A Verification Approach from MDE Applied to Model Based Systems Engineering: xeFFBD Dynamic Semantics

Blazo Nastov, Vincent Chapurlat, Christophe Dony, François Pfister

A verification approach from MDE applied to Model Based Systems Engineering: xeFFBD dynamic semantics

Blazo Nastov [1], Vincent Chapurlat [1], Christophe Dony[2] and François Pfister[1]

bstract. Model Based System 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*" [1]. Among other principles, it promotes creating and analyzing models all along systems engineering. These models are used to discuss, to argue and finally to make decisions that impact the achieved system (in terms of functioning, costs, safety, etc.). One of the main expectations of MBSE is to permit engineers to dispose of models with a high level of confidence. For this purpose, several model Verification and Validation (V&V) approaches exist, aiming to ensure models' quality in terms of construction (models are correctly built) and in terms of relevance for reaching design objectives and stakeholders' requirements. This paper aims at discussing and evaluating an approach originally developed in the field of Model Driven Engineering by proposing some adaptations. The approach is illustrated on a well-known functional modeling language dedicated to MBSE field.

1. Introduction

Systems Engineering (SE) [2] is a process and a system thinking oriented approach for designing complex systems. In this field, Model Based System Engineering (MBSE) is considered as "*the formalized application of* [system] *modeling"* [3] all along engineering activities requested in SE processes [1]. A model represents a system under study so called System of Interest (SOI), considering it under one of the various possible views e.g. functional, behavioral, physical, requirements, contextual, etc. Once created, models allow system engineers to argue and make architectural decisions. These decisions have impact on the realized system, its functioning, its safety, induced costs and so on [4]. Therefore, engineers should have a high level of confidence in models they are handling, before making any decision based on them. So first, models have to be "well-constructed" (correctly built, respecting (meta) modeling requirements and building rules) and second, they have to be the "right models" in terms of 1) relevance for reaching design objectives and 2) respect to a part of, or all, stakeholder's requirements. Moreover, the level of confidence increases if models verify their "well-constructiveness" and "model-rightness", considered separately and in interaction with other models of the SOI.

[1] LGI2P, Ecole des Mines d'Alès, Parc Scientifique G. Besse, 30000 Nîmes, France
{Blazo.Nastov, Vincent.Chapurlat, Francois.Pfister}@mines-ales.fr

[2] LIRMM, 161 rue Ada, 34392 Montpellier, France – dony@lirmm.fr

Classically, models are the subject of study of Model Driven Engineering (MDE) [5] and nowadays they are built using Domain Specific Modeling Languages (DSMLs). So first and foremost is the creation of DSMLs, a process, defining the languages' abstract and concrete syntaxes. An abstract syntax is provided by a metamodel representing, through a graph of classes and associations, the concepts of a domain and their relations. A concrete syntax defines how instances of abstract syntax concepts (forming a model) are concretely represented in a textual or graphical form. In MBSE context, we focus on graphical representation of models. So proposing the abstract syntax and a graphical concrete syntax of a DSML makes it operational to create models, each of them seen as a graphical representation of a particular point of view of a SOI. The literature highlights several frameworks for creating DSMLs and graphical editors [6], [7], [8]. Unfortunately, such editors are of "two dimensional drawing-board" nature because they do not deal with model well-constructiveness or rightness. As a consequence, created models turn-out to be "simple two dimensional drawings" of modeled concepts and relationships.

The main idea of this work is to add a third dimension, allowing designers to create models that can be simulated and questioned, achieving some of the Verification and Validation (V&V) objectives and to ensure the coherence between all models of the same SOI. Some solutions are proposed in the field of MDE [9], [10], [11], but unfortunately they remain, not or hardly applied in the field of SE due to their insufficiency for reaching SE objectives. In this paper, we discuss and evaluate an existing approach coming from MDE by applying it on a DSML, often considered by systems engineers and architects as a relevant functional description language named eFFBD [12] in order to emerge an extension that reaches SE objectives.

The paper is structured as follows. Section 2 discusses the importance of DSML semantics and introduces approaches and concepts that allow defining such semantics. Some limitations of studied approaches are discussed and afterwards an approach exceeding discussed limitations is proposed. Section 3 evaluates that approach by applying it on the eFFBD language in order to create an executable extension. Section 4 proposes contributions that allow the adaptation of the approach in the field of SE, before concluding about research perspectives in Section 5.

