- A-temporal dependent behavioral constraint properties (ADBCP), see *Definition 39*
- Temporal behavioral constraint properties (TBCP), see *Definition 19*
- Temporal dependent behavioral constraint properties (TDBCP), see *Definition 39*

Namely, the A(D)BCP are used to verify the well-formedness of the behavior models before being used for the purpose of simulation, specifying:

- *The hypotheses of the used behavioral language*: the behavioral language imposes several hypotheses that designed behavioral models must respect.
- *Alternative or Stakeholders' hypotheses*: sometimes stakeholders impose, in addition to the hypotheses of a behavioral language, several other hypotheses.

The T(D)BCP are used to verify the eISM models during simulation. For this purpose, a model checked must be integrated with a simulator, as proposed for instance by UPPAAL. For instance, the following property must be verified every execution step by all eISM behavioral models:

"at a given time step, there is one and only one current state".

Both temporal and a-temporal properties must be formalized by using the LTL. For instance, the above quoted property is specified by the following LTL formula:

$$P_1 := \square\left(\boldsymbol{s}_i \supset \neg \boldsymbol{s}_j\right), \forall i, j \in \{1, .., |States|\}, \qquad i \neq j$$

3) Tool

An adequate model checking tool is under construction considering the Rozier's survey on formal verification techniques of LTL symbolic model checking (Rozier 2011).

As an example of LTL formulas checking mechanisms for the ISM, (Larnac et al. 1995; Vandermeulen et al. 1995; Vandermeulen 1996) propose the *Temporal Boolean Difference (TBD)* mechanism inspired by (Kohavi & Jha 2009).

The TBD mechanism is applied on a *UVF* with respect to a current state or a firing condition propositional variable, composing them into a *Derived Valid Formula (DVF)*:

$$DVF(E_t, x) := \frac{\partial UVF(E_t)}{\partial x} = UVF(E_t|x) \oplus UVF(E_t|\neg x)$$

The result of an evaluation of $DVF(E_t, x)$ can either be:

i. *False – UVF(E_t)* is independent of *x*. In other words, the change of value of *x* has no influence over the occurrence of E_t.

ii. *Not False* – in this case, we obtain a LTL formula which expresses the sensitivity of *UVF(Et)* with respect to the changes of *x*.

In summary, the proof mechanism proposed above aims at "direct" verification of LTL properties based on the elementary valid formulas (EVF) abstracted from eISM models, without transforming the eISM models into third-party formalisms. An adequate model checking tool is under construction. The model-checker must be able to consider multiple eISM models abstracted through EVFs for the verification of "dependency" properties. We aim at integrating this model-checker with a simulator for the verification of temporal properties.

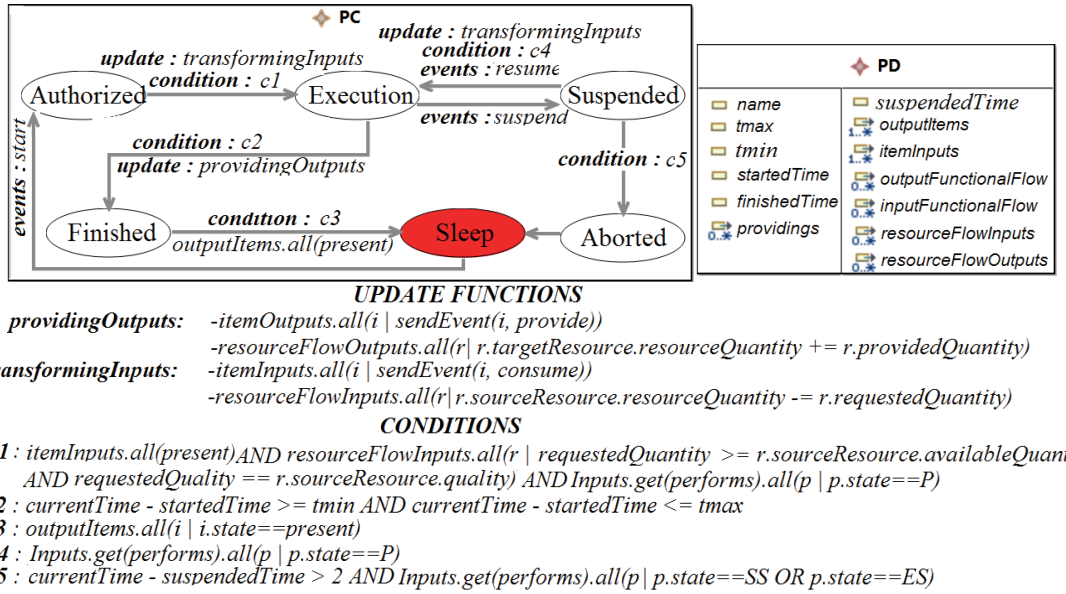
4.2.6 Example 1: modelling the behavior of the eFFBD concept Function

We show in this section the design of the behavior of the eFFBD concept Function by using the eISM language.

The behavior of the eFFBD concept function is described in Section 4.1.1 as a six-state behavioral model composed of the following states: *Sleep*, *Authorized*, *Execution*, *Finished*, *Suspended* and *Aborted*.

The corresponding eISM behavioral model is illustrated in Figure 65 and is described as follows. A Function is initially in the Sleep state, waiting for a request to start execution (*start* event). When the request arrives, the Function enter *Authorized* state, meaning that that input/output transformation is possible depending on the availability of all input Items and Resources as well as the state of the Components on which the Function is allocated (*condition : c1*). When the previous condition is satisfied, the update *transformingInputs* is activated (i.e. the real transformation of energy, material and / or data happens) and the Functions enters *Execution* state. The transformation least a certain time period (*condition: c2*), before producing outputs (*update:*

providingOutputs) forcing the Function into *Finished* state. In case of dysfunction of the component on which the function has been allocated (*suspended* event), a function is *Suspended* and eventually *Aborted*, assuming the component does not reply on time (*condition: c5*).



UPDATE FUNCTIONS

providingOutputs: *-itemOutputs.all(i | sendEvent(i, provide))*
-resourceFlowOutputs.all(r| r.targetResource.resourceQuantity += r.providedQuantity)

transformingInputs: *-itemInputs.all(i | sendEvent(i, consume))*
-resourceFlowInputs.all(r| r.sourceResource.resourceQuantity -= r.requestedQuantity)

CONDITIONS

c1 *: itemInputs.all(present) AND resourceFlowInputs.all(r | requestedQuantity >= r.sourceResource.availableQuantity AND requestedQuality == r.sourceResource.quality) AND Inputs.get(performs).all(p | p.state==P)*
c2 *: currentTime - startedTime >= tmin AND currentTime - startedTime <= tmax*
c3 *: outputItems.all(i | i.state==present)*
c4 *: Inputs.get(performs).all(p | p.state==P)*
c5 *: currentTime - suspendedTime > 2 AND Inputs.get(performs).all(p | p.state==SS OR p.state==ES)*

Figure 65. An eISM behavioral model describing the behavior of the concept Function.

To complete the behavior of the concept Function, we propose in the next, to model the behavior of the concept Component of the PBD language, based on eISM. This behavior is initially introduced in Section 3.3 and illustrated in Figure 32 as a a five-state behavioral model.

The corresponding eISM behavioral model is illustrated in Figure 66 and is described as follows. A component is initially non-active (*NA*) waiting for energy (*activate* event) to get prepared for a state. When the signal is received, the update *activating* is activated and the component enters activates state (*A*). It starts producing, when the *start* signal is received, activating the update *producing* (i.e. the component performs its function) and it enters producing state (*P*). Components perform their functions until they receive, either a *stop* signal, which put them in the previous state (update *stopping* is activated), or a *breakdown* signal (update *emergency* is activated), which immediately makes them stop producing and puts them in waiting states (*SS* or *ES*) depending on the signal nature (*internal default* or *external default*). Additionally, a component provides its performing functions with its current state (see the *notify* update), allowing them to take

the component's current state into account inside their behavioral model (see Function's conditions).

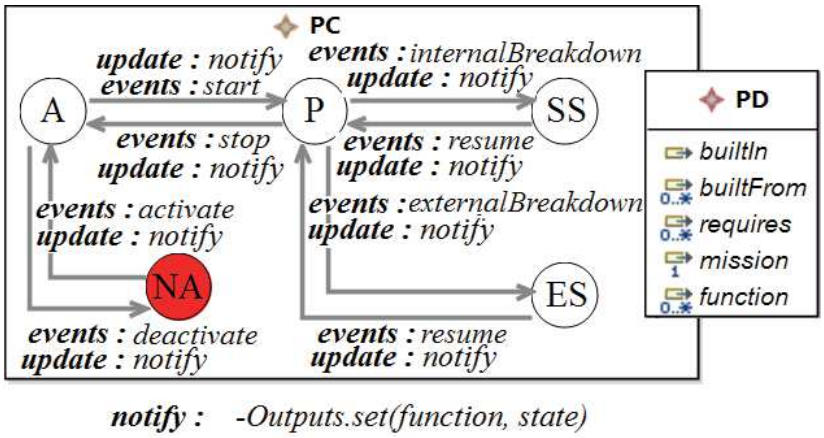


Figure 66. An eISM behavioral model describing the behavior of the concept Compàonent.

4.2.7 Example 2: executable WaterDistrib DSML

In this section we demonstrate a from-scratch design of an executable DSML for modeling water storage and distribution systems, denoted WaterDistrib (initially introduced in Section 2.4.3).

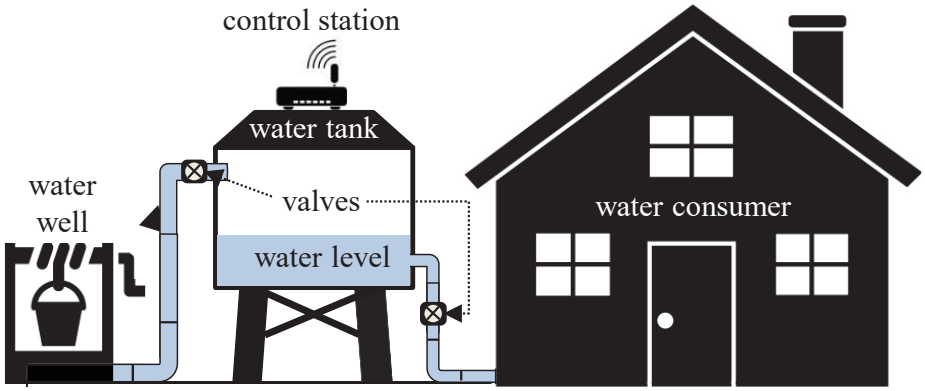


Figure 67. a WaterDistrib model – an example of a water storage and distribution system.

A model created by WaterDistrib is illustrated in Figure 67. It is composed of a water tank, a water-source that is connected to the tank with pipes and a control station. A house is supplied with water thanks to the tank. There are valves on each of the pipes, controlled (opened or closed) by a control station, based on the water request and the

water level inside the tank. The goal of this case study is to observe the changing water level in the tank based on the consumers demand.

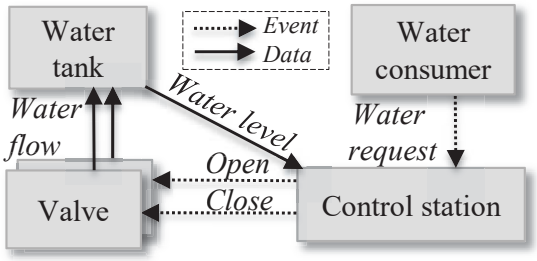


Figure 68. Imagined functioning of WaterDistrib.

The imagined functioning of this system is illustrated in Figure 68 as described in the next. Note that, the purpose of this schema is to illustrate the exchange of information i.e., data or signals (events) between different components. The control station monitors the *water level* inside the tank. It responds to a *water request* from the house, based on the tank's current water level and the tank's allowed minimal or maximal water level. As a result, the control station sends *Open* or *Close* signals (events) to valves, changing their state that consequently impacts on the volume of *water flow* they provide, through pipes, to the tank. Finally, the water level of the tank varies depending on the incoming and outgoing water flow.

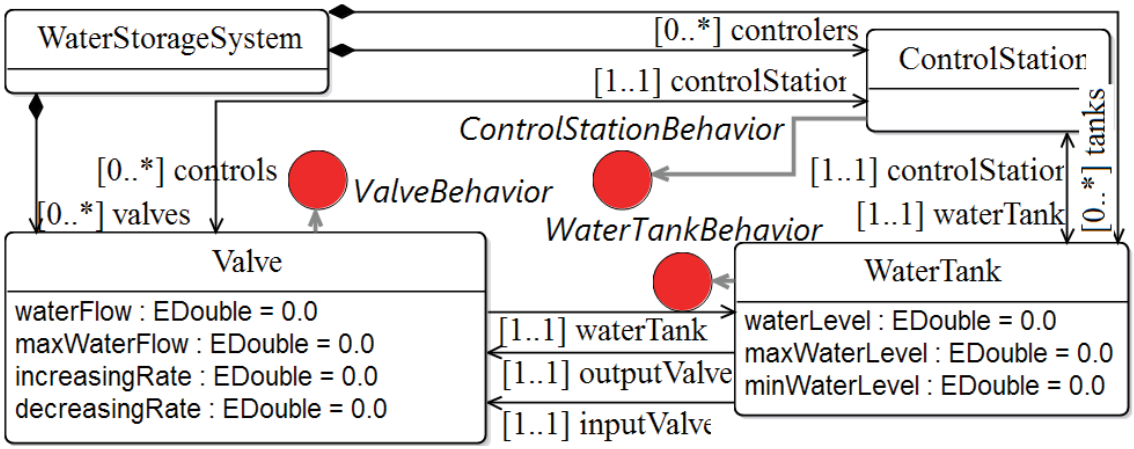


Figure 69. WaterDistrib: a new DSML for a water storage and distribution systems.

The metamodel of WaterDistrib is illustrated in Figure 69 composed of three principle components: *WaterTank*, *Valve* and *ControlStation*. We design hereafter the behavior of each concept by eISM behavioral models, considering the previously imagined functioning.

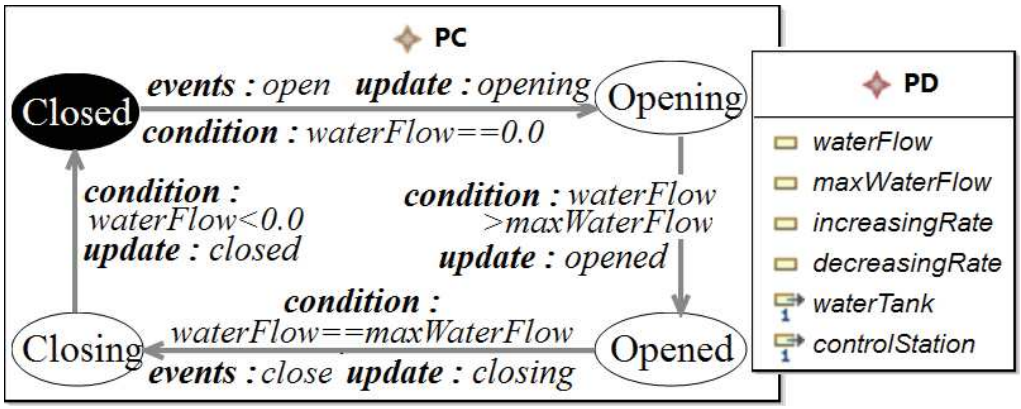


Figure 70. eISM behavioral model associated to the class Valve.

The behavior of the concept *Valve* is composed of four states: *Closed*, *Opening*, *Opened* and *Closing* as illustrated in Figure 70.

Table 1. Valve's updates

Update	***Language Data***
closed	*waterFlow=0*
opening	*waterFlow+=increasingRate*
opened	*waterFlow=maxWaterFlow*
closing	*waterFlow-=decreasingRate*

A valve is initially *Closed*, not providing any water flow (update *closed* is activated, see Table 1), awaiting a request to open itself. When the *open* request arrives, the update *opening* is activated (see table 1) and the valve enters *Opening* state. Once the valve's water flow reaches its maximum value, the update *open* is activated (see Table 1) and the valve enters *Opened* state. Now the valve awaits a request to close itself. When the *close* request arrives, the update *closing* is activated (see Table 1) and the valve enters *Closing* state. As soon as the valve's water flow reaches 0, the update *closed* is activated and the valve enters its initial *Closed* state.

The behavior of the concept *ControlStation* is composed of three states: *Mode1*, *Mode2* and *Mode3* as illustrated in Figure 71.

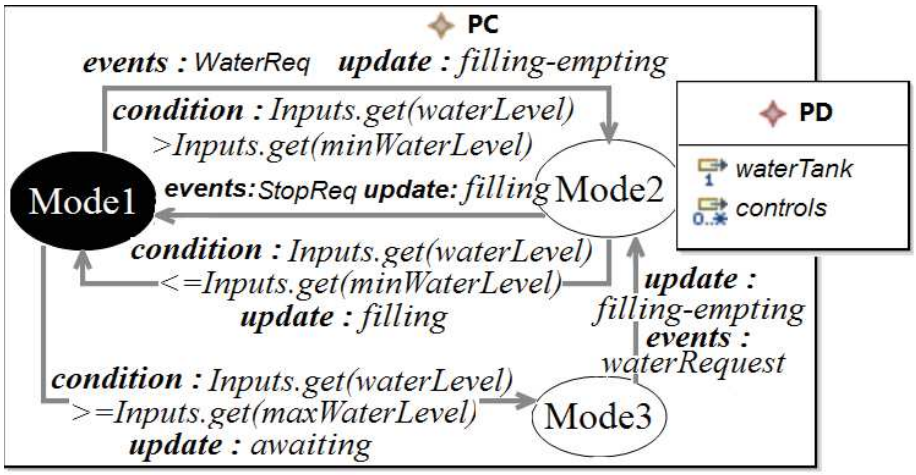


Figure 71. eISM behavioral model associated to the class ControlStation.

