A Concept Map of Terminologies and Disciplines for the Executable Model Lifecycle

Bruno G. A. Lebtag[1], Paulo G. Teixeira[1], Mohamad Kassab[2], Valdemar Vicente Graciano Neto[1]

[1]Institute of Informatics – Federal University of Goiás (UFG)
Post code 43017-6221 – 74.690-900 – Goiânia – GO – Brazil

[2]Pennsylvania State University
Post code 814-865-4700 – Pennsylvania, USA.

{bruno.gabriel,paulogabriel}@inf.ufg.br

muk36@psu.edu, valdemarneto@ufg.br

Abstract. *The Executable Models (ExM) research area is an ascending discipline that explores the use of models capable of being executed during the software development process. The research area is overloaded with terms and concepts derived from several areas. There is also ambiguities and an absence of consensus on definitions. This is a problem not only for newcomers in the area but it can generate unproductive studies and conflicts among researchers. However, there is not yet a study presenting those concepts in a concise and structured manner. Thus, this study aims to collaborate with the research area with a collection of relevant concepts and their definitions. The concepts were obtained from a literature review where we collected relevant studies from several disciplines. We organized those concepts into a concept map that establishes the relationship between concepts while presenting them separated by disciplines.*

1. Introduction

ExM are a technology that uses models defined by an executable modeling language that offers execution semantics to be used to describe those models and thus making them executable [Hojaji et al. 2019]. ExM brings to Software Engineering (SE) and Model driven engineering (MDE) the benefits of using models to interrogate, interact and manipulate the software being developed in a simplified manner while removing the problem of having to manually update the models by providing mechanisms to automatically synchronize them with the software [Vogel and Giese 2010].

Because ExM is still an ascending technology in the intersection of different computer science (CS) disciplines, there is a great number of concepts that newcomers in area will have to be familiar firstly in order to fully comprehend the research area. Moreover, from studies obtained [Levis and Wagenhals 2000, Dahmann et al. 2017, Hojaji et al. 2019] from a systematic mapping study (SMS) conducted by our group, we observed an absence of consensus on ExM concepts and definitions such as different definitions for ExM and ambiguities in the use of the terms: notations, language and formalism. Ambiguities and a lack of consensus can lead to unproductive researches and conflicts among researchers. To the best of our knowledge, there is no study yet tackling this problem and presenting the concepts in a concise and organized manner. Therefore,

this study aims at gathering relevant concepts that are used in the ExM research area obtained from a SMS. This study provides the concepts associated with their definitions and source references. We also organize those concepts into a concept map that establishes the relationship between concepts divided by CS disciplines.

This paper is organized as follows: Section 2 presents the methodology used to develop the study; Section 3 describes the important concepts involved in ExM research area; and Section 4 presents the discussion of the results together with the study limitations; finally, Section 5 concludes the paper with final remarks.

2. Methodology

Figure 1 present the methodology used in this research. We used as a basis the systematic and empirical methodology reported in the work of Dias-Neto *et al.* [Dias-Neto et al. 2010].

Figure 1. The methodology adopted in this study

We performed a SMS on ExM. Details are not provided here due to space restrictions, however the entire SMS protocol can be found externally[1]. We conducted the systematic mapping from September 2019 to February 2020 and we identified 311 studies, from which we included 51 studies, extracting information from them. We were able to collect a list of relevant studies and terms for ExM together with their definitions. A Complementary research was performed using Google Scholar[2] in grey literature (traditional CS textbooks and dictionaries) for additional definitions. In order to ease the understanding of the concepts and visualization of the relationships among terms, we elaborated the concept map we present herein.

3. Executable Model Concepts

ExM is a discipline in the intersection of many other CS disciplines such as programming language theory (PLT), Formal Methods (FM), Modeling and Simulation (M&S), Compiler Theory (CT) and MDE. Fig. 2 presents a *concept map* that encapsulates the main concepts and their relationships. We highlight that, for traceability and auditability purposes, we concatenate the given definitions with the source study from which it was acquired.

We present the concepts going from the most general to the most specific terms. A **notation** (blue area) is any system of graphic representation for presenting thoughts,