2. Towards execution and validation of DSMLs

Abstract syntaxes of DSMLs partially define language semantics through their underlying structure and the vocabulary naming concepts and relationships. Unfortunately, such semantics may sometimes be ambiguous, since different engineers may have different understanding of a single model. Therefore, in order to have equal and non-ambiguous understanding, it is essential to define in a precise and non-ambiguous manner DSMLs semantics.

Semantics are either *static*, independent of any behavior, or *dynamic*, describing the dynamic comportment of models' elements (can be advisedly called "dynamic model" or "dynamic comportment" or "DSML behavior"). There are three ways to formalize dynamic semantics description. First, *operational semantics* describes model comportment as a sequence of states, transitions between states and a machine that executes such a state model. Second, *denotational (translational) semantics*

transforms DSML concepts into other DSML concepts with predefined dynamic comportment. Last, *axiomatic semantics* describes in a declarative way the evolution of model properties [13]. In this paper we focus on defining DSMLs behavior using dynamic semantics.

Literature highlights several approaches and tools for defining dynamic semantics for a given DSML. For instance, Kermeta [9] is an executable metamodeling language that defines operational semantics for a given DSML (in imperative way). Another example is the Atlas Transformation Language (ATL) [14] that (in declarative way) defines operational semantics through endogenous transformations and denotational semantics through exogenous transformations. Additionally, metamodeling languages together with constraints definition languages can be used to define axiomatic semantics. Meta Object facilities (MOF) [15] is usually used to define metamodels and OCL (Object Constraint Language) [16] to add constraints to metamodel e.g. pre and post conditions, invariants and so on. However, these tools and approaches are related to software engineering and programming languages which somehow make them difficult to use for SE experts. Indeed, dynamic semantics of dedicated DSML is to be described and formalized with minimal efforts from experts by assisting them and automating the process as much as possible.

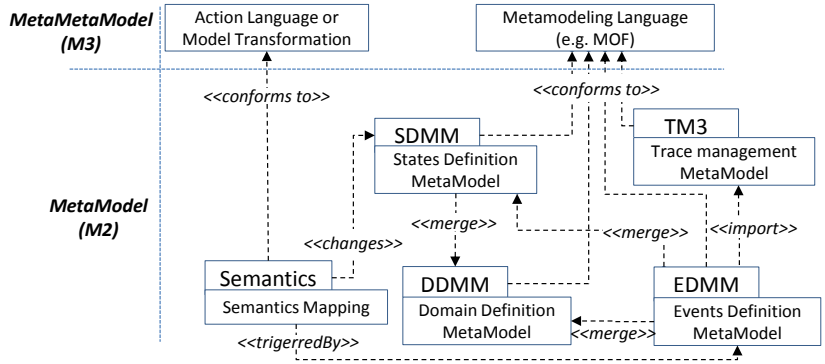

Figure 1: The executable DSML Pattern [11]

Another approach, supporting state-based execution of DSMLs is proposed in [11]. The approach is schematized in Figure 1 as a pattern composed of four structural parts related to each other and of a fifth part providing the dynamic semantics relying on the previous four. Modeling concepts and relationships between them are defined in the *Domain Definition MetaModel (DDMM)* package. The DDMM does not usually contain execution-related information. Such kind of information is defined in the *State Definition MetaModel (SDMM)* package through several sets of states assigned to DDMM concepts that can evolve during execution. Model execution is described as successive state changes of DDMM concepts provoked by stimuli. The *Event Definition MetaModel (EDMM)* package defines different types of stimuli (events), together with their relationship to DDMM concepts and SDMM states. Applied stimuli are either injected by the environment (exogenous kind) or produced internally by the system in response to other stimuli (endogenous kind). The *Trace Management MetaModel (TM3)* provides monitoring mechanism of model execution by capturing

stimuli. The last and key part is the package *Semantics*, composed of evolution rules, describing how the running model SDMM evolves over DDMM concepts (changing their state) according to the stimuli defined in the EDMM. Evolution rules can be either defined as operational semantics or as denotational semantics.