A control station is initially in the *Mode1* state, filling the tank (update *filling* is activated, see Table 2) awaiting water request. When the request arrives and if there is a sufficient water level in the tank, the *filling-empting* update is activated (see Table 2) and the control station enters *Mode2* state. If the tank is empting faster than filling, when its current water level reaches the critical min level, or if a "Stop Water Providing Request" is received, the control station enters again *Mode1* state, activating the *filling* update.

Table 2. Control Station's updates

Update	***Output Data***
filling	*Outputs.set(waterTank.inputValve, Open)* *Outputs.set(waterTank.outputValve, Close)*
filling-empting	*Outputs.set(waterTank.inputValve, Open)* *Outputs.set(waterTank.outputValve, Open)*
awaiting	*Outputs.set(waterTank.inputValve, Close)* *Outputs.set(waterTank.outputValve, Close)*

For the sake of simplicity, the case when the tank is filling faster than empting is not modeled in Figure 71. When the station is in *Mode1* state, if a water request has not yet arrived and the tank reaches its critical max level, the *awaiting* update is activated (see Table 2). The control station enters *Mode3* state, waiting for a water request. The

request arrival activates the *filling-empting* update and the control station enters *Mode2* state.

The eISM behavioral model associated to the class Water Tank should be a continuous behavioral models. However, at the current stage of this research, continuous behavioral models are out of the scope. Therefore, it is represented by a one-state eISM model that has two, always active, update functions. The function that increases the tank's water level based on on the incoming water flow:

- waterLevel+=Inputs.get(inputValve,waterLevel)

and the function that decreases the tank's water level based on on the outgoing water flow:

- waterLevel-=Inputs.get(outputValue,waterLevel)

Additionally, the tank provides information to the control station about its current, minimal allowed and maximal allowed water level by the following update functions:

- Outputs.set(controlStation,waterLevel)
- Outputs.set(controlStation,maxWaterLevel) and
- Outputs.set(controlStation,minWaterLevel)

The next phase consists to formally verify for well-formedness of previously designed eISM behavioral models. For this purpose, their formal underlying structure is developed and exploited.

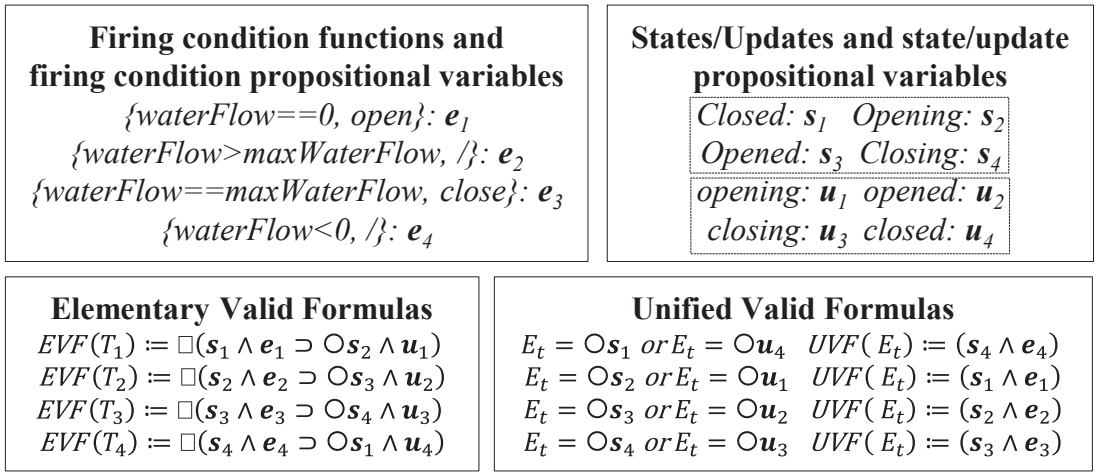


Figure 72. The formal underlying structure of the Valve's eISM behavioral model

Figure 72 illustrates the formal underlying structure of the Valve's eISM behavioral

model. At the upper side of the figure the states, updates and firing conditions are specified, along with their corresponding propositional variables. Using these variables allows the specification of EVFs that are furthermore used for the specification of the UVFs. In the same way, one can specify the formal underlying structure of any eISM model.

Concerning formal properties, let's consider the transition exclusion hypothesis: "at any given time step, for the current active state (which must be unique), there is one and only one output transition that can be fired". In other word, all firing condition of output transitions of any state from the PC, are to be exclusive, modelled as:

$$\forall S_i \in S, E_{s_i} = \{e_j | \forall T_k \in post(S_i), pre(FVE(T_k) = S_i \wedge e_j)\} \left| \oplus_{j=1/card(E_{S_i})} \ e_j = 0 \right|$$

Finally, an adequate model-checker should be used to verify this property on the formal specification.

4.3 Modeling the behavior of a DSML with a formal rule-based language

This section proposes a Formal Rule-Based Language (FRBL) to ease and assist the design of dynamic semantics as much as possible for discrete-events (DE) languages with pre-defined semantics such as eISM. A discussion about the positioning and problematic is proposed in Section 4.3.1. Introduction to the FRBL, its syntax and semantics are proposed in Section 4.3.2, 4.3.3 and 4.3.4. An example is shown in Section 4.3.5. and Section 4.3.6 introduces an approach for "on the fly design and integration" of new discrete-events languages with the EMOF.

4.3.1 Positioning and Problematic: DSMLs with predefined formal semantics

We have previously shown how to model the dynamic semantics of a DSML by a set of discrete-events behavioral models designed by using the discrete-events language eISM. Following the design is the execution of models created by a DSML, using the eISM behavioral models. The eISM behavioral models are executed based on the dynamic semantics of the eISM language. Namely, the eISM language has a syntax (abstract and concrete) but also a semantics (static and dynamic). For instance, its abstract syntax is shown in Figure 56 and its concrete syntax is illustrated for the examples shown in

Figure 65, Figure 70 and Figure 71. The semantics of eISM is implicitly but partially defined by the formal specification introduced in Section 4.2.2. Nonetheless, such formal specification can only be used to understand the functioning of eISM models. For the execution of eISM models, an adequate implementation of the formal specification is needed, for instance, designed by using the action language Kermeta, as discussed in Chapter II.

Similarly to eISM, there are various other languages with formal predefined semantics, e.g., PetriNets, Statechart, Finite State Machine, FCCS, etc. Some of them have even various semantics that might be considered valid and usable. In this section, a particular attention is given on making such languages executable, by easing and assisting the design process of dynamic semantics as much as possible. For this purpose, we aim at reusing the formal pre-defined semantics, rather than completely rewriting and rethinking it. This idea is inspired by the boilerplate-based approaches (see Chapter II) where models are built on the top of templates that contain crucial, already validated information, providing a solid basis. We argue that this can considerably reduce the needed efforts and time for the design of dynamic semantics for DSML with formal pre-defined semantics, such as eISM, Statechart, FCCS, etc.

4.3.2 General introduction to the FRBL

This section introduces the Formal Rule-Based Language (FRBL). FRBL is used for the design of dynamic semantics of DSMLs with formal pre-defined semantics through formal expressions, denoted rules, mixed with classical control flow (conditional, iterative, rule calls, etc.). The goal of FRBL is to assist and ease the design of dynamic semantics for a particular category of DSMLs that have formal pre-defined semantics based on discrete-events (DE) hypothesis. For this purpose, FRBL is based on two principles, mentioned above and detailed hereafter:

- Reuse of the predefined formal semantics of DE languages
- Design based on templates

According to (Chapurlat 1994), the evolution of any DE model can be generalized based on three phases, illustrated in Figure 73:

- Phase 1 - Reading Inputs (RI)
- Phase 2 - Calculating Future State(s) (CFS)

- Phase 3 - Writing Outputs (WO)

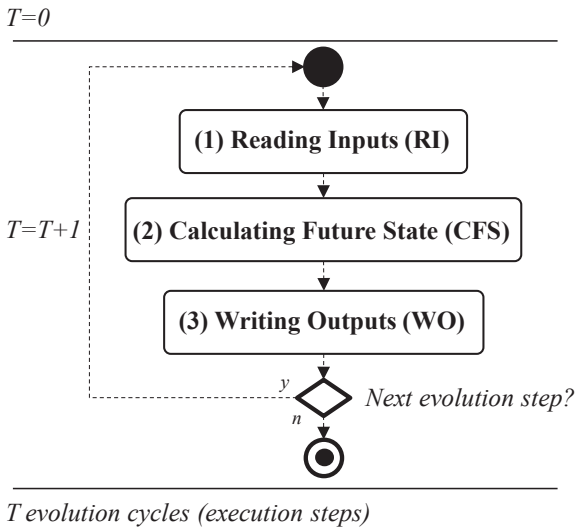


Figure 73. Generic evolution algorithm for discrete-events models (Chapurlat 1994).

During phase 1, DE models must read the requested inputs, forming the necessary data to evolve. Inputs are provided by an external source, for instance, by the environment. Next, DE models must calculate their future state based on the dynamic semantics of the DE language that is used to create them (e.g., the future state of PetriNets models determined by the number of tokens in places, is calculated based on the dynamic semantics of the PetriNets language). During this process, the data provided by the inputs is potentially changed. Finally, the data is provided back to the external source through the writing outputs phase.

Based on the above presented generalization of DE behaviors, the FRBL language proposes a generic template that can be reused for any DE language based on three main rules for each one of the above quoted phases: 1) Reading Inputs Rule, 2) Calculating Future State Rule and 3) Writing Outputs Rule.

To furthermore ease the process of dynamic semantics design, designers need to consider the formal pre-defined semantics of the DE language they are designing, to complete the template, creating a fully functional dynamic semantics that can be used for DE models execution.

The syntax of the FRBL is designed to be similar to formal semantics, easing the reuse of the pre-defined formal semantics of DE languages.

We propose in the next, the syntax (abstract and concrete) and the dynamic semantics of the FRBL. The syntax is designed in a form of xText grammar (similar to EBNF) by the xText approach (Bettini 2013). To ease readability, we show in parallel to the EBNF rules, the corresponding abstract syntax through a metamodel. The dynamic semantics are designed in the form of transformations to Java code (i.e., Java code generator) that can be executed on the JVM (Java Virtual Machine).

4.3.3 Introduction to the syntax of the FRBL

This section presents the syntax of the FRBL language as an xText grammar, specified throughout Listing 1 – Listing 4.

Listing 1 is shown below. It is described as follows.

```
Behavior: rules+=Rule*;

Rule:
      '[rule' name=Name ('parameters:' parameters+=VarDeclaration*)?
                              ('output' returnType=(VARTYPE|SET))? ']'
            (expressions+=Expression)*
      '[/rule]';
```

Listing 1. xText grammar for FRBL rules.

A *Behavior* consists of an arbitrary number (*) of *Rules*.

Each *Rule* is marked by the tags "*[rule]*" that contains a "*rule declaration*", and "*[/rule]*". A "*rule declaration*" is composed of a name (i.e., the name of the rule) and optionally (?) of a "*set of parameters*" and a "*return type*". The "*set of parameters*" is preceded by the keyword "*parameters*" and it contains an arbitrary number of parameters, each one being a *Variable Declaration* (defined below). The "*return type*" is preceded by the keyword "*output*" and can either be a VARTYPE of SET (defined below). Inside the tags is a set of *Expressions* that define the body of the rule.

The above quoted description of Listing 1 is also modeled by the metamodel shown in Figure 74.

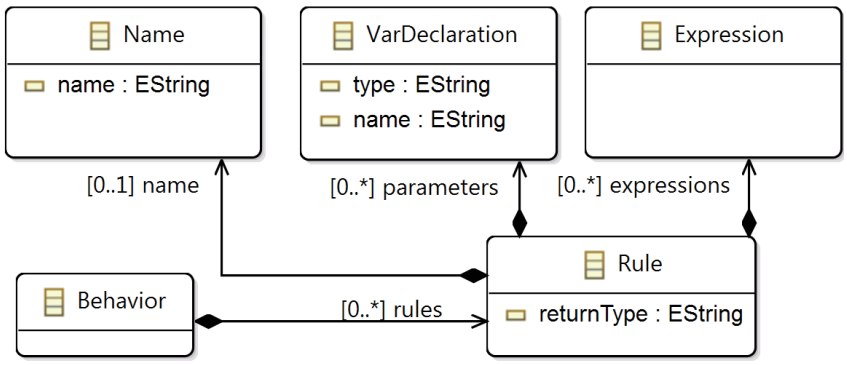


Figure 74. The metamodel describing the FRBL rules shown in Listing 1.

Listing 2 is shown below. It is described as follows.

```
Expression: ArithmeticExp | VarExp | CondExp | SetExp;

ArithmeticExp: value=Literal (RHS=BinaryExp)?;

VarExp: VarDeclaration | VarAssignment;

CondExp: '[if' condition= ArithmeticExp ']' (ifBody+=Expression)* '[/if]';

SetExp: '[forall' var=Name 'in' set=Name ']' (setBody+=Expression)* '[/forall]';
```

Listing 2. xText grammar for FRBL expressions.

There are four types of *Expressions*:

- Arithmetic Expressions
- Variable Expressions,
- Conditional Expressions
- Set Expressions

An *Arithmetic Expression* is composed of a *Literal* and an optional (?) right hand side that when defined, makes the *Arithmetic Expression* a *Binary Expression* (defined below).

A Variable Expression can either be a Variable Declaration or a Variable Assignment (both defined below).

A *Conditional Expression* is marked with the tags "*[if]*" that contains an "*if condition*" ,and "*[/if]*". The "*if condition*" is an *Arithmetic Expression* that can evaluate to 0, meaning that the condition is false, or any other number, meaning the condition is true. Inside the tags is a set of *Expressions* that define the body of the conditional expression. These expressions are evaluated only if the "*if condition*" is true.

A *Set Expression* is marked with the tags "*[forall]*" that contains a "*set declaration*", and "*[/forall]*". The "*set declaration*" is composed of an iterative variable over a set, both identified by a *Name* (defined below). Inside the tags is a set of *Expressions* that define the body of the set expression. These expressions are evaluated every time the variable iterates over the set.

The above quoted description of Listing 2 is also modeled by the metamodel shown in Figure 75.

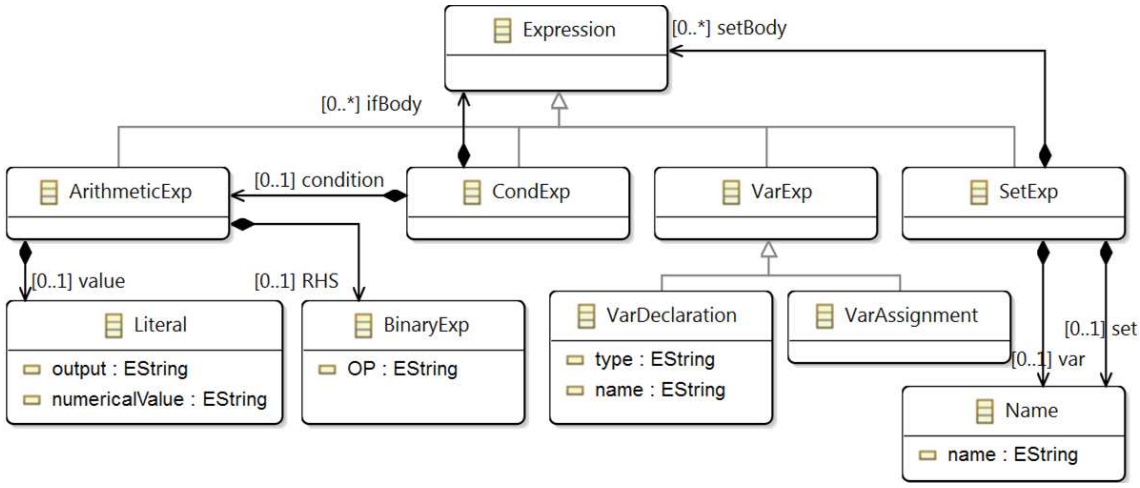


Figure 75. The metamodel describing the FRBL expressions shown in Listing 2.

Listing 3 is shown below. It is described as follows.

```
VarDeclaration: type=(VARTYPE|SET|ID) name=ID (':' defaultValue=Literal)?;

VarAssignment: varName=Name ':=' arithmeticExp=ArithmeticExp;

BinaryExp:
      OP=('+'|'-'|'*'|'/'|'<'|'>'|'>='|'<='|'='|'!='|'and'|'or') ae=ArithmeticExp;

Literal:
      (output = 'output')?
      numericalValue=Number |
      nameValue=Name;
```

Listing 3. The xText grammar for Variables, Binary Expressions and Literals.

A *Variable Declaration* is composed of a *type* that can either be a VARTYPE, a SET (defined below) or a unique identifier (ID), a variable name that must be unique (ID) and an optional (?) default value defined by a *Literal*.

A *Variable Assignment* is composed of the name of a variable, followed by the assignment keyword "*:=*" and an *Arithmetic Expression*.

A *Binary Expression* specifies the optional right hand side of an *Arithmetic Expression* (defined above). It is composed of a *binary operator* followed by another *Arithmetic Expression*. Note that for the sake of simplicity, *Logical Expressions* are specified as *Arithmetic Expressions* and thus among the *binary operators* are the comparative operators (>; <; >=; <=; =; !=) and the logical operators (and; or).

A *Literal* specified a *numerical value* (*Number*) or a *name value* (*Name*) (defined above). Optionally, the literal might be the result (i.e., the output) of a rule if preceded by the keyword "output".

The above quoted description of Listing 3 is also modeled by the metamodel shown in Figure 76.

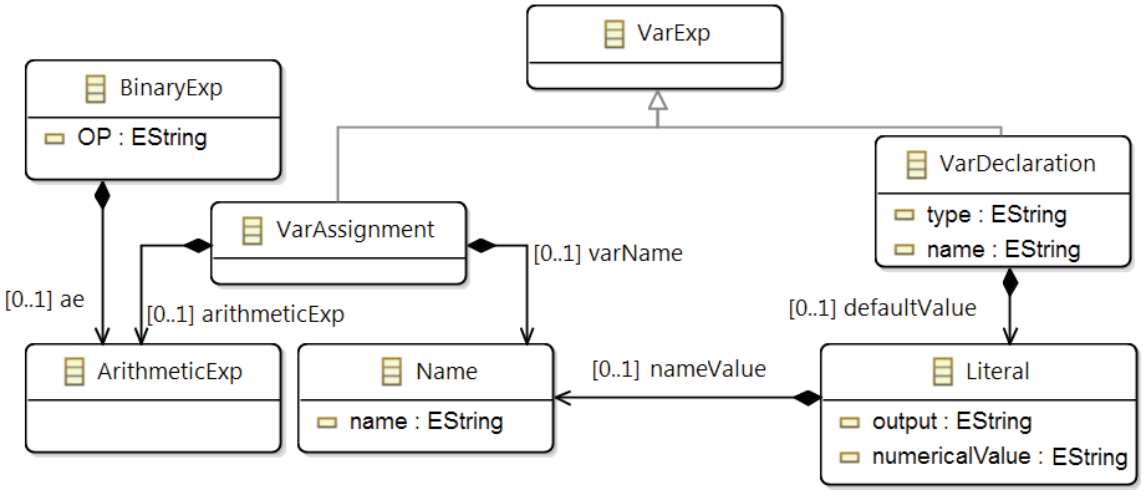


Figure 76. The metamodel describing the FRBL Variables, Binary expressions and Literals shown in Listing 3.

Listing 4 is shown below. It is described as follows.

A *Name* is either a simple STRING or a "*navigable entity*". A "*navigable entity*" is composed of a unique identifier (ID) and an arbitrary number (*) of *Navigations*.

A *Navigation* is composed of a *connector* followed by a unique identifier (ID) and optionally a *Predicate*. A *connector* might either be "." (used when navigating to an element) or "->" (used when navigating to a set of elements).

A *Predicate* is composed of an opening parenthesis "(" and a closing parenthesis ")" that regroup one or several *parameters* specified by unique identifiers (*ID*) and separated by the separator ",".

A *Number* is either a *whole number* (INT) or a *decimal number* specified as two whole numbers separated by a "." separator.

```
Name:
	STRING |
	(name=ID navigations+=Navigation*);

Navigation: connector=('.'|'->') name=ID (predicat=Predicat)?;

Predicat: leftP='(' (parameter=ID)? (',' additionalParameters+=ID)* rightP=')';

Number hidden():
	INT ('.' INT)?;

terminal VARTYPE: 'Integer'|'Float'|'String'|'Boolean';
terminal SET: 'Set' '<' VARTYPE '>';
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING:
	'"' ('\\' ./*('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\')*/| !('\\'|'"') )* '"'? |
	"'" ('\\' ./*('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\')*/| !('\\'|"'") )* "'"?;
```

Listing 4. The xText grammar for Name, Number and Terminals.

There are four terminals: VARTYPE, SET, INT and STRING. VARTYPE is for the declaration of types. SET is for the declaration of a SET. INT is for the specification of whole numbers. STRING is for the specification of string values. A string value must be framed into simple or double quotes.

The above quoted description of Listing 4 is also modeled by the metamodel shown in Figure 77.

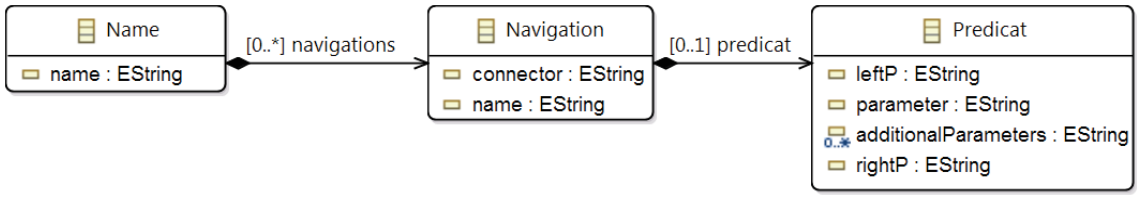


Figure 77. The metamodel describing the FRBL Name shown in Listing 4.

4.3.4 Introduction of the semantics of the FRBL

This section presents the dynamic semantics of the FRBL language as a code generator that allows the transformation of FRBL code (represented as a FRBL model) into Java code. The generated Java code is based on the EMF library and can be executed on the JVM (Java Virtual Machine). Note that, in the field of programming languages, such code generators are commonly referred as compilers (e.g., C compilers allow the transformation of C code to Assembler code).

The xText approach provides a code generation facility based on xTend (Bettini 2013) that can be used to generate Java code. In this section we propose a part of the code generator for the FRBL language, written in xTend.

Listing 5 shows the *FRBLGenerator* class that contains the implementation of the FRBL code generator. The *doGenerate* method is called from the builder infrastructure whenever a FRBL model has changed. This method calls the *generateFile* that opens a Java file (or creates a new one if the file does not exist) and writes the Java code that is returned by the *compile* method in this file. The *compile* method takes on parameter the changed FRBL model and iterates the objects contained in this FRBL model, selecting all Rules. The *compileRule* method is then called for each Rule.

```
class FRBLGenerator implements IGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess fsa) {
		fsa.generateFile(
				getFileName()+'.java',
				resource.complie)
	}

	def CharSequence complie(Resource resource)'''
		«FOR r:resource.allContents.toIterable.filter(Rule)»
			«r.compileRule»
		«ENDFOR»
	'''
```

Listing 5. An extraction of the code-generation template defined by the *FRBLGenerator* class.

The method *compileRule* is illustrated in Listing 6. It returns, for each Rule, a skeleton of a Java method composed of a name, return type, parameters and a body. The method's body is generated based on the *compileExpression* method that is called for all expressions of a Rule (see also Listing 1 and Figure 74).

```
def compileRule(Rule rule)'''
	public «IF rule.returnType!=null»«rule.returnType»«ELSE»void«ENDIF»
	«rule.name»(«rule.parameters.compileParameters»){
		«FOR e:rule.expressions»
			«e.compileExpression»
		«ENDFOR»
	}
'''
```

Listing 6. The *compileRule* method, extracted from the code-generation template.

The method *compileExpression* dispatch the method call based on the type of the expression, as shown in Listing 7.

```
def compileExpression(Expression e)'''
	«IF e instanceof ArithmeticExp»«(e as ArithmeticExp).compileArithmeticExp
»«ENDIF»
	«IF e instanceof VarExp »«(e as VarExp).compileVarExp»«ENDIF»
	«IF e instanceof CondExp»«(e as CondExp).compileCondExp»«ENDIF»
	«IF e instanceof SetExp»«(e as SetExp).compileSetExp»«ENDIF»
'''
```

Listing 7. The *compileExpression* method, extracted from the code-generation template

For instance, the method *compileCondExp* is called if the expression is indeed a *Conditional Expression*. In this case, a Java *if-then* statement is generated, as illustrated in Listing 8.

```
def compileCondExp(CondExp condExp)'''
	if(«condExp.condition.compileArithmeticExp»){
		«FOR e : condExp.ifBody»
			«e.compileExpression»
		«ENDFOR»
	}
'''
```

Listing 8. The *compileCondExp* method, extracted from the code-generation template.

The method *compileCondExp* is called is the expression is indeed a *Set Expression*. In this case, a Java *for* statement is generated, as illustrated in Listing 9.

```
def compileSetExp(SetExp setExp)'''
	«"for(EObject " + setExp.^var.name + " : this.eContents())
	{
		if(" + setExp.^var.name + " instanceof EClass && ((EClass) " +
setExp.^var.name + ").getName().equals(\"" + setExp.set.name + "\"))
		{"»
		«FOR e : setExp.setBody»
			«e.compileExpression»
		«ENDFOR»
		}
	}
'''
```

Listing 9. The *compileSetExp* method, extracted from the code-generation template.

For instance, Listing 10 shows the FRBL rule "*ReadingInputs*" and the resulting Java code (i.e., the result of the code generation process). Note that the Java code is based on the EMF library.

FRBL Rule:

```
[rule ReadingInputs]
	[forall input in Input]
		input.getReadInputsFrom().read(input)
	[/forall]
[/rule]
```

Resulting Java code (compatible with the EML library):

```
public void ReadingInputs(){
	for(EObject input : this.eContents())
	{
		if(input instanceof EClass &&
			((EClass) input).getName().equals("Input"))
		{
			((Input)input).getReadInputsFrom().read((Input)input);
		}
	}
}
```

Listing 10. An FRBL rule and the resulting Java code.

4.3.5 Example: designing the behavior of eISM by using the FRBL

Before designing the dynamic semantics of the eISM language, let's first introduce the generic template for DE behaviors (discussed in Section 4.3.2 and illustrated in Figure 73) that is automatically generated.

```
[rule ReadingInputs]
	[forall input in Input]
		//read inputs from the blackboard
		input.getReadInputsFrom().read(input)
	[/forall]
[/rule]

[rule CalculatingFutureState]
	//Complete this rule based on the formal pre-defined semantics
[/rule]

[rule WritingOutputs]
	[forall output in Output]
		//write outputs into the blackboard
		output.getWritingOutputsInto().write(output)
	[/forall]
[/rule]
```

Listing 11. Template for DE behaviors based on three rules.

The template is shown in Listing 11 composed of three general rules that every DE language must implement:

- Rule 1: reading inputs
- Rule 2: calculating future state
- Rule 3: writing outputs

There rules are managed by the Controller of the blackboard design pattern for synchronized model execution. Chapter V provides details for the synchronized execution and the handling of these rules. See also Figure 59 for the structure of the blackboard design pattern, i.e., the relations of the input and output concepts with the blackboard concepts. Note that the methods *read(Input i)* and *write(Output o)* are defined based on the blackboard design pattern for the blackboard concept.

These three rules must furthermore be completed based on the formal pre-defined behavior of the considered DE language. For instance, in our case, we consider the formal semantics of eISM discussed in Section 4.2.2. Based on these semantics, Listing 12 shows how the templated can be completed introducing several auxiliary Rules:

- WriteInputInDataPart (contained in the main ReadingInputs rule)
- ReadOutputFromDataPart (contained in the main WritingInputs rule)
- EvaluateFiringConditionPropositionalVariables, FireTransitions and Evaluate Updates (contained in the main CalculatingFutureState rule)

```
[rule ReadingInputs]
      [forall input in Input]
            //read inputs from the blackboard
            input.getReadInputsFrom().read(input)
            //write inputs into the data part
            WriteInputInDataPart(input)
      [/forall]
[/rule]

[rule CalculatingFutureState] //Calculatinf future state for eISM models
      EvaluateFiringConditionPropositionalVariables()
      FireTransitions()
      EvaluateUpdates()
[/rule]

[rule WritingOutputs]
      [forall output in Output]
            //load outputs from the data part before writing
            output:=ReadOutputFromDataPart(output)
            //write outputs into the blackboard
            output.getWritingOutputsInto().write(output)
      [/forall]
[/rule]
```

Listing 12. Completing the template for the eISM language.

The auxiliary rules are show in Listing 13, based on the eISM metamodel shown in Figure 59.

```
[rule WriteInputInDataPart parameters: Input i]
      getDP().getLanguageData().add(i)
[/rule]

[rule ReadOutputFromDataPart parameters: Output o output: Output]
      Integer index := getDP().getLanguageData().indexOf(o)
      output getDP().getLanguageData().get(index)
[/rule]

[rule EvaluateFiringConditionPropositionalVariables]
      [forall fcf in FiringConditionFunction]
            Boolean expVal := eval(fcf.getBooleanExpression(),
                                   getDP().getLanguageData(),
                                   getDP().getInternalData())
            Boolean eventVal := eval(fcf.getRequestedEvents(),
                                   getDP().getLanguageData())
            fcf.setActivates(expVal and eventVal)
      [/forall]
[/rule]

[rule FireTransitions]
      [forall t in Transition]
            [if t.getSource().isCurrent()=true and t.getFiringCondition()=true]
                  t.getSource().setCurrent(false)
                  t.getTarget().setCurrent(true)
                  t.getUpdatePropVar.setVal(true)
            [/if]
      [/forall]
[/rule]

[rule EvaluateUpdates]
      [forall uv in UpdatePropVar]
            [if uv.getVal()=true]
                  [forall fld in uv.getActivates().getUpdateFctForLanguageData()]
                        eval(fld)
                  [/forall]
                  [forall fid in uv.getActivates().getUpdateFctForInternalData()]
                        eval(fid)
                  [/forall]
                  [forall fod in uv.getActivates().getUpdateFctForOutputData()]
                        eval(fod)
                  [/forall]
            [/if]
      [/forall]
[/rule]
```

Listing 13. The auxiliary rules for eISM.

The *WriteInputInDataPart* rule is used to write the inputs provided through the blackboard in the data part. The *ReadOutputFromDataPart* rule is used to load the data from the data part (that has potentially changed after calculating the future state) and to write it in the blackboard. The *EvaluateFiringConditionPropositionalVariables* rule evaluates the firing condition propositional variables based on the data contained in the data part. The *FireTransitions* rule fires transitions, deactivating the source state and activating the target state of transitions. It activates also the update variables. Note that there is at most one transition that can be fired, otherwise the model violated the

deterministic functioning hypothesis. Finally, the *EvaluateUpdates* rule evaluates the update functions associated to the activated update.

Note that the firing conditions and the update functions are specified as a String. These strings represent model level code defined by the designer (i.e., written using an opaque action language). According to (Combemale et al. 2013), such model level code can be written by using scripting languages allowing dynamic invocation, as they demonstrate by using the Groovy language. The Groovy language is an object-oriented programming language for the Java platform (Koenig et al. 2007). Fortunately, the FRBL code generator discussed in Section 4.3.4 generates Java code and thus FRBL can be integrated with Groovy for the evaluation of such String expressions.

Finally, the generated Java code is fully compatible with the EMF library that is generated from the metamodel illustrated in Figure 59. Therefore, they must be integrated before the promotion to the M3 layer illustrated in Figure 58.

4.3.6 On the fly design and integration of new DE languages with EMOF

Within the MBSE context, stakeholders must create their own DSML for modeling a viewpoint of a SoI (see Section 2.2.2 for more details). Achieving then model V&V requires DSML with semantics (static and dynamic) for simulation and formal proof. We have stressed the need of discrete-events (DE) languages for modeling the behavior (dynamic semantics) of DSML, introducing the eISM language in Section 4.2.

However, a real consensus about the use of one language for the design of DSML dynamic semantics does not currently exists and different approaches propose the use of different languages.

This section proposes an approach for “on the fly design and integration” of DE languages with EMOF. Executable DSMLs can then be designed based on EMOF (for the DSML abstract syntax) and on the newly designed DE language (for the DSML dynamic semantics). In such a way, stakeholders can design their own DE language for the design of dynamic semantics.

For this purpose, we propose an approach based on the FRBL and the EMOF. The approach is illustrated in Figure 78 as an extension of the initial EMOF-eISM integration process illustrated in Figure 58. It is composed of five steps:

- Step 1: design the abstract syntax of the DE language by using EMOF

- Step 2: design the dynamic semantics of the DE language by using FRBL
- Step 3: download the meta-metamodel to the M2 layer
- Step 4: specify the dependencies between the new DE language and EMOF
- Step 5: promote the result at the M3 layer

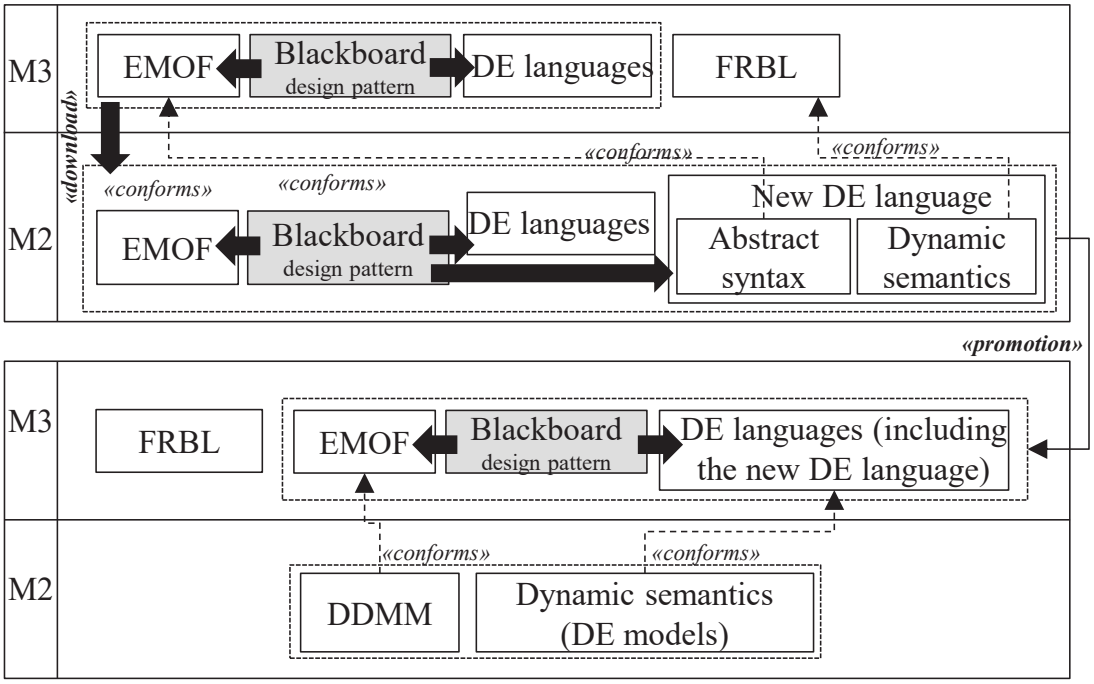


Figure 78. On the fly design and integration of DE languages with EMOF

During step 1 and 2, a DE language can be design by using EMOF (for the DE language abstract syntax) and FRBL (for the DE language dynamic semantics). This is for instance illustrated in Figure 56 for the eISM abstract syntax and in Listing 12 and Listing 13 for the eISM dynamic semantics.