3. Application in the field of SE

This section proposes to extend the eFFBD functional modeling language by building and tooling its dynamics in order to become able to interpret and animate eFFBD models, and to prove their evolution rules. Similar solution has been proposed by [17] using translational semantics, transforming eFFBD models into Petri Nets models. [18] argues that translational semantics solutions limit V&V objectives. Fortunately, such limitations are exceeded by operational semantics that allow achieving V&V objectives on a source model rather than a third party model. So the expected result is xeFFBD i.e. an executable eFFBD that we consider as "self-sufficient" in achieving these V&V objectives. In [19] a short history and various evolutions of a particular DSML named FFBD (Functional Flow Block Diagram) and its main evolution named eFFBD (enhanced FFBD) is presented. It is considered as a functional-modeling language preferred in the SE community [20]. This DSML provides system designers with a framework to describe the behavior of complex, distributed, hierarchical, concurrent and communicating systems [21]. For instance, Figure 2 shows the functional architecture of an interface highlighting parallelism, selection, loop and other complex constructs allowed in eFFBD.

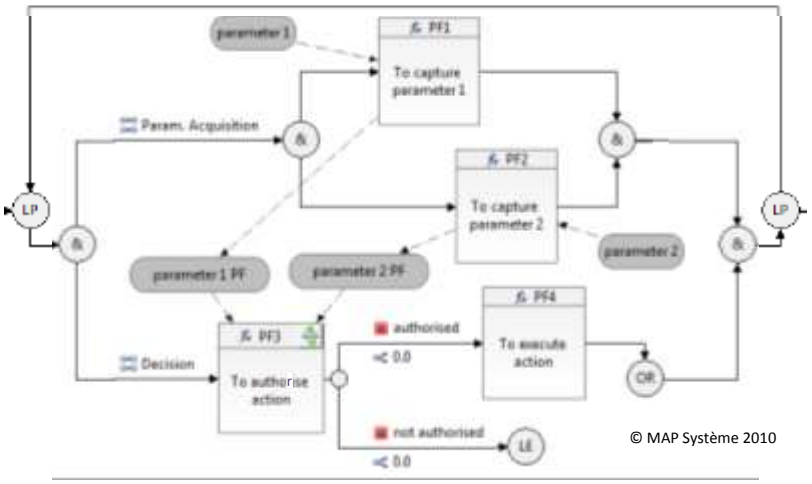

Figure 2: eFFBD example

Our proposition to achieve executable eFFBD along the above discussed methodology is presented in the following section.

Creating an executable DSML is divided into two major phases. First, a language executable metamodel is defined containing domain (in DDMM), execution-related

(in SDMM and EDMM) and monitoring (in TM3) information. Second, semantics are defined describing how execution-related information evolves over domain concepts.

3.1 Phase 1: executable metamodel definition

Executable metamodel definition phase includes four construction stages: DDMM definition, SDMM definition, EDMM definition and TM3 definition. We apply such process to eFFBD as shown in Figure 3.

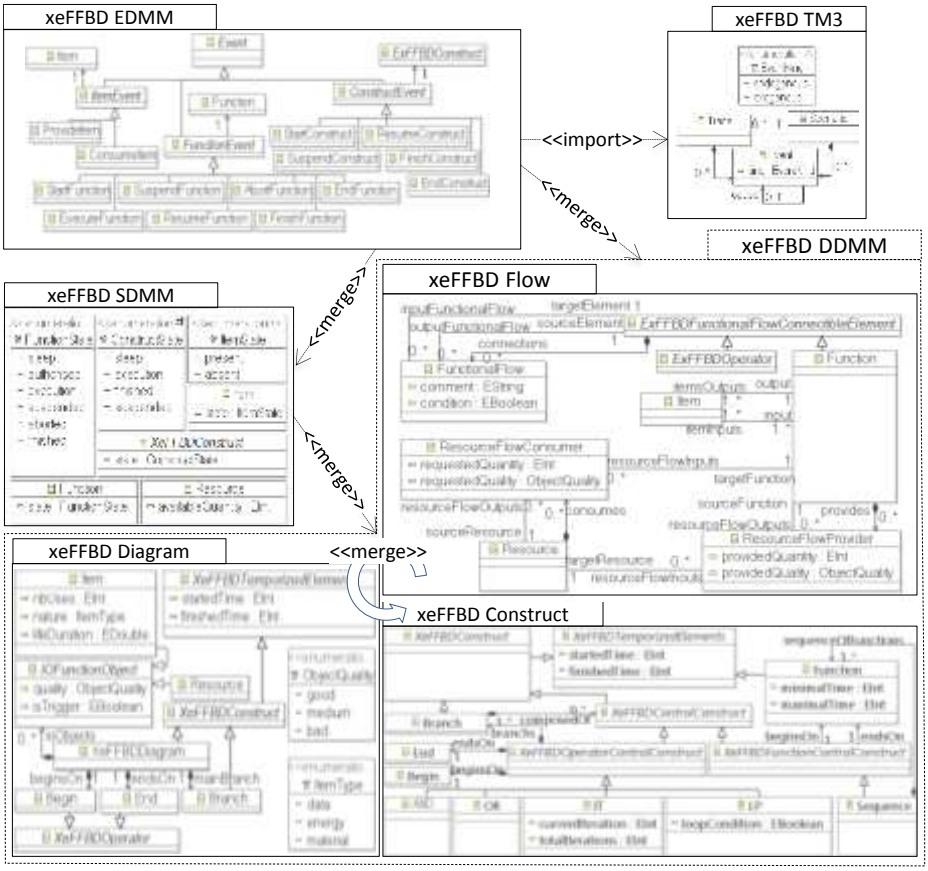

Figure 3: Creating executable metamodel for eFFBD - construction stages

First, language or domain concepts and relationships are defined. This construction stage is identical to a (non-executable) DSML metamodel definition. The created metamodel is called DDMM since it contains domain concepts and relationships. In order to reduce language complexity and to improve readability and understandability, we split the DDMM into three packages: *xeFFBD Diagram*, *xeFFBD Construct* and *xeFFBD Flow*, each one representing a different aspect of the eFFBD language. *xeFFBD DDMM* is then created by merging all three packages,

using the “merge” predefined package operator of MOF.

The xeFFBD language defines three kinds of core elements: Function, Resource and Item. Function describes 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.

xeFFBD Diagram is the core package describing a xeFFBD 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 several control constructions named exFFBD Constructs, described hereafter. 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 represents different constructions recurring into a xeFFBD Diagram. These constructions allow engineer to describe how functions are chained and the different manners of their execution, introducing the possibility to describe function parallelism, sequence, exclusion, and selection. A construct can either be 1) a function control construct composed of a set of functions (eventually one unique function) put in a 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 describes three types of flows that can be represented in a xeFFBD model: 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 construction stage consists in defining SDMM. In this stage, we define in the package xeFFBD SDMM, the possible states of some of previously defined domain concepts. First, domain concepts that may evolve have to be chosen. For instance, we chose the following: Construct, Function, Item and Resource. The third construction stage consists in defining the events requested for the evolution of evolving concepts together with their relationship with corresponding concepts. Such information is defined in the EDMM. For instance, we defined three types of events: construct event, function event and item event.

Additionally, Figure 4 shows finite states automate associated to the concept Function. Here, on the one hand, function states are represented by automates' states and on the other hand, different types of function event are represented by automates' transitions. The evolution of concepts is represented as transition firing. In this sense, we consider here 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*) e.g. in case of dysfunction of the component on which the function has been allocated.

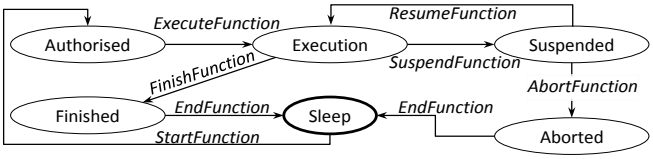

Figure 4: A finite states automate for the concept Function representing state evolution as transition firing

In the case of Item and Resource, the state model is replaced by defining state variables named quantity and quality allowing us to reduce the number of possible states. For example, a resource of oil can be in state 50 liters, but also in state 100 liters if a function provides as output another 50 liters of that same resource.

The executable metamodel definition process ends by defining a monitoring mechanism considered as fourth stage of this process. For this, we propose the generic trace mechanism described in the approach, the TM3 package, shown in Figure 3.

3.2 Phase 2: semantics definition

As previously defined, xeFFBD metamodel contains execution-related (dynamic) information (e.g. packages SDMM and EDMM). Yet, xeFFBD metamodel is static, until the package Semantics (Figure 1) is defined. This package defines how and when dynamic information is executed on domain concepts, allowing state and property changes. For this, we adopt a proposed property-driven approach detailed in [22]. This approach describes how to formally define execution rules under the form of properties (described below), and how to become able to check some of those properties. Three types of properties can be expressed: structural properties, temporal properties and quantitative properties. The approach distinguishes properties checked on each model execution called universal properties from those checked once called existential properties.