The third step consists in recovering (downloading) the meta-metamodel at M2 layer.

The fourth step consists in establishing the relationships between the downloaded meta-metamodel and the new DE language. Note that, to address the issue of model interoperability in terms of behavioral dependencies and synchronized execution of DE models, the integration process is established following the blackboard design pattern. The generated Java code from the FRBL dynamic semantics must be integrated with the generated Java code of the meta-metamodel by using the EMF. Chapter V provides more details on the blackboard design pattern and on the synchronized execution algorithm.

Finally the resulting metamodel is promoted to the M3 layer, replacing the previous meta-metamodel.

This process must be repeated for each newly added DE language. For instance, Figure 79 shows the result of the above process applied on the meta-metamodel shown in Figure 59 that contains only one DE language, i.e., the eISM. The result is a meta-metamodel that contains the new DE language along with the eISM. Note that, in addition to this process, new graphical editors must be designed for the design and management of "new DE" models, as discussed for the eISM language in Section 4.2.4.

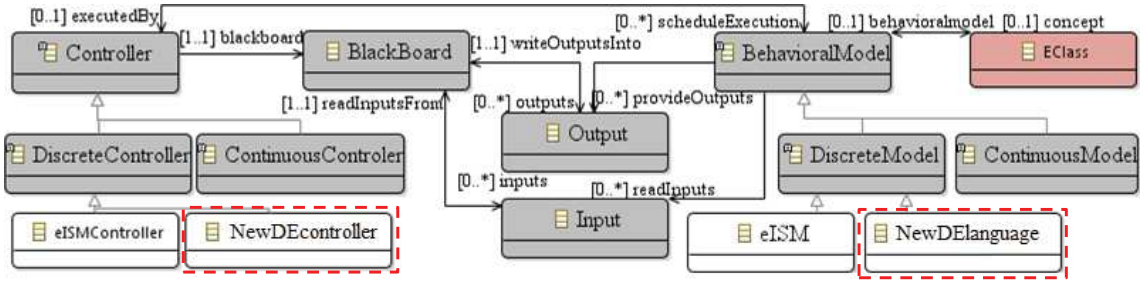


Figure 79. Integrating EMOF with a new DE language based on the blackboard design pattern.

The meta-metamodel with multiple DE languages allows the design of DSML dynamic semantics based on multiple DE languages, denoted *mixed dynamic semantics*. A mixed dynamic semantics includes behavioral models designed by different behavioral languages. The synchronization between different behavioral models (e.g., synchronization between Statechart models and eISM models) is guaranteed by the blackboard design pattern and the synchronization rules, introduced in Chapter V.

4.4 Conclusion

This chapter focuses on modeling behavior for MBSE, i.e., on the design of executable DSMLs that allow simulation (i.e., model execution). It evaluates first a well-known design pattern for executable DSML for its effective adaptation in the field of MBSE. The goal is to create an executable version of a well-known language to MBSE experts, i.e., an executable eFFBD (enhanced Functional Flow Block Diagram), denoted xeFFBD. This application example allows us to highlight several issues, as well as possible improvements for the effective adaptation of this design pattern in the field of MBSE. Based on the feedback, Chapter IV introduces two languages that can be used to design the behavior of a DSML.

The first language is an extended version of the Interpreted Sequential Machine denoted (eISM). eISM is a behavioral language based on discrete-events hypotheses. In comparison to other discrete-events languages eISM has several advantages: it operates with typed input/output data and complex expressions build using types data, it separates classical state/transition specification from data specification, allowing the specification of some states using variables and it has formal underlying structure. For the design of executable DSMLs, eISM is integrated with the metamodeling language EMOF, creating an executable metamodeling language. In such a way, the behavior of a DSML is specified as a set of discrete-events behavioral models, each one associated to different domain concepts of the DSML abstract syntax.

The second language is a formal rule based language denoted FRBL. The goal of FRBL is to ease and assist the design of the behavior of a DSML that have formal pre-defined semantics based on the one hand, on the reuse of the DSML's predefined formal semantics and on the other hand, based on a generic template. The behavior of a DSML is finally specified as a set of formal rules, among which the following three rules are considered as main rules defined by the generic template: 1) read inputs, 2) calculate future state and 3) write outputs. The syntax and the semantics of FRBL and designed using the xText approach.

CHAPTER V

VERIFICATION AND VALIDATION

This chapter presents the last part of the conceptual, methodological and technical contributions of this work, i.e., an approach for system modeling and V&V denoted "xviCore". A map of Chapter's outline with respect to the type of contributions is shown in Figure 80.

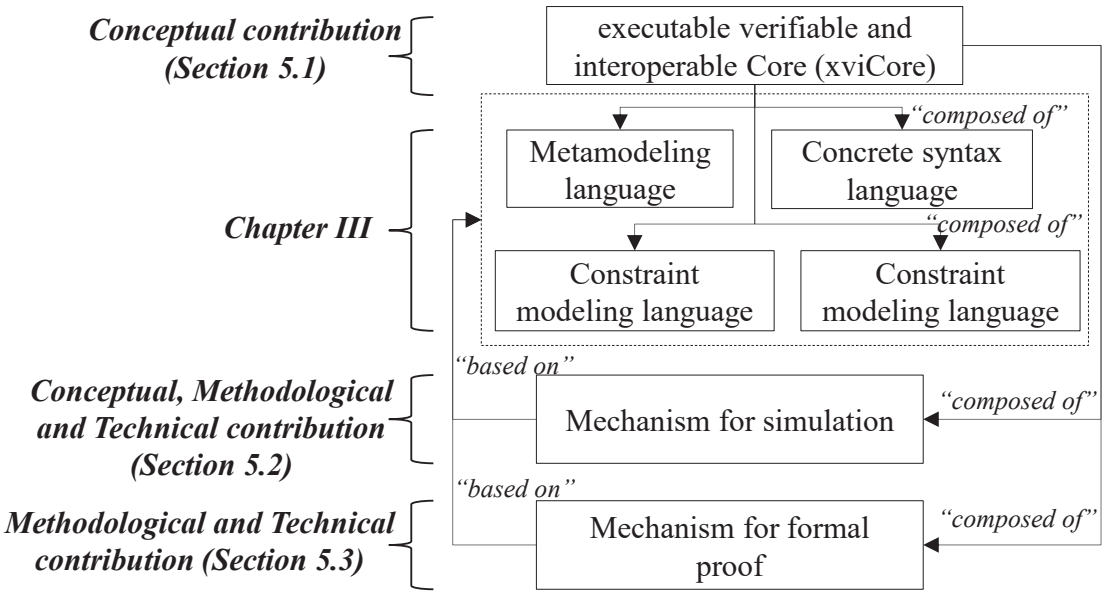


Figure 80. Map of conceptual, methodological and technical contributions of Chapter V.

xviCore promotes mechanisms for simulation based on model execution, and mechanisms for formal properties proof. The chapter is structured as follows. Section 5 introduces xviCore. Section 5.2 proposes xviCore's mechanism for coordinated simulation based on the blackboard design pattern and the use of dynamic semantics for model execution. Section 5.3 introduces xviCore's mechanisms for formal proof based on the CREI property modeling language for the specification of all types of properties and on adequate model-checking tools for properties proof. Finally Section 5.4 concludes the contribution.

5.1 Introduction: executable, verifiable and interoperable Core

We have issued several working hypotheses and choices in the previous chapters.

First, as previously discussed, DSML semantics is often neglected or, when needed, provided by means of translating the DSML into third-party formalisms (Nastov Blazo 2014). This is, from our perspective, a limitation for the V&V of models in the context of MBSE. A discussion on this topic is proposed in Chapter II.

Second, as proposed in Chapter III, both DSML syntax (abstract and concrete) and DSML semantics (static and dynamic) can be formalized as a set of properties following the DSML and model lifecycle. Property proof and model simulation are then classically achieved based on transformation mechanisms (Mahfouz et al. 2013). This technique leads to information loss, especially when considering a composite model (i.e., an integration of several viewpoint models). Indeed, on the one hand, each of the viewpoint models must be correctly transformed into a single formal specification. On the other hand, achieved results must be correctly translated back and interpreted for each of the originating viewpoint models.

Third, in Chapter IV we propose two languages for designing the behavior of a DSML: eISM and FRBL, along with a process for their integration with the metamodeling language EMOF. The resulting executable metamodeling language is used to specify DSML abstract syntaxes and DSML dynamic semantics.

However, the modeling based on properties introduces in Chapter IV, highlights, in addition to a metamodeling language (MML) and a behavioral modeling language (BML), the need for a concrete syntax language (CSL) and a constraint modeling language (CML), allowing then the design of all parts of a DSML (see Chapter II), i.e., DSML abstract syntax, DSML concrete syntax, DSML static semantics and DSML dynamic semantics, and different types of dependencies, i.e., structural, representational and behavioral.

This chapter introduces *eXecutable, Verifiable and Interoperable Core* (xviCore) a method that integrates a MML, a CSL, a BML and a CML, for the design of executable, verifiable and interoperable DSMLs (xviDSMLs). The design process of xviCore is that of the composite DSML and model lifecycle introduced in Chapter III. An xviDSML is a composing DSML in a composite DSML that is composed of:

iii. **Abstract syntaxes** define through metamodels the core concepts and attributes that specify a particular SoI viewpoint as well as the relationships that bound together these concepts.

iv. **Concrete syntaxes** define the graphical or textual representation of concepts. This information is later used to represent graphically or textually the instances of concepts in an editor. For this work we consider only concrete syntaxes (see Chapter II for more details).

v. **Static semantics** define constraint properties, i.e., restrictions and additional information on the syntaxes or the behavior (the dynamic semantics) that cannot be implicated.

vi. **Dynamic semantics** (behavioral specifications) define the behavior of DSML through behavioral models. Can be specified by using different techniques discussed in Chapter IV.

vii. **Dependencies** define the relationships between syntaxes (abstract and concrete) specifying how different DSML are structurally and graphically bound together, and the relationships between the semantics (static and dynamic) specifying how different DSML are behaviorally bound together, but also constraint properties based on the dependencies.

Additionally, to put in use the semantics of an xviDSML, we propose in Section 5.2 a mechanism for simulation and in Section 5.3 a mechanism for formal proof.

5.2 Simulation mechanisms

Prior to simulation is the specification of behavior. Two different techniques for the design of DSML behavior are proposed in Chapter IV, by using a formal behavioral modeling language based on discrete-event hypothesis and by using a formal rule-based language. The first technique promotes the eISM language for the design of discrete-events behavioral models to specify the behavior of DSML concepts. Let's remember that this choice is here considered as an example and the behavioral DSML (i.e., eISM) can be chosen differently, e.g., by using classical States Machine, Temporised or Temporal Petri Nets or even FCCS. The choice of eISM is justified in Chapter IV. The second technique promotes FRBL to ease and assist the design of dynamic semantics as much as possible for discrete-events (DE) languages with pre-defined semantics.

The process of simulation consists in using the DSML behavior (dynamic semantics) to execute models created by a DSML. In our case, the behavior is defined by a set of behavioral models, for instance based on discrete-events eISM models. These behavioral models requires mechanisms for synchronization and centralized data and events exchanges. So, each step of the execution (execution step), all behavioral models from one DSML (or several composing DSMLs when considering composite DSML) must be synchronously executed based on a data that is derived from the domain model.

The data changes in the process, consequently changing the characteristics of the domain model. Stakeholders observe these changes and judge about the relevance of the model vis-à-vis the expected reality.

We propose a solution of the concurrent execution of behavioral models and centralized data and events exchanges by applying the blackboard design pattern proposed by (Engelmore & Morgan 1988) and on new hereafter introduced synchronization rules.

5.2.1 The blackboard design pattern

The blackboard design pattern is illustrated in Figure 81. It is a behavioral pattern "*affecting when and how programs react and perform*".

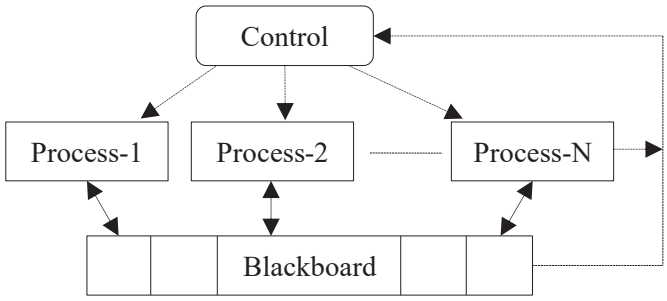


Figure 81. An overview of the blackboard design pattern.

A "blackboard" is a shared and structured *memory* that establishes *relationships* between independent *modules* called "autonomous processes" where each process is individually able to solve a sub-problem. Processes can solve a "global problem" when they are put together, reading and writing data in the blackboard that is iteratively updated. Each process has a set of triggering *conditions* that have to be satisfied by particular kinds of *events*, sent by a *controller*.

The processes synchronization is handled by a controller that monitors the *data* stored into the blackboard and decides which autonomous processes to *prioritize*. The controller reacts to global changes in the blackboard resulting from external *inputs* or previously executed processes. Processes can be simultaneously executed, having a concurrent access to the relevant blackboard data. This may potentially produce a situation of deadlock (if two or more processes are each waiting for the other to finish, and thus neither ever does) (Lalanda 1997).

Our solution based on the blackboard design pattern is composed of three main components:

- a shared and structured memory denoted Blackboard;
- behavioral models that represent the concurrent processes;
- a controller that schedules the execution of behavioral models;

The *Blackboard* is a common and time dependent base of information where behavioral models write their output data (*O*) and read their input data (*I*), enabling information exchange. It is formally defined as a 5-uplet $BB \stackrel{def}{=} (AT,LT,V,S,R)$ where: *AT* is the set of variables specifying the time of adding. *LT* is the set of "lifetime" variables, indicating the remaining time before updating messages from the blackboard. *V* is the set of "variables carried out by the messages. S is the set of "sender" variables specifying the behavioral model that sent the message and $R=\{R_1,..,R_k\}$ is the set of "receivers" variables indicating the behavioral models that read the message.

Behavioral models are designed as proposed in Chapter IV by using a behavioral modeling language.

Controller is used to schedule the execution of all behavioral models from one DSML (or several composing DSMLs when considering composite DSML). The execution scheduling process is based on:

- a multiscale time
- a reconciliation rule
- a cadence rule
- (optional) stability management
- an execution scheduling algorithm

They are introduced and formally defined in the next section.

Note that, although, only discrete-events behavioral models are currently experimented, the following rules are envisioned so that continuous behavioral models can also be integrated and evolve interchangeably.

5.2.2 Execution scheduling

***Multiscale time*:** managing the behavior of several DSMLs at once, represented through several behavioral models, requires two time scales, as for instance proposed by the Ptolemy approach (Lee 2003) or for the synchronization and stability management of FCCS proposed in (Chapurlat 1994). One of the time scales must be related to the environment, denoted "environmental" or "global" time. The global time is identical for

all behavioral models (i.e., for all parametrized behavioral models) and is used by the Controller for synchronization. Behavioral models are executed by the Controller based on this time scale. The other time scale is unique to each behavioral model, denoted "model" time. The model time is used to monitor the execution steps of different behavioral models, i.e., the time through which a behavioral model evolves, from reading inputs to writing outputs. Both time scales are logical times scales, i.e., they define an ordered relationship between instants that can be referenced in logical time units (LTU) without any relation with a real time scale e.g. hour, mn, s, or ms.

For instance, the behavior of the WaterDistrib DSML shown in Figure 69 is composed of three eISM behavioral models, one for the concept Valve (see Figure 70), one for the concept Control station (see Figure 71) and one for the concept Reservoir. To manage the execution of these behavioral models we need:

- A model time scale for each parametrization of the Valve eISM model (for instance, one for the input valve and one for the output valve)
- A model time scale for each parametrization of the Control Station eISM model
- A model time scale for each parametrization of the Reservoir eISM model
- A global time scale to synchronize the behavioral models

The execution of the WaterDistrib model shown in Figure 67 can then be managed by one global time scale and four model time scales: one for the parametrization of the Valve eISM model for the input valve instance, one for the parametrization of the Valve eISM model for the output valve instance, one for the parametrization of the Control Station eISM model for the control station instance and one for the parametrization of the Reservoir eISM model for the reservoir instance. This is illustrated in Figure 82.

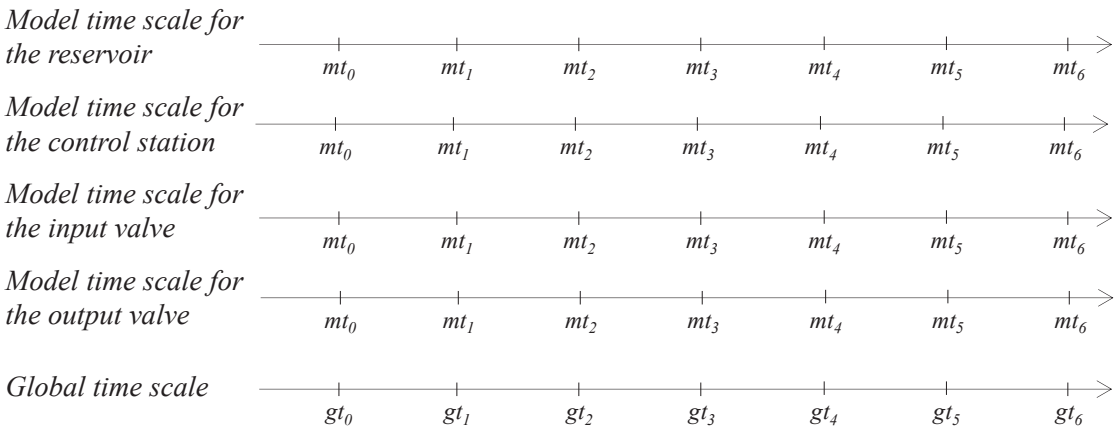


Figure 82. Time scales to manage the execution of the WaterDistrib model shown in Figure 67.