Model evolution is first, defined through universal and existential properties by preconditioning events, second, through transitions that are defined between domain concepts states and finally, through event-based transition firing. When fired, transitions invoke state changing of domain concepts. Figure 4 illustrates different states of Function, event-based transitions between states and corresponding events. For instance, if event *StartFunction* is applied on an instance of *Function* that is in state *Sleep*, a transition is fired changing its state into *Authorized*.

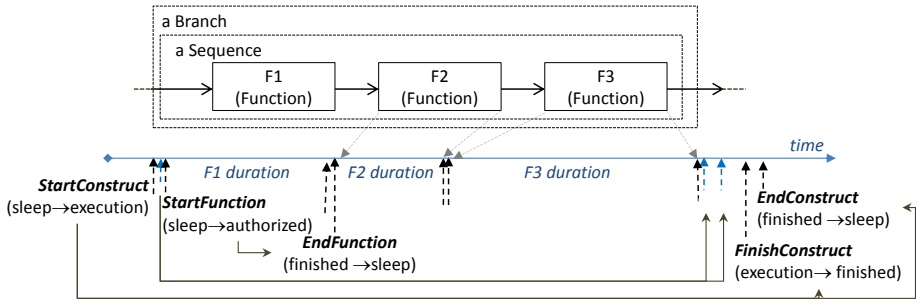

Figure 5: execution of a simple exFFBD model

We consider that execution of lower level embedded constructions is controlled (*i.e.* started and finished) by higher level embedding constructions respecting an ordering given by a functional flow. Figure 5 illustrates an example of such execution control describing applied events and invoked state changes of components represented by different type of arrows. A simple xeFFBD diagram is represented by a starting point (entering arrow), an ending point (exiting arrow) and a main branch. A sequence is placed inside that branch, containing three functions: F1, F2 and F3. In this example, input and output object flows are not represented in order to ease the readability of the figure. 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 a sequence contains functions, it also controls their execution. First, it starts the beginning function (F1), and afterwards it waits for finished functions, to end their execution and to start the execution of following functions (end F1 and start F2). This process is repeated until the ending function (F3) ends and then the sequence itself finishes execution. The main branch ends the sequence, before finishing its proper execution. The diagram

waits for the main branch to finish execution, in order to end it. The end of the main branch execution means that the diagram can first, finish and then end its execution.

Functions are contained in a sequence, so the formal definition of their execution starting and ending is defined by the dynamic behavior of the Sequence construct as described in previous example. Next, due to lack of space we formally define only a dynamic behavior of functions using the previously described property-driven approach. 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 6, Eq.1) before being able to make the real transformation of energy, material and / or data (Figure 6, Eq.2) providing then the outputs items and resources and finishing its execution respecting minimal and maximal execution time (Figure 6, Eq.3).

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 6: Semantics mapping by defining evolution properties of Function concept

Let us note that due to external events, a function can be suspended temporarily, can resume its execution or can abort (*Suspended*, *Aborted*). These external events can be then shared with other constructs from other modeling languages. For instance, the function behavior can depend on the component behavior that performs this function. So, the event Suspended can be a common event shared between xeFFBD and a future DSML named xPBD (executable Physical Block Diagram currently under study).

4. Application discussion and expected contributions

State notion and formalization. Considered approach describes concepts' execution as successive state change. This induces to define a set of states considered by the user as sufficient for his V&V objectives. Unfortunately, some concepts, such as Resource from xeFFBD, may be characterized by a continuum of states. For this, we propose three solutions. The first solution consists in defining a finite number of states. For instance, resource states model can be reduced to a two state model, containing: *sufficient* or *insufficient* states. However, this solution is too limitative for V&V objectives. The second solution consists in introducing a set of state variables

describing possibly infinite number of states that can evolve continuously. This solution allows describing high level of detail. For instance resource state model can be represented by *quality* and *quantity* variables. The third one consists in mixing the previous two solutions, linking discrete states defined in the state model and state variables. For instance, Item state model is composed of a state quality variable and two states: present and absent. These three solutions are applied in SDMM package and are now under development.