As previously quoted, the global time scale is used by the controller to synchronize behavioral models, i.e., to determine the time instants on the global time scale when models are executed. For instance, some behavioral models are executed each global time instants, while other are executed each three time instants. For this purpose, the controller calculates the time instants for model synchronization based on two rules: the reconciliation rule and the cadence rule.

***Reconciliation rule*:** aims to establish synchronization points between discrete-events models based on logical time scales and continuous behavioral models based on physical time scales, then to mix and make comparable various instants. Note that the physical time scale can moreover be put in correspondence with the real time scale. For instance, 1 physical time unit (PTU) corresponds to 7s. We introduce here a reconciliation function denoted $\underline{\omega}$ and formally defined as follows:

$$\underline{\omega}: \mathbb{N}^n \rightarrow \mathbb{N}$$
$$(ep_1, \dots, ep_n) \rightarrow r = \underline{\omega}(ep_1, \dots, ep_n) = lcm(ep_1, .., ep_n)$$

Where:

a) ep_i (*estimation parameter*) define the duration of one execution step (from reading inputs to writing outputs) in physical time units (PTU) of a discrete-event model. Estimation parameters are defined by a current value $cvalue_{ep}$, a domain definition E_p and a type $\mathbb{N}$, such that $E_p \subseteq \mathbb{N}$.
b) r (*reconciliation parameter*) is calculated by the reconciliation function $\underline{\omega}$ and used by the Controller for synchronization. The reconciliation parameter has a current value $cvalue_r$, a domain definition R and a type $\mathbb{N}$, such as $R \subseteq \mathbb{N}$. r is computed by using *lcm* which is a least common multiple function.
c) n is the number of behavioral models that define the behavior of a DSML.

For example, let the reaction time of a valve be 1ms, the reaction time of the Control Station be also 1ms, and the reaction time of the reservoir be 2ms. Let 1PTU be 1ms. The estimation parameters of the valve eISM model and the control station eISM models are equal to 1 (ep_1=ep_2=1) and the estimation parameter of the reservoir is equal

to 2 (ep_3=2). The reconciliation parameter r of the controller is then computed by using the reconciliation function $\underline{\omega}$. In this case, $r = lcm(ep_1, ep_{2,} ep_3) = lcm(1,1,2) = 1$.

***Cadence rule*:** the reconciliation rule suggests that the duration of execution steps of different behavioral models, measured on the global time scale, is different. At a given time stamp, the execution step of one behavioral model might start, while of another might still be in progress. The cadence rule aims to identity the duration of execution steps of different behavioral models, according to the global time scale. We introduce here a cadence function, denoted $\underline{\tau}$ and formally defined as follows:

$$\underline{\tau}: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$\forall i \in [1..n](ep_i, r) \to cp_i = \frac{ep_i}{r}$$

Where:

a) cp_i (cadence parameter) define the duration of an execution step for each discrete-event model. Cadence parameters are computed by the cadence function $\underline{\tau}$, taking into account the controller's reconciliation parameter r and the estimation parameter ep_i of considered (i^{th}) behavioral model. Each cadence parameters have a current value $cvalue_{cdi}$, a domain definition C_i and a type $\mathbb{N}$, such as $C_i \subseteq \mathbb{N}$.

For example, the cadence parameters of the valve eISM model and the control station eISM models are equal to 1 (cp_1=cp_2=1/1=1), meaning that the execution steps of Valve and Control Station eISM model occur and least for a time unit. The cadence parameter of the reservoir eISM model is equal to 2 (cp_3=2/1=2), meaning that the execution steps of Reservoir eISM model occur and least for two time unit. This is shown in Figure 83.

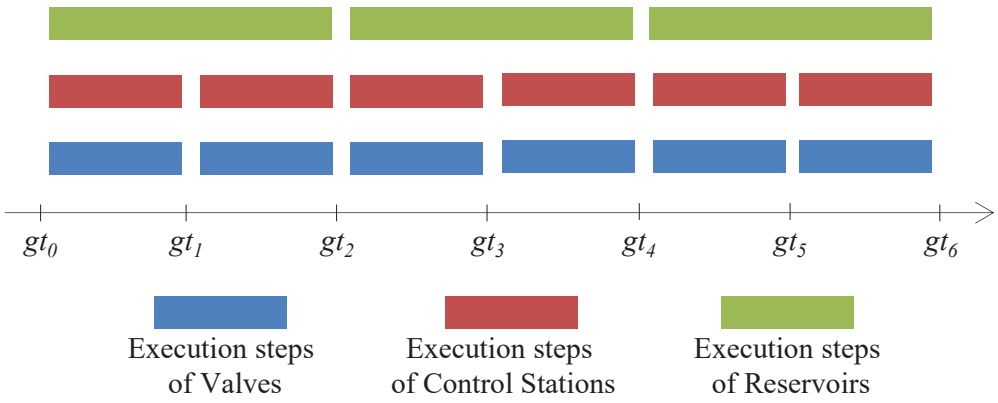


Figure 83. The duration of the execution steps of WaterDistrib components.

***Stability management*:** is already discussed in Chapter IV (Section 4.1.3). Let's recall that a behavioral model is "*stable*" if succeeding an evolution cycle, taking into account the same inputs, the model cannot evolve in another state. Otherwise, the model is "*unstable*" and its current state is named "*transient*" state, as defined in the case of Sequential Function Chart (IEC 1999). Stability management consists in checking the stability of a behavioral model every evolution cycle.

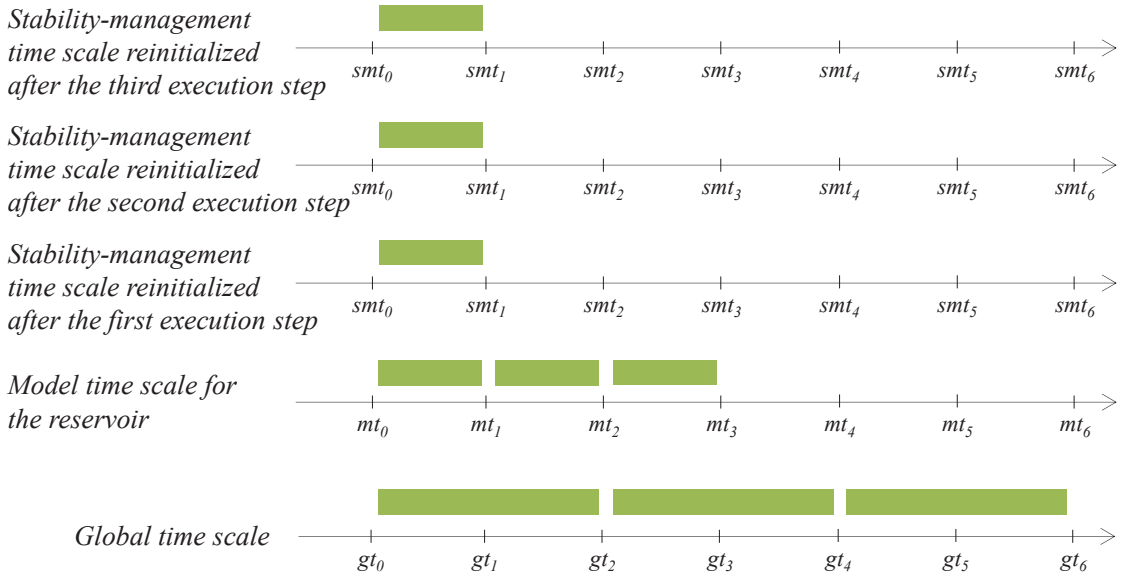


Figure 84. The three time scales involved in the execution of a reservoir.

Managing models stability involves a transient state detection algorithm that, in addition to the model time scale, introduces a third "stability-management" time scale that is reinitialized every time the model evolves and incremented while the model is in transient state, every time calculating its future state, eventually reaching its stability. In contrary, the model time scale is unique for each behavioral model and is initialized with the global time scale and incremented every evolution cycle of its corresponding model. In other words, it indicates, how many times this model evolved based on duration that is measured on the global time scale. Moreover, one time unit of the model time scale is equal to cd_i (cadence parameter) time units of the global time scale.

For example, Figure 84 shows the three time scales for the execution of a reservoir.

***Execution scheduling*:** the controller synchronizes behavioral models, i.e., it organizes each execution steps of all behavioral models during an execution of a domain model. The execution scheduling is based on the generic evolution algorithm shown in Figure

73 that consists in reading inputs (RI), calculating future state (CFS) and writing outputs (WO).

Let us remind you that behavioral models are related to a domain concept and they are used to compute data provided by instances of domain concepts. The execution scheduling process splits into two main phases: (1) *preparation* and (2) *execution*.

a) *Preparation* begins by computing the reconciliation parameter r of the controller, using the reconciliation function $\underline{\omega}$. Next, the cadence parameter cp_i of each behavioral model is calculated, using the cadence function $\underline{\tau}$. Then the synchronization process begins, initializing the global time scale, so behavioral models can start their evolution steps.

b) *Execution* begins when preparation is finished. Each behavioral model manages simultaneously the execution of several instances of the corresponding domain concept. Each execution consists of reading inputs (RI) from the blackboard, computing future state (CFE) considering stability management and writing outputs (WO) into the blackboard. The controller monitors the duration of all executions according to the global time scale, the reconciliation parameter and the cadence parameters. For each execution, it transmits a current state, data provided from an instance and the time unites of the model time scale. The execution algorithm is illustrated in Figure 85.

The execution scheduling is illustrated in the next section based on two examples.

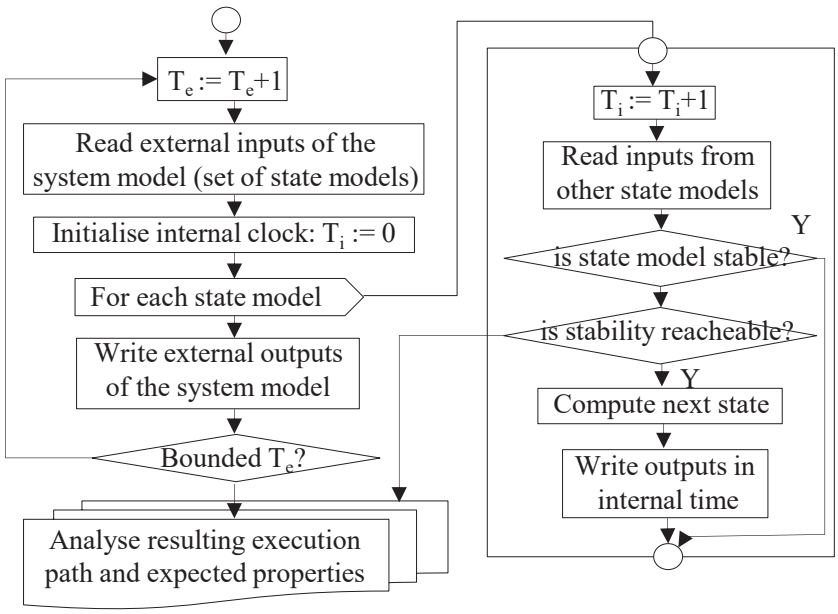


Figure 85. Execution algorithm.

5.2.3 Demonstration

Two examples are shown in this section to illustrate the execution scheduling, the prior showing the execution of a WaterDistrib model, i.e., a model designed by the WaterDistrib DSML, and the latter showing the coordinated execution of a eFFBD and a PBD models designed by eFFBD and PBD.

Example 1: WaterDistrib model execution. During the phase DSML run time / Model run time, behavioral models are used to execute models. The execution of a WaterDistrib model is schematized in Figure 86 where a simplified scenario is illustrated, showing how the WaterDistrib model reacts to a water request.

The whole process splits into two phases: (1) *preparation* and (2) *execution*.

1) *Preparation*: during this phase, first, the reconciliation parameter *r* of the controller is computed, using the reconciliation function $\underline{\omega}$. In this case, $r = lcm(ep_1, ep_{2,} ep_3) = lcm(1,1,2) = 1$. Next, the cadence parameters cp_1, cp_2 and cp_3 are computed using the cadence function $\underline{\tau}$. In this case, $cp_1 = cp_2 = 1/1 = 1$, meaning that the execution steps of Valve and Control Station eISM model occur and least for a time unit. The cadence parameter of the reservoir eISM model is equal to 2 (cp_3=2/1=2), meaning that the execution steps of Reservoir eISM model occur and least for two time unit, as shown in Figure 83. Then the synchronization process starts, initializing the global time scale, and behavioral models can start their evolution cycles. In the example shown in Figure 86, there is one instance of both *ControlStation* and *WaterTank* concepts, i.e., *cs* and *wt*, and two instances of the *Valve* concept, i.e., input valve *vIn* and output valve *vOut*. We denote: *Control* eISM model to describe the use of *ControlStation* eISM model for the execution of *cs* instance and *I or O Valve* eISM model to describe the use of *Valve* eISM model for the execution of *vIn and vOut* instances. All eISM behavioral models read inputs (RE) from the blackboard, computing future state (CFE) considering stability management and writing outputs (WO) into the blackboard.
2) *Execution*: during this phase, the experiment consists to manually add, just before time x, a *Water Request* message for the *cs* component in the blackboard (see the table of Figure 86). The *Control* eISM model reads inputs at time x, and enters in *Mode2* state, after calculating its future state, short after reading inputs.

Blackboard										
	LT	*1*	*1*	*/*	*/*	*/*	*1*	*1*		
	V	*WaterReq*	*Open*	*/*	*/*	*/*	*StopReq*	*Close*		
	S	*User*	***cs***	*/*	*/*	*/*	*User*	***cs***		
	R	*{cs}*	*{vIn, vOut}*	*/*	*/*	*/*	*{cs}*	*{vOut}*		

Figure 86. A simplified scenario showing how the WaterDistrib model is simulated.

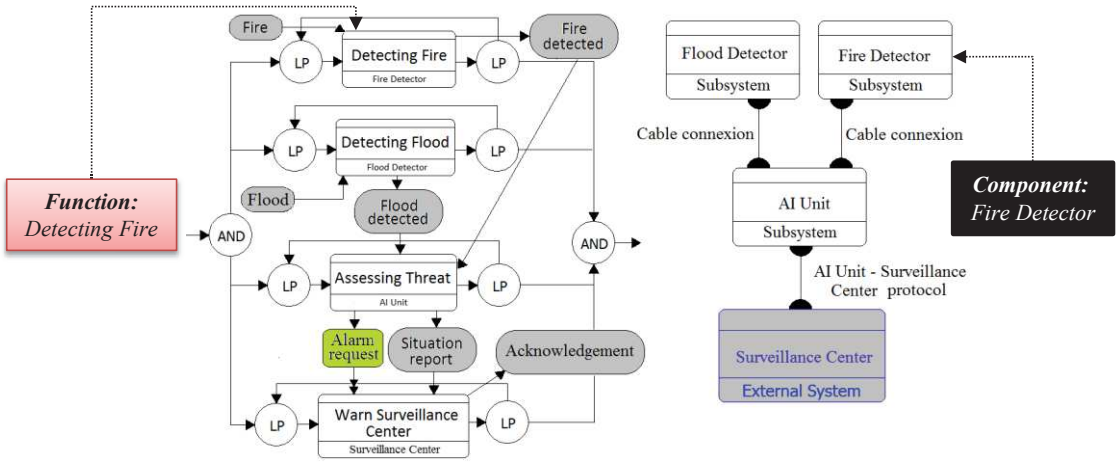
This activates the *filling-empting* update (see Figure 86), writing the *Open* message as an output into the blackboard at time x+1 (see the blackboard table of Figure 86). The *IValve and OValve* eISM models read then this message, at time x+1 and enter *Opening* state short after. At this point of time (*x+3*), both input and output valves (*vIn* and *vOut*) provide water flow to the water tank (*wt*) causing change in the water level inside this water tank. At a given point in time, the consumer had enough water and sends the stop water providing request (*StopReq*). For this, we manually add the *StopReq* message in the blackboard just before time *x+5* (see the table of Figure 86). The *Control* eISM model reads this message, at time *x+5* and after calculating future state, activates the *filling* update (see Figure 86), writing the *Close* message as outputs into the blackboard at time x+6 (see the table of Figure 86). The *OValve* eISM model reads then this message at time x+6 and enters *Closing* state short after. At this point of time (*x+7*), the water tank starts increasing its water level.

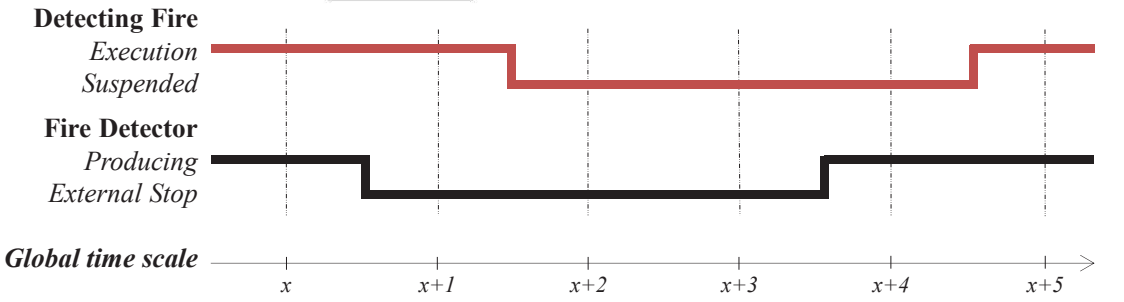
Example 2: coordinated execution of eFFBD and PBD models. This example, illustrated in Figure 87, shows the behavior of the architecture of a fire and flood security system through a coordinated execution of architecture's functional and physical viewpoint models. The scenario here consists in stressing the architecture's physical viewpoint model by sending a breakdown signal, putting one of the components (i.e., the fire detector component) into a non-functional breakdown state. The goal is to observe the reactions and the side-effects of the model and of the dependent viewpoint models (i.e., in this case the functional model) under such critical circumstances.