Towards condition and event based transition approach. In order to understand concepts' evolution, one has to simultaneously visit three packages: DDMM, SDMM and EDMM, and the evolution rules defined in the semantics package. We consider that this makes created languages difficult to read and understand. In order to ease readability and improve understandability, we propose a representation of previously stated packages, using finite state automata. Conditions and events are responsible for a transition firing. First, a Condition (True by default) is a Boolean function computed on variables, attributes of any concept from the local DDMM and external variables corresponding to other concepts from another DDMM. So *inter* and *intra* conditions are distinguished. Intra conditions have to be satisfied by a currently manipulated model, while inter conditions correspond to conditions that have to be satisfied by one or several other models from the same SOI whose behavior interacts with the behavior of studied model. Second, it is always possible to distinguish two events and there exist a default event *e* always occurring. A Transition can be then fired when associated event is received, and if and only if, associated condition is verified. The work now consists of formalizing these notions and linking the transition behavior with discrete event system theory.

Towards model transient states detection and management. Temporal evolution rules (named properties by [13]) are currently defined using Temporal OCL (TOCL). This induces the examination of defined properties taking into account a unique scale of time. However, the notion of "*model stability*" is by hypothesis essential for models representing critical, parallel or distributed systems. A "*transient state*" of a concept is a state such that it is possible to change that state without modifying the inputs, as defined for instance in the case of Sequential Function Chart in [23]. A model is stable if and only if each instance of a modeling concept used in this model is itself in stable state. We propose here to extend considered approach by a double scale of time named external and internal time modeled by two independent logical clocks. Values of each variable v_i appearing in conditions and occurring events associated to transitions of a state model M are read and then frozen in external time. M evolves taking into account v_i by using then an internal scale when performing execution rules allowing then to detect transient states and to reach the next stable state of M. The external time depends on environment evolution and is a logical modeling of physical scale time. It is defined as a set of moments ordered by taking into account events apparition. It is initialized when a simulation starts. The internal time is initialized at each moment defined in external time and there are no common temporal dimensions between internal and external scales. The evolution algorithm allowing transient states detection is schematized in Figure 7.

Towards properties modeling languages and checking techniques. Literature highlights several property-driven approaches with associated V&V techniques [24], [25] that will be explored and applied in the prosed frame of work.

Towards modeling languages and models interoperability. As illustrated in

Section 3, we propose to become able to link formally the resulting interpretation and execution of several DMSLs each dedicated to the description and the analysis of a view of a given SOI (behavioral, physical, functional, etc.). This will allow contributing to become able to check the coherence of SOI models even considering different points of view and different modeling objectives. It is a question of linking the dynamic semantics i.e. SDMM and EDMM have to be extended introducing requested and shared concepts and evolution rules.

Tooling. Unfortunately, tools supporting the considered approach do not exist at this moment. An extension of Diagraph [6] is now under development taking into account proposed improvements.

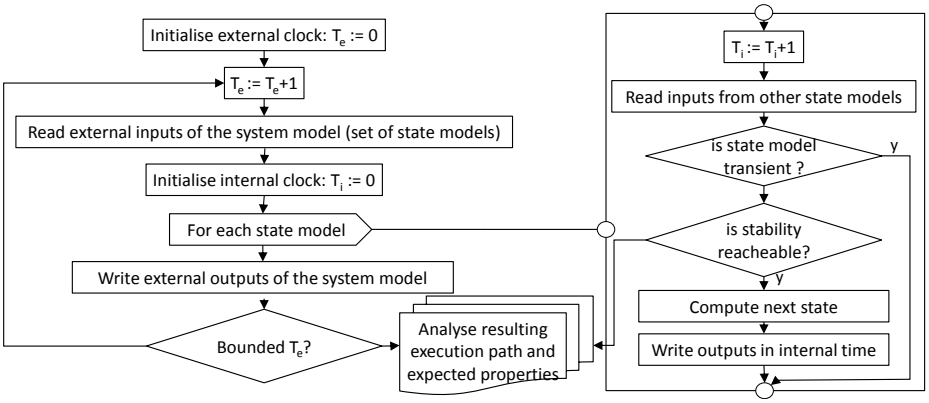

Figure 7: proposed evolution algorithm including stability reaching objectives

5. Conclusion and Perspectives

This paper presents an approach from the field of MDE for defining semantic of a DSML. The approach is considered here as a formal and relevant way for achieving models V&V objectives. It is applied to a functional modeling language largely used in MBSE domain. This application however, makes appear some questions that seem crucial and remain partially or completely uncovered. Conceptual as technical improvements are then proposed in order to complement this approach. The research and development work is now on going intending to fully support executable DSMLs creation process and to deploy it on MBSE domain.