The whole process splits into two phases: (1) *preparation* and (2) *execution*.

- *Preparation*: during this phase, first, the reconciliation parameter *r* of the controller is computed, using the reconciliation function $\underline{\omega}$. In this case, $r = lcm(ep_1, ep_2) = lcm(1,1) = 1$. Next, the cadence parameters cp_1 and cp_2 are computed using the cadence function $\underline{\tau}$. In this case, $cp_1 = cp_2 = 1/1 = 1$, meaning that for both (*Function* and *Component*) eISM models, evolution cycles (an execution) occur and least for a time unit. Then the synchronization process starts, initializing the global time scale, and behavioral models can start their evolution cycles. In the example shown in Figure 87, there is one instance of

both *Function* and *Component* concepts, i.e., *Detecting Fire* and *Fire Detector*. We denote: *Detecting Fire* eISM model to describe the use of *Function* eISM model for the execution of *Detecting Fire* instance and *Fire Detector* eISM model to describe the use of *Component* eISM model for the execution of *Fire Detector* instance. Both models read inputs (RI) from the blackboard, computing future state (CFE) considering stability management and writing outputs (WO) into the blackboard.





Blackboard							
	LT	*1*	*1*	*/*	*/*	*/*	*/*
	V	*ExternalBreakdown*	*SuspendFunction*	*/*	*ResumeProduction*	*ResumeFunction*	*/*
	S	*User*	*Fire Detector*	*/*	*User*	*Fire Detector*	*/*
	R	*{Fire Detector}*	*{Detecting Fire}*	*/*	*{Fire Detector}*	*{Detecting Fire}*	*/*

Figure 87. A simplified scenario showing how a system architecture reacts when the Fire Detector enters in External Stop state

- *Execution*: during this phase, the experiment consists to manually add, at time x, an *External Breakdown* message for the *Fire Detector* component in the blackboard (see the table of Figure 87). The *Fire Detector* eISM model reads inputs at time x, and enters in *External Stop* state, after calculating its future

state, short after reading inputs. This activates *notify* update (see Figure 66), writing the *Suspend Function* message as outputs into the blackboard at time x+1 (see the table of Figure 87). The *Detecting Fire* eISM model reads then this message, at time x+1 and enters *Suspended* state short after. At this point of time, the behavioral property IBC_1 is respected and the *Fire Detector* component and its performing function *Detecting Fire* are not working (*ES* and *Suspended* states). To get the system back on running, the message *Resume Production* is manually added for the *Fire Detector* in the blackboard, at time x+3 (see the table of Figure 87). *The Fire Detector* eISM model reads this message, at time x+3 and after calculating future state, activates again the *notify* update, writing the *Resume Function* message as outputs into the blackboard at time x+4 (see the table of Figure 87). The *Detecting Fire* eISM model reads then this message at time x+4 and enters *Execution* state short after. At this point of time, the system is back in normal and the *Fire Detector* component and its performing function *Detecting Fire,* are working (Producing and Execution states).

5.3 Mechanism for formal proof

The mechanism for formal proof proposed here put in use the static semantics of a DSML. The static semantics are composed of different types of constraint properties, as proposed in Chapter III.

Generally, a formal verification process is based on 1) a formal specification, used as an underlying structure on which 2) formal constraint properties are verified by 3) an adequate model-checking tool. The goal is to check if the formal specification respects the formal properties.

Similarly, the formal proof mechanism proposed in this section is grounded on a formal specification defined as a set of structural, representational and behavioral properties (as proposed in Chapter III), different types of formal constraint properties and an adequate model-checking tool. We discuss below each of these parts individually.

5.3.1 Formal specification

Within the context of MBSE, the formal specification required for a verification process is extracted from the DSML's abstract syntax, concrete syntax and dynamic semantics as a set of structural, representational and behavioral properties that are here-denoted

formal structural, representational and behavioral specifications. For instance, abstract syntaxes can be defined as metamodels that naturally have an underlying structure based on an oriented graph that can be used for formal verification. Dynamic semantics designed by the formal behavioral modeling language eISM has a formal underlying structure based on a set of elementary valid formals (EVF), as previously discussed in Chapter IV.

In some cases, the formal structural, representational or behavioral specifications are not directly verified considering the DSML's constraint properties, but rather transformed into the formal specification of a third party formal approach before being verified. In such cases, the use of transformation techniques is leveraged, mapping the source DSML specification (i.e., the DSML's abstract syntax, concrete syntax or dynamic semantics) to an adequate target specification of a formal model (e.g., the Networked Timed Automata model in the case of UPPAAL tools (Larsen et al. 1997) for the DSML's dynamic semantics, or to the COGITAN library (Chein et al. 2009) for the DSML's abstract syntax).

However, we argue in Chapter II that transformation approaches have several limitations. On the other hand, the analysis results are only available in the target spaces, so they should always be interpreted back to the source space, to compare the result based on the source model. The relevance between source and target models should be demonstrated to assure that the behavior defined by the target model corresponds to the one of the source model. In addition, a good knowledge and expertise in the chosen target domain and in transformation languages and tools is required.

This work leverages a strategy for "direct verification", i.e., a verification of the formal DSML's specifications without transforming them into adequate third-party specifications. It focuses particularly on the verification of the structural specification (the abstract syntax) and the behavior specification (the dynamic semantics) of a DSML. The verification of the representation specification (the concrete syntax) is currently out of the scope.

For the above state purpose, the formal structural specification is here-limited to an EMOF metamodel form designed by the EMF approach (Steinberg et al. 2008), leaving other approaches out of the scope. The extensibility of xviCore for on the fly design and

integration of new behavioral modeling language (see Chapter IV) doesn't permit the use of one specific type of behavioral specification (e.g., behavioral specification based only on eISM models), as for the structural specification which is limited to EMOF metamodels. Hence, for the purpose of formal proof, it is mandatory to use a behavioral modeling language with formal underlying structure (such as for instance eISM) that allow direct verification without transforming into third party approaches.

5.3.2 Formal constraint properties

The information that cannot be implicitly defined by a formal specification must be explicitly defined as formal constraint properties by using a constraint modeling language and verified by an adequate model-checking tool.

Chapter III introduces several types of constraint properties as a particular type of the overall modeling and system properties. In general, all types of constraint properties are specified for the structural specification, denoted structural constraint properties, for the representational specification denoted representational constraint properties or for the behavioral specification denoted behavioral constraint properties. We focus here on the specification and the verification of structural and behavioral properties. Representational properties are currently out of the scope. For more details on different types of constraint properties, their purpose and use, readers are encouraged to see Chapter III.

Constraint properties are defined by using a constraint modeling language (CML). There are currently different types of CMLs (e.g., OCL, TOCL, LTL or the UPSL framework) that are used for different purposes. For example, OCL (Object Constraint Language) (OMG 2014) is complementary to UML and is used to express properties that cannot be defined using the UML's graphical notations. It is also applied in the Eclipse / MOF environment, proposing verbose predicates specification that is based on object-oriented notation and navigation. OCL allows the specification of a-temporal structural properties through invariants, derivations, initializations, etc., but also the specification of a-temporal behavioral properties, e.g., pre and post conditions, body, etc. However, OCL can neither be used for the design of temporal (structural and behavioral) properties, nor for behavioral properties that are not specified for operations-like behaviors (designed by an action language). To fill this gap, the temporal extension of OCL denoted TOCL (Ziemann & Gogolla 2003) can be used for

the design of temporal properties. TOCL is a mixture of OCL with logico-temporal operators, i.e., next, sometimes, once, eventually. Yet, similarly to OCL, TOCL is intended for operations-like behaviors, and cannot be used for the specification of behavioral properties when the behavior is designed by a behavioral modeling language, such as eISM. LTL (Linear Temporal Logics) (Pnueli 1977) can be used for the design of temporal behavioral properties for automata-like behaviors. LTL belongs in the group of formal languages. In general, formal languages are exhaustive, tool-supported allowing formal proofs. However, they remain difficult to use, are often considered as time consuming and require particular set of skills, tolls and proof techniques.

On the other hand, the UPSL framework (Chapurlat 2013) seems more adapted and finally usable to specify all types of properties. However, currently provided verification techniques of UPSL are based on Conceptual Graphs for structural properties and on UPPAAL (Larsen et al. 1997) for behavioral properties. This requires transforming, on the one hand, both DSML's structural specification into conceptual graphs specification based on the COGITANT library (Chein et al. 2009) and DSML's behavioral specification into the Networked Timed Automata model of the UPPAAL tool. On the other hand, it requires transforming both structural properties into a COGITANT formalism and behavioral properties into TCTL (time computational tree logic) when using UPPAAL. Properties can then be verified in these third-party formalisms. Unfortunately, such transformations are here-considered to be limited because obtained results are only available in these third-party formalisms, so they must be translated back and interpreted for the initial model, making it a potential source of information loss.

To overcome this issue, let's first introduce the UPSL's constraint modeling language CREI (Cause Relation Effect and Indicators) that is initially introduced in (Lamine 2001). CREI is intended to encourage and facilitate the work of engineers that are not specialized in formal modeling, offering the reuse of formal constraint property modeling and proof mechanisms. CREI constraint properties are composed of a group of causes (C) related to a group of effects (E), by a parametrized and constrained relation (R) and evaluated considering indicators (I). CREI properties are specified based on a formal specification given in a form of modeling variables, parameters, or predicates, defined by the set F as follows:

- $F = SP \cup BP$ where *SP* is the set of structural properties and *BP* is the set of behavioral properties (see Chapter III).

CREI properties are then formally defined as follows:

$$P := \langle reference_p, C, R, E, I \rangle$$

With:

- $reference_p \in String$ is a unique handle for property proof traceability
- $C := \{v_i | v_i \in F, i \in \mathbb{N}^+ and\ i \leq card(F)\}$ is the set of causes. C can be empty ($C \in \emptyset$)and in this case the property is denoted "*proper" property*, composed solely on effects.
- $E := \{v_j | v_j \in F, j \in \mathbb{N}^+\ and\ j \leq card(F)\}$ is the set of effects. E cannot be empty ($C \neq \emptyset$).
- *I* (optional) is a set of criteria that characterize the truthfulness of the property ($I \subseteq F$).
- $R := \langle type, \theta_c, \theta_e, \theta_i, T_p \rangle$ where:
    1. $type \in \{'implies', 'influences'\}$ defines the relation type.
    2. $\theta_c : T^k, C^m, \Re^{+*n} \rightarrow \{True, False\}$ constraints the interpretation of causes, i.e., a boolean condition that must evaluate to true to interpret C. By default $\theta_c = True$.
    3. $\theta_e : T^o, E^p, \Re^{+*n} \rightarrow \{True, False\}$ constraints the interpretation of effects, i.e., a boolean condition that must evaluate to true to interpret E.
    4. θ_i (optional, when *type=influences* and $\theta_c = True$) is an influence factor characterizing the link between C and E, which cannot be formalized as a temporal or logical relation. θ_i is defined as *"knowing with certainty C, we can deduce with certainty what E is"* i.e., knowing the values (and their variations) of causes defined in C allows us to deduce the values (and the variations) of effects defined in E. θ_i allows interpreting a beneficial or harmful influence depending on its value that varies between [-1,1], formally defined as follows:
        - $\theta_i = 0$: there is no real influence between the causes and the effects. The default value of $\theta_i = 0$.

- $\theta_i \to 1$ *(beneficial influence):* each variation in causes results into a variation in effects that is considered as beneficial for the system.
- $\theta_i \to -1$*(harmful influence)*: each variation in causes results into a variation in effects that is considered as harmful for the system.

5. $T_p = C \cap E$ is the set of variables the can be interpreted as causes and as effects at the same time.

Similarly to our classification of constraint properties, a CREI constraint property can be either:

- *Static (a-temporal):* expressing the rules and consistency characteristics of the model (see example in Figure 88) regarding its metamodel, consistency between model (inter-view and inter-languages), and time independent requirements.
- *Dynamic (i.e. temporal):* it can be used to describe the behavioral expectations (see an example in Figure 88) of the model or time-dependent requirements of the SoI.

Natural language	***CREI***
P_1(a-temporal): If a component C has a mission function, then this function is allocated and performed by C	***Cause:*** $\forall c \in \text{Component} \mid \exists f \in c.\text{mission} \mid f \neq \emptyset$ ***Relation:*** $(\Longrightarrow)$ ***Effect:*** $f \in c.\text{functions AND } c \in f.\text{performs}$
P_2(temporal): If a component enters a breakdown state (internal or external), its functions will be unable to continue execution	***Cause:*** $\forall c \in \text{Component} \mid c.\text{State} = \text{SS OR } c.\text{State} = \text{ES}$ ***Relation:*** $(\Longrightarrow)$ ***Effect:*** $o(\forall f \in c.\text{performs} \mid f.\text{State} \neq \text{Execution})$

Figure 88. Example of temporal and a-temporal CREI properties

A complete EBNF grammar for the CREI language is proposed in (Chapurlat 2013). Based on this grammar, we have developed an xText editor for CREI. A snapshot of the editor is shown in Figure 89.

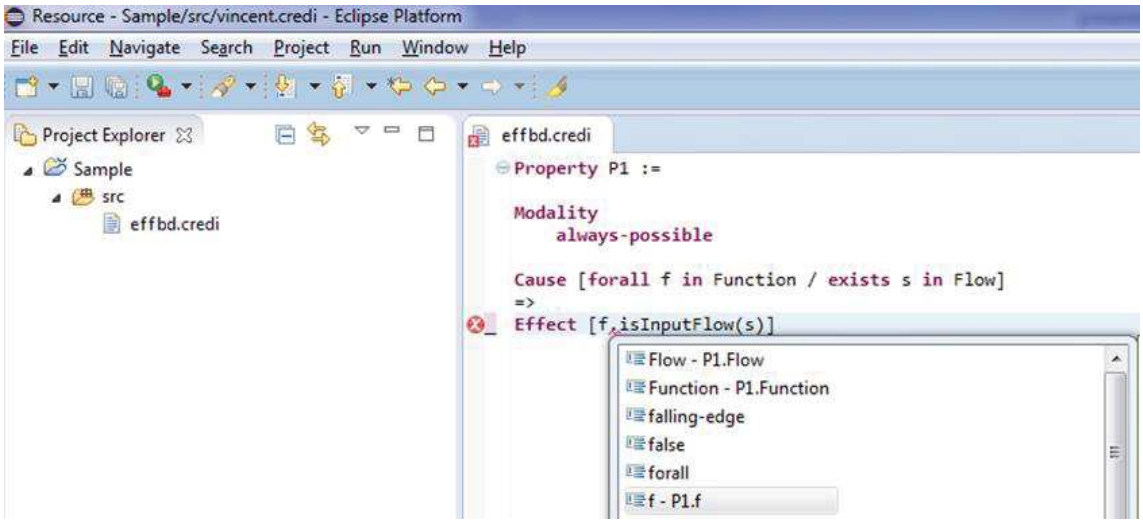


Figure 89. The CREI editor for constraint property modeling.

5.3.3 Model-checking tool

The UPSL framework does not currently support an adequate model checker, since it relies on third party approaches (i.e., UPPAAL and COGITANT). This requires the transformation of the source formal specifications into third party specification, as well as the constraint properties into third party constraint properties, as illustrates in Figure 90. Such V&V approaches benefit the reuse of a 3th party model checker. However, the V&V result must be interpreted back to the source space, a process that might result into an information loss.

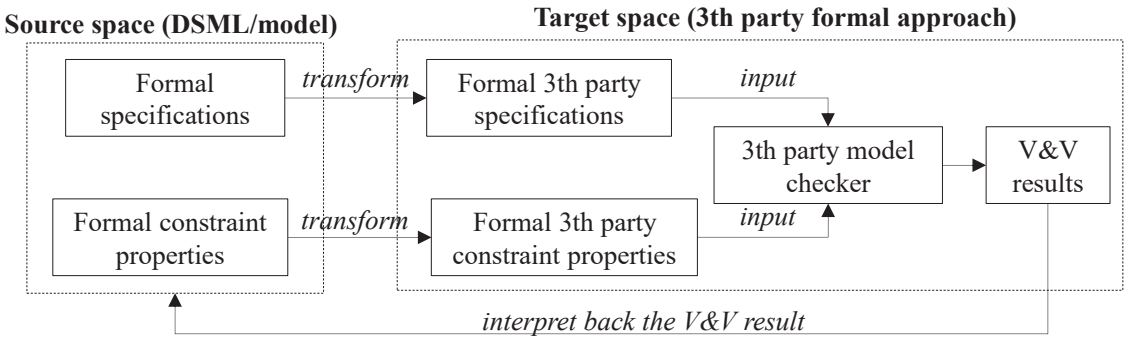


Figure 90. The current architecture of UPSL for V&V based on third party approaches.

We propose in this section a new architecture for V&V for the UPSL framework that is partially based on third party approaches. The goal is 1) to benefit from existent 3th party model checker, rather than designing new one, 2) while obtaining a V&V result that is directly interpretable on the formal specification without interpreting it back as classically proposed. This new architecture is illustrated in Figure 91. It requires

transforming the formal constraint properties into a third party properties that can be verified by a 3th party model checker based on the source formal specification. The V&V result is directly interpretable for the source specification (i.e., it doesn't need to be interpreted back).