References

1. INCOSE, “Systems Engineering Vision 2020,” INCOSE-TP-2004, September, 2007.
2. ISO/IEC, ISO/IEC 15288 : Systems and software engineering - System life cycle processes, vol. 2008, no. 1. IEEE, 2008, p. 5.
3. J. a Estefan, “Survey of Model-Based Systems Engineering (MBSE) Methodologies 2 . Differentiating Methodologies from Processes , Methods , and Lifecycle Models,” Jet

Propuls., 2008, vol. 25, pp. 1–70.
4. BKCASE Project, “System Engineering Book of Knowledge,” SEBoK v1.2. [Online]. Available: http://www.sebokwiki.org/.
5. S. Kent, “Model Driven Engineering,” Integr. Form. Methods, pp. 286–298, 2002.
6. F. Pfister, V. Chapurlat, H. Marianne, and C. Nebut, “A light-weight annotation-based solution to design Domain Specific Graphical Modeling Languages,” in Proceedings of Modelling Foundations and Applications - 9th European Conference, 2013.
7. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, EMF: Eclipse Modeling Framework. Addison-Wesley Professional, 2008, p. 744.
8. N. Pontisso and D. Chemouil, “TOPCASED Combining Formal Methods with Model-Driven Engineering,” 21st IEEE/ACM Int. Conf. Autom. Softw. Eng., 2006.
9. J. M. Jézéquel, O. Barais, and F. Fleurey, “Model driven language engineering with Kermeta,” in Lecture Notes in Computer Science, 2011, vol. 6491 LNCS, pp. 201–221.
10. S. J. Mellor and M. J. Balcer, Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley Professional, 2002, p. 416.
11. B. Combemale, X. Crégut, and M. Pantel, “A Design Pattern for Executable DSML,” in The 19th Asia-Pacific Software Engineering Conference (APSEC), 2012, pp. 282 – 287.
12. DoD, “Systems Engineering Fundamentals,” Def. Acquis. Univ. Press, 2001.
13. B. Combemale, “Approche de métamodélisation pour la simulation et la vérification de modèle -- Application à l’ingénierie des procédés.,” Phd - INPT, 2008 [in French].
14. F. Jouault, F. Allilaire, and J. Bézivin, “ATL: a QVT-like transformation language,” Companion to 21st ACM SIGPLA, 2006, pp. 719–720.
15. OMG, “MOF Core specification,” v2.4.1, 2013. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/PDF/.
16. OMG, “OCL : Object Constraint Language,” v2.4, 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4.
17. C. Seidner, “EFFBDs Verification: Model checking in Systems Engineering,” Pdh University of Nantes, 2009 [in French].
18. V. Chapurlat and C. Braesch, “Verification, validation, qualification and certification of enterprise models: Statements and opportunities,” Comput. Ind., 2008, pp. 711–721.
19. C. Haskins, K. Forsberg, and M. Krueger, “Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities”, Systems Engineering, no. August. INCOSE (International Council on Systems Engineering), 2011.
20. H. Chesnut, Systems Engineering Methods. Wiley & Sons, 1967.
21. B. Aizier, V. Chapurlat, S. Lisy-Destrez, D. Prun, C. Seidner, and J.-L. Wippler, “xFFBD: towards a formal yet functional modeling language for system designers,” in 22nd Annual INCOSE International Symposium, 2012.
22. B. Combemale, X. Cregut, P.-L. Garoche, X. Thirioux, and F. Vernadat, “A Property-Driven Approach to Formal Verification of Process Models,” in Enterprise Information Systems, 2009, pp. 286–300.
23. IEC 60848, Specification language GRAFCET for sequential function charts. Second edition, 2000.
24. P. Dasgupta, A roadmap for formal property verification. Springer, 2010.
25. V. Chapurlat, “UPSL-SE: A model verification framework for Systems Engineering,” Comput. Ind., 2013, vol. 64, no. 5, pp. 581–597.