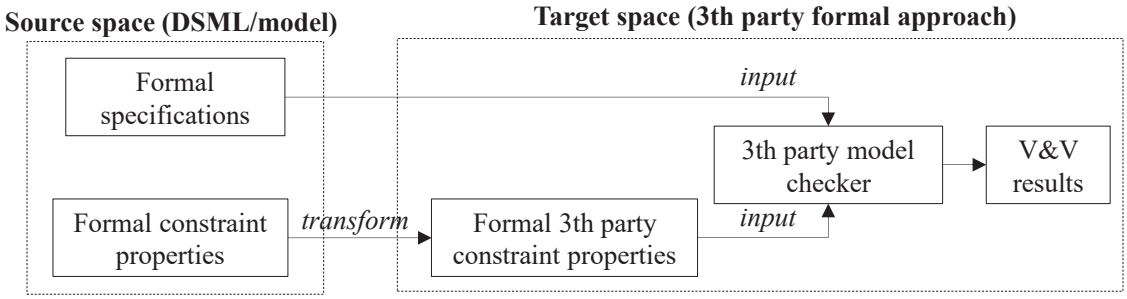


Figure 91. The new architecture of UPSL for V&V.

The choice of the formal third-party approach is crucial. Namely, the third-party approach must be equipped with a model-checker that can verify the 3th party constraint properties based on the source formal specification. For example the a-temporal structural constraint properties can be transformed into OCL properties. The OCL interpreter can then be used for verification directly based on the formal structural specification, i.e., the SP (structural properties) and the MSP (model structural properties). We propose here-after several rewriting rules for the transformation of CREI a-temporal structural constraint properties into OCL constraints.

Literals rewriting rules: CREI literals are directly rewritten into OCL. There is no need to modify them:

- CREI Number to OCL Number (ex: 15; 2; 1.54;)
- CREI Boolean to OCL Boolean (true; flase;)
- CREI String/Char to OCL String/Char (ex: name; age ; s)
- CREI Predicate (method call) to OCL Predicate (method call, e.g., getName(); setAget(36)

Expressions rewriting rules: the structure of CREI expressions is directly rewritten into OCL.

- Additive expression (+ | -)
    - a+b
- Relational Expression (< | > | >= | <= | = | !=)

    - a!=b
- Boolean Expression (and | or | xor)
    - true or false
- Set Expression (union | intersect | difference)
    - A union B

Quantifier expressions rewriting rules: the structure of CREI quantifiers expressions is rewritten into OCL as described in the next:

Let x be a variable, X be a set (class) and *expr* be an expression and let everything in red be optional:

- *FOR ALL quantifier expressions rewriting rules*: [∀x ∈ X |$expr$/] is rewritten into [eContents(X)->forAll(x : X | *expr*)/]
- *AT LEAST ONE quantifier expressions rewriting rules*: [∃x ∈ X |$expr$/] can be rewritten into [eContents(X)->one(x : X | *expr*)/]

Combining for all and at least one:

- [∀x ∈ X |∃y ∈ Y | $expr$/] can be rewritten into [eContents(X)->forAll(x : X | eContents(Y)->one(y : Y | *expr*)/]
- [∃x ∈ X |∀y ∈ Y | $expr$ /] can be rewritten into [eContents(X)->one(x : X | eContents(Y)->forAll(y : Y | *expr*)/]

Relating causes and effects rule: the structure of CREI properties is composed of causes and effects, related by a relation and a set of potential indicators. Our approach currently supports only the implication relation (=>) that is transformed to the "*implies*" OCL function, neglecting the complementary information defined by the relation R.

We propose below two examples written in a natural language, their specification in CREI and their transformation to OCL constraints.

Example 1: "all Persons that have a car must be majors"

- CREI : [$\forall p \in Person \mid p.cars <> null\ implies\ p.age > 18$ /]
- OCL / Acceleo [eContents(Person)->forAll(p | p.cars <> null *implies* p.age>18/]

Example 2: "all Persons must have at least one 'Renault' car"

- CREI : [$\forall p \in Person \mid \exists c \in Car \mid c.type = 'Renault'\ and\ c \in p.cars$ /]

- OCL: [eContents(Person)->forAll(p | eContents(Car)->one(c | c.type = ‘Renault’ and p.cars.includes(c))/]

Unfortunately, for behavioral properties (temporal and a-temporal) we are facing the issue related with the extensibility of xviCore for promoting any behavioral modeling language. Indeed, the possible variability of the form of the behavior designed by different behavioral modeling language does not allow choosing one third party approach. However, formal verification of behavioral properties (temporal and a-temporal) can be achieved, if the used behavioral modeling language are supported by an adequate mechanisms for model checking or proof, as discussed in Chapter IV for the eISM language. In the case of eISM, CREI properties can be rewritten in LTL and then verified based on the Temporal Boolean Difference (TBD). For instance, if the behavior is designed as timed automates, CREI properties can be transformed into computational tree logic (CTL) and the UPPAL environment can be used for formal verification and proof.

5.4 Conclusion

This chapter introduces an approach for system modeling and V&V denoted “xviCore”. xviCore is composed of four language: a metamodeling language, a language for concrete syntax, a behavioral modeling language and a constraint modeling language, along with a mechanism for simulation and a mechanism for formal property proof. The mechanism for simulation is based on the blackboard design pattern, a multiscale time, a reconciliation rule, a cadence rule and an execution scheduling algorithm that includes stability management. The mechanism for formal proof introduces a new architecture for V&V that is partially based on transformation techniques. The goal is to benefit from existent 3th party model checker, rather than designing new one, while obtaining a V&V result that is directly interpretable on the formal specification without interpreting it back as proposed by classical transformation approaches.

Conclusion and Perspectives

Summary

Within the context of MBSE (model based systems engineering) or MDE (model driven engineering) models are first class citizens. They are to be created and managed, checked and simulated prior to any use for discussion, deliberation or decision. Models support stakeholders and increase their confidence during decision making processes. The made decisions impact the development of the real system, up until its deployment on site, its exploitation and even its dismantling. Namely, they impact on system's functioning, safety, security, induced costs, and so forth. It is thus very important to assure the quality of models before making any decision by applying model verification and validation (V&V) activities. However, this is currently an ongoing issue in both MBSE and MDE. This thesis contributes on the matter, focusing on two general problems:

(1) the design of modeling languages

(2) the verification and validation of models

To this end, we propose a new tool-equipped method allowing 1) to create dedicated modeling languages, denoted Domain Specific Modeling Languages (DSML), 2) to compose (syntactically and semantically) different DSMLs, and 3) to include semantics (static and dynamic), a key-component for model V&V.

Our method is based on concepts for modeling, verification and validation, languages that formalize the means to design and manage the concepts and operating approach that put in use the languages for the design and V&V of models. This V&V includes a mechanism for simulation based on model execution, and opens the way to become able to use mechanisms for formal properties proof.

The concepts of the method consist of a typology of properties for modeling and a formalized lifecycle for property management. The typology consists of properties that are used to conceptualize domain knowledge, forming a modeling language, and properties that are used to concretize domain knowledge, forming a model. Namely, stakeholders must first conceptualize their domain knowledge through different types of modeling properties. This is done by using a design process that involves different types of languages. We distinguish:

- Structural properties (SP) and dependencies between structural properties (DSP) designed by a metamodeling language;
- Representational properties (RP) and dependencies between representational properties (DRP) designed by a concrete syntax language;
- Behavioral properties (BP) and dependencies between behavioral properties (DBP) designed by a behavioral modeling language;
- Constraint properties (CP) and dependency constraint properties (DCP) designed by a constraint modeling language;

Stakeholders can then use such DSMLs to concretize their domain knowledge. More specifically, they use:

- The SP and the DSP to design the structure of a model as model structural properties (MSP) and the model structural dependencies (MSDP)
- The RP and the DRP to design the representation of a model as model representational properties (MRP) and the model representational dependencies (MRDP)
- The BP and the DBP to parametrize the behavior for a model as model behavioral properties (MBP) and the model behavioral dependencies (MBDP)

Furthermore, system properties express a part of the system and stakeholders requirements of systems or stakeholders based on a modeling artefact that is defined by a modeling artefact. We distinguish two types of system properties: model constraint properties (MCP), object constraint properties (OCP), dependency model constraint properties (DMCP) and dependency object constraint properties (DOCP).

The management of these different types of properties is defined through two formalized lifecycles denoted respectively “DSML and model lifecycle” and “composite DSML and model lifecycle”. These lifecycles are composed of several phases and sub-phases. Each phase highlights which of the above quoted properties need to be designed and the V&V analyses that need to be performed.

Among the different types of languages for modeling different types of properties, the contribution of this work is based on behavioral languages for the design of DSML behavior (i.e., dynamic semantics) for MBSE. For this purpose, we propose first an evaluation of a well-known design pattern for executable DSML based on its effective adaptation in the field of MBSE. This is here applied to create an executable version of

a well-known language for MBSE experts, i.e., an executable eFFBD (enhanced Functional Flow Block Diagram), denoted xeFFBD. This application example allows us to highlight several issues, as well as possible improvements for the effective adaptation of this design pattern in the field of MBSE. Based on the feedback, we propose two languages that can be used to design the behavior of a DSML.

The first language is an extended version of the Interpreted Sequential Machine denoted (eISM). eISM is a behavioral language based on discrete-events hypotheses. In comparison to other discrete-events languages eISM has several advantages: it operates with typed input/output data and complex expressions build using types data, it separates classical state/transition specification from data specification, allowing the specification of some states using variables and it has formal underlying structure. For the design of executable DSMLs, eISM is integrated with the metamodeling language EMOF, creating an executable metamodeling language. In such a way, the behavior of a DSML is specified as a set of discrete-events behavioral models, each one associated to different domain concepts of the DSML abstract syntax.

The second language is a formal rule based language denoted FRBL. The goal of FRBL is to ease and assist the design of the behavior of a DSML that have formal pre-defined semantics based on the one hand, on the reuse of the DSML's predefined formal semantics and on the other hand, based on a generic template. The behavior of a DSML is finally specified as a set of formal rules, among which the following three rules are considered as main rules defined by the generic template: 1) read inputs, 2) calculate future state and 3) write outputs. The syntax and the semantics of FRBL and designed using the xText approach.

The operating demarche of our method for the design and V&V of models includes a mechanism for simulation based on model execution, and mechanisms for formal properties proof. The mechanism for simulation is based on the blackboard design pattern, a multiscale time, a reconciliation rule, a cadence rule and an execution scheduling algorithm that includes stability management. The mechanism for formal proof introduces a new architecture for V&V that is currently partially based on transformation techniques. The goal is to benefit from existent 3th party model checker, rather than designing new one, while obtaining a V&V result that is directly

interpretable on the formal specification without interpreting it back as proposed by classical transformation approaches.

List of contributions

This work presents five contributions.

The first contribution "Modeling based on Properties" introduced in Chapter III, presents a concept alignment between MDE and MBSE. It aligns the components of DSMLs (abstract syntax, concrete syntax, static semantics and dynamic semantics) and models considering the four selected SE challenges for MBSE introduced in Chapter I and the SE vision on the concept property (system properties and modeling properties). This contribution is applied on two thread examples, one demonstrating the design of DSMLs based on the eFFBD and PBD languages, and the other demonstrating the design of models based on one eFFBD model and one PBD model for the functional and physical architectures of a fire and flood detection system. This contribution has also been presented during the international symposium of INCOSE and appear in the symposium's proceeding (Blazo Nastov et al. 2016b) and in the INCOSE's magazine INSIGHT (Blazo Nastov et al. 2016a).

The second contribution presented in Section 4.2 is a new approach for modeling dynamic semantics for executable DSMLs for MBSE. This approach responds to the 5 raised issues related to executable DSML for MBSE discussed in Section 4.1.2: *(1) state notion and formalization, (2) improved readability, (3) transient states detection and management, (4) mechanism for formal proof* and *(5) designing dependencies in modeling languages – a way for model interoperability*. As a starting point, we chose the ISM formal behavioral modeling language because it covers *issue 1, issue 2* and *issue 4*. We propose then an extended version (eISM) that covers also *issue 3* and *issue 5*. As an illustrative example, in Section 4.2.7 we propose an executable version of the WaterDistrib DSML. This contribution has been presented during two international conferences (CSD&M'14 and ENASE'16) and appear in the conferences' proceedings (Nastov et al. 2015; B. Nastov et al. 2016). The first paper validates the 5 raised issues related to executable DSML for MBSE as a result to the design of an executable version of the eFFBD language by using an MDE approach for executable DSMLs. The second paper validates the ISM language and the extended version of ISM as a response to these 5 issues for the modeling of dynamic semantics for DSMLs.

The third contribution “Formal Rule Based Language – FRBL” introduced in Section 4.3 is a new approach for an assisted design of dynamic semantics for a particular category of DSMLs that have formal pre-defined semantics based on discrete-events (DE) hypothesis, such as ISM. FRBL is based on two principles. The first principle is the “reuse of the predefined formal semantics of DE languages”. For this purpose, the syntax of FRBL is designed to be similar to formal semantics, easing the reuse of the pre-defined formal semantics of DE languages. The second principle is the “design based on templates”. The FRBL language proposes a generic template that can be reused for any DE language based on three main rules: 1) Reading Inputs Rule, 2) Calculating Future State Rule and 3) Writing Outputs Rule. As illustration, we design the dynamic semantics of eISM (formalized in Section 4.2.2) using FRBL.

The fourth contribution presented in Section 5.2 introduces a mechanism based on the Blackboard design pattern and an original execution scheduling algorithm that includes (1) a multiscale time, (2) a reconciliation rule, (3) a cadence rule and optionally (4) stability management, for coordinated execution of behavioral models from one or several DSMLs allowing the execution of models created by these DSMLs. The contribution is applied on two thread examples, on the WaterDistrib DSML allowing the execution of WaterDistrib models and on the eFFBD / PBD DSMLs allowing coordinated execution of eFFBD and PBD models. This contribution has been presented during the international conferences ENASE’16 and appear in the conferences’ proceedings (B. Nastov et al. 2016).

The last contribution presented in Section 5.3 introduces a mechanism for formal proof by property verification. The verification process is preformed directly on SoI models without using exogenous transformations. We reuse the UPSL-SE framework for the design of all types of properties.

Limitations and perspectives

We note hereafter the main limitations having to be studied and developed in order to improve the proposed method.

First, model interoperability working hypothesis introduced in Chapter III (see Section 3.3) does not take consideration to semantic interoperability problematic when designing a composite DSML. This must include detection and management of classical

semantic problems (e.g. same name for concepts or reversely different names for defining a common concept or a concept shared between two view points, and so on).

Second, even if partial transformations can remain necessary and beneficial for formal properties proof by promoting the use of existing model checkers such as UPPAL, TINA, SPIN or other, it is today requested to study and develop proof mechanisms adapted for instance to CREDI and FRBL modelling languages.

Third, the centralized data exchange mechanism for model execution promoted by the chosen Black Board design pattern does not currently consider fully concurrent data access and management (e.g. potential deadlocks on access). This problematic is studied in various domains such as Data Bases access and management so we think that it is necessary to give a particular attention to the existing solutions.

Fourth, FRBL language for rules modeling can be enriched by considering mechanisms for rules prioritization and scheduling, and possibly massive parallelism execution allowing simulation optimization.

Fifth, Continuous and Hybrid behavioral models must be considered by our method.

Last, the current tools that support the proposed method must be rapidly disseminated in the SE community in order to test and improve all the proposed contributions.

REFERENCES

Abrial, J.-R., 2005. *The B-book: assigning programs to meanings*, Cambridge University Press.

AFIS, 2012. *Ingénierie système: la vision AFIS pour les années 2020-2025* A. Kerbrat, ed., AFIS (French Association for Systems Engineering) [in French]. Available at: https://books.google.mk/books/about/Ing%25C3%25A9nierie_syst%25C3%25A8me.html?id=hTZ6MwEACAAJ&redir_esc=y.

Alagar, V.S. & Periyasamy, K., 2011. Vienna Development Method. *Specification of Software Systems*, 1996(8), pp.405–459. Available at: http://dx.doi.org/10.1007/978-0-85729-277-3_16.

Association for Computing Machinery, 2015. Computing degrees and careers. *computingcareers.acm.org*, p.Web. Available at: http://computingcareers.acm.org/?page_id=6.

Bérard, B. et al., 2013. *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer. Available at: https://books.google.fr/books?id=xJGqCAAAQBAJ&dq=systems+and+software+verification&lr=&hl=fr&source=gbs_navlinks_s.

Bertot, Y., 2006. *Coq in a Hurry*, Available at: http://arxiv.org/abs/cs/0603118.

Bettini, L., 2013. *Implementing Domain-Specific Languages with Xtext and Xtend*, Available at: http://www.packtpub.com/implementing-domain-specific-languages-with-xtext-and-xtend/book.

Bézivin, J. et al., 2006. Model Transformations? Transformation Models! In *International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. Genova, Italy: Springer publishing, pp. 440–453. Available at: http://www.springerlink.com/index/31pt5242j7745410.pdf.

Bézivin, J., 2005. On the unification power of models. *Software and Systems Modeling*, 4(2), pp.171–188.

CESAR, 2012. Cost-efficient methods and processes for safety relevant embedded

systems. Available at: http://www.cesarproject.eu/.

Chapurlat, V., 1994. *CSY-R: un modèle de spécification, conception et simulation de la commande de systèmes discrets complexes répartis*. University of Montpellier II.

Chapurlat, V., 2013. UPSL-SE: A model verification framework for Systems Engineering. *Computers in Industry*, 64(5), pp.581–597. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0166361513000468.

Chapurlat, V., 2008. *Vérification et validation de modèles de systèmes complexes: application à la Modélisation d'Entreprise*. University of Montpellier II [HDR in French]. Available at: https://tel.archives-ouvertes.fr/tel-00204981/en/.

Chapurlat, V. & Daclin, N., 2013. Proposition of a guide for investigating, modeling and analyzing system operating modes: OMAG. In A. Marc et al., eds. *Proceedings of the Poster Workshop at the 2013 Complex Systems Design and Management Conference (CSDM 2013)*. Paris, France, pp. 87–97. Available at: http://ceur-ws.org/Vol-1085/09-paper.pdf.

Chein, M. et al., 2009. *Graph-based knowledge representation: computational foundations of conceptual graphs*, Available at: http://books.google.be/books?id=iz3y6WK2EMEC%5Cnhttp://books.google.com/books?hl=en&lr=&id=iz3y6WK2EMEC&oi=fnd&pg=PA1&dq=Graph-based+Knowledge+Representation+Computational+Foundations+of+Conceptual+Graphs&ots=Tij6hoRdZo&sig=kcby1q3i6iMxELo_V0ezMm0Z32k.

Chomsky, N., 1965. *Aspects of the Theory of Syntax*,

Combemale, B. et al., 2008. A Property-driven approach to formal verification of process models. In *International Conference on Enterprise Information Systems*. Springer Verlag, pp. 286–300.

Combemale, B. et al., 2009. Essay on semantics definition in MDE: An instrumented approach for model verification. *Journal of Software*, 4(9), pp.943–958.

Combemale, B. et al., 2013. Reifying concurrency for executable metamodeling. In *International Conference on Software Language Engineering*. Springer, pp. 365–384.

Combemale, B., 2016. The GEMOC Initiative: on the globalization of modeling languages. Available at: http://gemoc.org/.

Combemale, B., Crégut, X. & Pantel, M., 2012. A Design Pattern for Executable DSML. *The 19th Asia-Pacific Software Engineering Conference (APSEC)*, pp.282–287.

Czarnecki, K. & Helsen, S., 2003. Classification of Model Transformation Approaches. In *the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. pp. 1–17. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.6773&rep=rep1&type=pdf.

Doran, T., 2006. IEEE 1220: For practical systems engineering. *Computer*, 39(5), pp.92–94.

Douglass, B.P., 2002. Real-Time UML. In *the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002), Co-*

sponsored by IFIP WG 2.2. Oldenburg, Germany: Springer, pp. 53–70.

Engelmore, R. & Morgan, T., 1988. *Blackboard systems* R. Engelmore & T. Morgan, eds., Wesly publishing.

Faunes Carvallo, M., 2013. *Improving automation in model-driven engineering using examples*. University of Montréal. Available at: https://papyrus.bib.umontreal.ca/xmlui/handle/1866/10562.

Fleurey, F., 2006. *Langage et méthode pour une ingénierie des modèles fiable*. University of Rennes I. Available at: https://tel.archives-ouvertes.fr/tel-00538288.

Friedenthal, S., Moore, A. & Steiner, R., 2014. *A practical guide to SysML: The systems modeling language*, Morgan Kaufmann.

Greenfield, J. & Short, K., 2003. Software factories: assembling applications with patterns, models, frameworks and tools. In *In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. John Wiley & Sons, pp. 16–27. Available at: http://dl.acm.org/citation.cfm?id=949348.

Harel, D., 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), pp.231–274.

Harel, D. & Naamad, A., 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), pp.293–333.

Hausmann, J.H., 2005. *Dynamic Meta Modeling – A Semantics Description Technique for Visual Modeling Languages*. University of Paderborn. Available at: http://is.uni-paderborn.de/uploads/tx_sibibtex/Dynamic_Meta_Modeling_-_A_Semantics_Description_Technique_for_Visual_Modeling_Languages.pdf.

Hegedüs, Á., Horváth, Á. & Varró, D., 2015. A model-driven framework for guided design space exploration. *Automated Software Engineering*, 22(3), pp.399–436. Available at: http://link.springer.com/article/10.1007/s10515-014-0163-1.

Hilding Elmqvist, 1997. Modelica — A unified object-oriented language for physical systems modeling. *Simulation Practice and Theory*, 5(6), p.p32. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0928486997842577.

Holzmann, G.J., 1997. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), pp.279–295.

IEC, 1999. IEC 60848: Specification language GRAFCET for sequential function charts. , p.94. Available at: http://snmaicpc.chez.com/pdf_zip/MAI2/cours/norme grafcet.pdf.

IEC, 1992. *Programmable controllers: standard IEC 61131-3*, IEC. Available at: https://webstore.iec.ch/publication/4552.

INCOSE, 2016a. Guide to the Systems Engineering Body of Knowledge (SEBoK). *v. 1.6*. Available at: http://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK).

INCOSE, 2016b. INCOSE's data-base for Systems Engineering tools. Available at: https://acc.dau.mil/CommunityBrowser.aspx?id=530621.

INCOSE, 2010. *Systems engineering handbook. A guide for system life cycle processes and activities*,

INCOSE, 2007. Systems Engineering Vision 2020. *INCOSE-TP-2004*. Available at: http://www.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf.

ISO/IEC, 2008. *ISO/IEC 15288 Systems and software engineering - System life cycle processes*, IEEE Standard 15288-2008. Available at: http://www.canieti.com.mx/assets/files/828/ISO_IEC_FDIS_15288.pdf.

Juliot, E. & Benois, J., 2010. Viewpoints creation using Obeo Designer or how to build Eclipse DSM without being an expert developer.

Kahani, N. & R. Cordy, J., 2015. *Comparison and Evaluation of Model Transformation Tools*, Ontario, Canada. Available at: http://research.cs.queensu.ca/TechReports/Reports/2015-627.pdf.

Kleppe, A.G., 2007. A language description is more than a metamodel. In *the 4th International Workshop on Software Language Engineering*. Nashville, USA. Available at: http://doc.utwente.nl/64546/.

Koenig, D. et al., 2007. *Groovy in action (Vol. 1)*, Manning Publications.

Kohavi, Z. & Jha, N.K., 2009. *Switching and Finite Automata Theory*, Cambridge University Press. Available at: https://books.google.fr/books?id=Qv0LBAAAQBAJ.

Lalanda, P., 1997. Two complementary patterns to build multi-expert systems. *Pattern Languages of Programs*, pp.1–9. Available at: http://hillside.net/plop/plop97/Proceedings/lalanda.pdf.

Lamine, E., 2001. *Définition d'un modèle de propriété et proposition d'un langage de spécification associé : LUSP*. University of Montpellier II. Available at: http://www.theses.fr/2001MON20205.

Larnac, M. et al., 1995. Formal representation and proof of the interpreted sequential machine model. In *Computer Aided Systems Theory — EUROCAST'97*. Springer, pp. 93–107. Available at: http://link.springer.com/chapter/10.1007%252FBFb0025037.

Larsen, K.G., Pettersson, P. & Yi, W., 1997. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2), pp.134–152.

Lee, E., 2003. Overview of the Ptolemy Project (Technical Memorandum UCB/ERL M03/25). *Electrical Engineering*, pp.1–36. Available at: http://ptolemy.eecs.berkeley.edu/conferences/97/ilp_overview.pdf.

Long, J.E., 2007. MBSE in Practice: Developing Systems with CORE.

Mahfouz, A.A., Mohammed, M.K. & Salem, F.A., 2013. Modeling, Simulation and Dynamics Analysis Issues of Electric Motor, for Mechatronics Applications, Using Different Approaches and Verification by MATLAB/Simulink. *International Journal of Intelligent Systems and Applications*, 5(5), p.39.

Maiden, N.A. & Ncube, C., 1998. Acquiring COTS software selection requirements. *IEEE Software*, 15(2), pp.46–56.

Markovic, S. & Baar, T., 2008. Semantics of OCL specified with QVT. *Software and*

Systems Modeling, 7(4), pp.399–422.

Mathworks, 2014. Introduction to Simulink. *Matlab Simulink User's Guide R2014b*, pp.1–69.

Mavin, A. et al., 2009. Easy Approach to Requirements Syntax (EARS). In *The 17th IEEE International Requirements Engineering Conference (RE 2009)*. Atlanta, USA, pp. 317–322. Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5328509&isnumber=5328460.

Mayerhofer, T. et al., 2013. xMOF: Executable DSMLs based on fUML. In *In Software Language Engineering*. pp. 56–75.

Mens, T. & Van Gorp, P., 2006. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(1–2), pp.125–142.

Le Moigne, J.-L., 1999. *La modélisation des systèmes complexes* Dunod, ed., Available at: http://www.dunod.com/sciences-sociales-humaines/psychologie/psychologie-sociale/master-et-doctorat/la-modelisation-des-systemes-complexes.

Muller, P.-A., Fleurey, F. & Jézéquel, J.-M., 2005. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*. pp. 264–278. Available at: http://www.springerlink.com/content/l60r44862137214x.

Murata, T., 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4), pp.541–580.

Nastov, B. et al., 2016a. A Tooled Approach for Designing Executable and Verifiable Modeling Languages. *INSIGHT the quarterly magazine of International Council of Systems Engineering (INCOSE)*, 18(4), pp.31–33.

Nastov, B. et al., 2015. A verification approach from MDE applied to model based systems engineering: XeFFBD dynamic semantics. In *Complex Systems Design and Management - Proceedings of the 5th International Conference on Complex Systems Design and Management (CSD&M 2014)*. Paris, France: Springer publishing, pp. 225–235.

Nastov, B. et al., 2016. Towards semantical DSMLs for complex or cyber-physical systems. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*. Rome, Italy: Scitepress publishing.

Nastov, B. et al., 2016b. Towards V&V suitable Domain Specific Modeling Languages for MBSE: A tooled approach. In *the 26th Annual INCOSE International Symposium (IS 2016)*. Edinburgh, Schotland: Wiley publishing.

Nastov Blazo, 2014. Contribution to model verification: operational semantic for System Engineering modeling languages. In *the 3th National Conference on Software Engineering (CIEL 2014)*. Paris, France, pp. 88–90. Available at: http://ciel2014.i3s.unice.fr/Ciel2014_fichiers/ActesCiel2014.pdf.

Nipkow, T., Paulson, L.C. & Wenzel, M., 2002. *Isabelle/HOL: A proof assistant for higher-order logic*, Springer. Available at: http://books.google.com/books?hl=en&lr=&id=KjLZvSUbKvQC&

oi=fnd&pg=PA1&dq=Isabelle+-+A+Proof+Assistant+for+Higher-Order+Logic&ots=0_xrw6pfCJ&sig=h4nUaCdrLs0nJKXWoFhXkp2FAGg.

OMG, 2015a. *Meta Object Facility (MOF) Specification 2.5*, Available at: http://www.omg.org/spec/MOF/2.5/.

OMG, 2014. *Object Constraint Language (OCL) Specification v2.4*, Available at: http://www.omg.org/spec/OCL/2.4.

OMG, 2015b. *Systems Modeling Language (SysML) Specification 1.4*, Available at: http://www.omg.org/spec/SysML/1.4.

OMG, 2011. *Unified Modeling Language (UML) Specification 2.4.1*, Available at: http://www.omg.org/spec/UML/2.4.1/.

Pfister, F. et al., 2012. A proposed meta-model for formalizing systems engineering knowledge, based on functional architectural patterns. *Systems Engineering*, 15(3), pp.321–332. Available at: http://doi.wiley.com/10.1002/sys.21204.

Pfister, F., Huchard, M. & Nebut, C., 2014. A framework for concurrent design of metamodels and diagrams towards an agile method for the synthesis of domain specific graphical modeling languages. In *Proceedings of the 16th International Conference on Enterprise Information Systems (ICEIS 2014)*. SciTePress, pp. 298–306. Available at: http://www.scopus.com/inward/record.url?eid=2-s2.0-84902356552&partnerID=tZOtx3y1.

Pnueli, A., 1977. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp.46–57. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4567924.

Rivera, J.E. & Vallecillo, A., 2007. Adding behavioral semantics to models. In *Proceedings of the IEEE International Enterprise Distributed Object Computing Workshop (EDOC 2007)*. pp. 169–180.

Rozenberg, G. & Ehrig, H., 1997. Handbook of graph grammars and computing by graph transformation. *Handbook of Graph Grammars*. Available at: http://www.ulb.tu-darmstadt.de/tocs/52752569.pdf.

Rozier, K.Y., 2011. Linear Temporal Logic Symbolic Model Checking. *Computer Science Review*, 5(2), pp.163–203.

Sadilek, D.A. & Wachsmuth, G., 2009. Using grammarware languages to define operational semantics of modelled languages. In *In International Conference on Objects, Components, Models and Patterns*. Springer, pp. 348–356.

Scheidgen, M. & Fischer, J., 2007. Human comprehensible and machine processable specifications of operational semantics. In *Model Driven Architecture-Foundations and ...*. Springer, pp. 157–171. Available at: http://link.springer.com/chapter/10.1007/978-3-540-72901-3_12.

Schmidt, D.C., 2006. Model-Driven Engineering. *IEEE Computer*, 39(2), pp.25–31. Available at: http://www.cs.wustl.edu/~schmidt/PDF/GEI.pdf.

Seidner, C., 2009. *Vérication des EFFBDs : Model checking en Ingénierie Système*. University of Nantes. Available at: https://tel.archives-ouvertes.fr/tel-00440677.

Sendall, S. & Kozaczynski, W., 2003. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5), pp.42–45. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1231150.

Stachowiak Herbert, 1973. *Allgemeine Modelltheorie*, New York: Springer-Verlag. Available at: https://books.google.de/books?id=DK-EAAAAIAAJ.

Stålhane, T., Farfeleder, S. & Daramola, O., 2011. *Safety analysis based on requirements*,

Steinberg, D. et al., 2008. *EMF: eclipse modeling framework*, Pearson Education. Available at: http://portal.acm.org/citation.cfm?id=1197540.

Tolk, A. & Muguira, J., 2003. The Levels of Conceptual Interoperability Model. *Fall Simulation Interoperability Workshop*, (September), pp.1–9.

Vandermeulen, E., 1996. *Machine Séquentielle Interprétée: un modèle à états pour la représentation discrète et la vérification de systèmes*. University of Montpellier II. Available at: http://www.sudoc.fr/005446457.

Vandermeulen, E. et al., 1995. The temporal boolean derivative applied to verification of extended finite state machine. *Computer and Mathematics with application*, 30(2).

De Weck, O.L., Ross, A.M. & Rhodes, D.H., 2012. Investigating Relationships and Semantic Sets amongst System Lifecycle Properties (Ilities). In *The 3th International Engineering Systems Symposium CESUN 2012*. Delft, Netherlands.

Weisemöller, I. & Schürr, A., 2008. Formal definition of MOF 2.0 metamodel components and composition. In *In International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 386–400.

Woodcock, J. et al., 2009. Formal Methods: Practice and experience. *ACM Computing Surveys*, 41(4), pp.1–36.

Ziemann, P. & Gogolla, M., 2003. OCL Extended with Temporal Logic. *Perspectives of System Informatics*, 2890/2003, pp.617–633. Available at: http://www.springerlink.com/content/p71n7hwfy76xvdcv/.

Abstract. Within the context of organizational and engineering sciences, Systems Engineering (SE) is an interdisciplinary and collaborative approach for the design, realization and management of large scale complex systems. Among other processes, SE promotes modeling during all the design stages of a system; it can then be characterized as Model Based Systems Engineering (MBSE). In parallel to SE, within the field of software engineering for complex or cyber-physical systems the Model-Driven Engineering (MDE) takes an important role, providing the means for systems modeling through creation, checking and manipulation of various models.
Generally, on the one hand, modes represent a system under design (i.e., a system of interests - SoI) based on different viewpoints (i.e., requirements, functional, physical, performance, etc.) and on the other hand, models are used by stakeholders to verify and validate the modeled SoI, i.e., to assure that the SoI meet stakeholders' expectations and requirements (for example in terms of covering the needs, operational safety, production and use costs, etc.). This implies concepts, techniques and tools for creating and managing various SoI models (denoted viewpoint models) for the purpose of stakeholders, and for reaching and improving the quality of models helping then stakeholders during decision-making processes, to make decisions faster and efficiently with enough confidence. Indeed, these decisions impact all along the downstream phases of system engineering and development until the realization and deployment of the real system, its functioning, safety, security, induced costs and so on.
In this work, a particular attention is given to model verification and validation (V&V). The goals are to assure prior to decision-making processes, first, that models are coherent, well-formed and correctly build and represented, and second, that they are trustworthy and relevant, representing as accurately as possible the viewpoints of a system under design as expected by stakeholders. Such models provide stakeholders with confidence and trust, aiding them in making, but also in arguing decisions.
Models are created by using modeling languages that are specifically tailored for a given viewpoint of a system, denoted Domain Specific Modeling Languages (DSMLs). The basic principles on which a DSML is based are its syntax and its semantics, but current DSMLs have been more studied from the syntactical point than from the semantical one that is often neglected or, when needed, provided by transforming the DSML into external formal approaches. This is, from our perspective, a limitation for the deployment of V&V strategies in the MBSE context. To overcome this issue, we propose a new method denoted xviCore (executable, verifiable and interoperable core) for the design of DSMLs that can be used to design models that respect the needs of system architects having the required level of quality (discussed above). Our method is conceptualized as a meta-modeling language that combines four languages for the design of DSML syntax and semantics. xviCore includes concepts and mechanisms for simulation (i.e., model execution) and formal proof based directly of SoI models without transforming them to other third-party approaches as proposed by classical approaches for modeling and V&V. In addition, xviCore relays on a formalized design process denoted DSML and Model lifecycle. Finally, xviCore is tool-equipped as a deployable plugin within the Eclipse Modeling Framework (EMF) environment.
This thesis reflects the description of the xviCore method. The first chapter exposes the context and the problematic of this work. The rest of the thesis outline is highlighted in conclusion of the first chapter.