[1]https://bit.ly/3yn9NW1
[2]https://scholar.google.com.br/

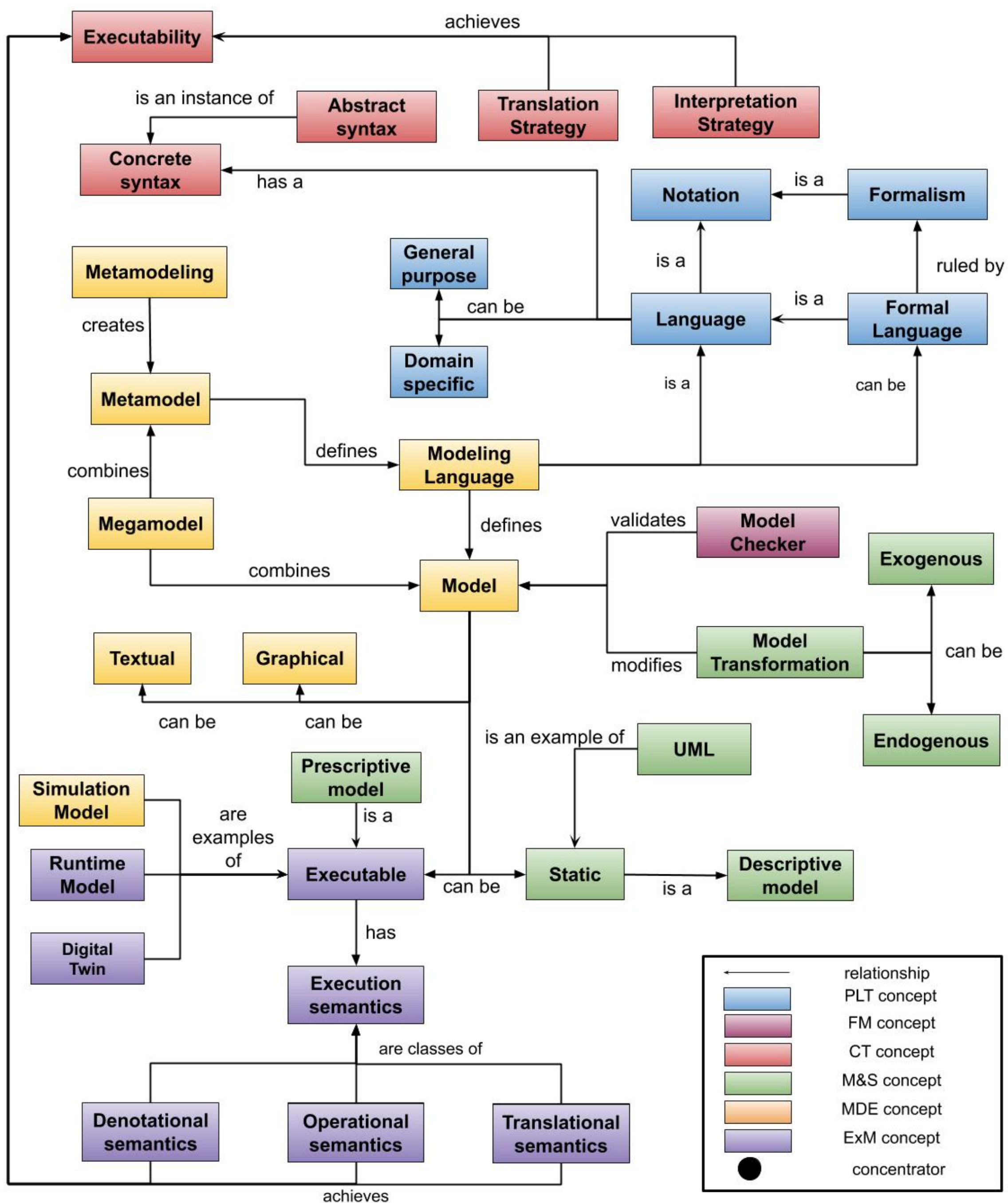

Figure 2. The proposed map for ExM concepts and their relationships.

concepts or ideas. It specifically refers to the set of symbols that allow linguistic analysis [Crystal 2008]. A **language** is a specialization of a notation which consists of a set of words belonging to an alphabet and a system for their use to communicate among peers [Rozenberg 1997]. It is an one-dimensional communication medium that is composed by a sequences of symbolic elements known as *constructs* [Reghizzi 2013, Sun et al. 2008]. In the context of computer programming languages, they are notations for describing to computer and people what machines are supposed to do [Aho 2006]. A language can be *general-purpose* (GPL), i.e. it is used to describe a broad set of contexts or

domain-specific (DSL), i.e. it is used only for a specific context (e.g. military, medical, etc.) [Selic 2008, Sun et al. 2008]. **Formalism** is another specialization of notation and it consists of rules and guidelines used in the process of formalization [Crystal 2008]. Formalization is the process of creating rules, principles and conditions that will govern the analysis of a specific notation in order to interpret it in a logical or mathematical way [Crystal 2008]. A **formal language** is a language that is ruled by a formalism [Rozenberg 1997].

ExM also import some terms and concepts from M&S research area. A **model** denotes an abstract representation of a object of interest[3], i.e. it abstracts out uninterested details [Selic 2008]. In general, a model can be anything capable of answer questions regarding the object of interest [ISO/IEC/IEEE:42010 2011]. We can also assume that the *source code* is a model since it is a simplified representation of the lower-level machine instructions and structures [Mens and Van Gorp 2006]. A model can also be described by a language [Rozenberg 1997]. A **model language** is a language used specifically to describe models. A modeling language can be *graphical* or *textual* [He et al. 2007]. Moreover, it can also be *general-purpose* (GPML) or *domain-specific* (DSML) [Selic 2008, Sun et al. 2008]. A model is considered to be a **descriptive model** when it is very abstract and potentially incomplete [Ciccozzi et al. 2019]. On the other hand, a model is considered to be a **prescriptive model** when it is precised and detailed enough to be executable [Ciccozzi et al. 2019].

A **metamodel** is a model of a model. It is a model describing a language/family of models [Favre 2011]. In a metamodel, we prescribe the models representing the constructs of a modeling language and their relationships. They are pre-requirements for performing automated model transformation [Mens and Van Gorp 2006]. A **megamodel** is the combination of different models, its elements and their associated metamodels in a single model while establishing the relationship among them [Mens and Van Gorp 2006, Vogel et al. 2011]. The process of defining the modeling languages is called **metamodeling**. In this process, it is defined the syntax of the language and its semantics [He et al. 2007]. The syntax consists of the concrete and abstract syntax.

Concrete and abstract syntax are terms derived from compiler theory [Aho 2006]. **Concrete syntax** is the grammar of the language and it is composed by the symbols and terminals string of its alphabet. When the concrete syntax is graphically represented in the format of a tree, the resulting tree is known as the *concrete syntax tree* (CST) [Aho 2006, Krahn et al. 2007]. The **abstract syntax**, on the other hand, is an "instantiated" concrete syntax derived from the actual model written in the language [Aho 2006, Krahn et al. 2007]. The tree representation of the abstract syntax is the *abstract syntax tree* (AST). Programs capable of parsing it will read the text and generate the AST representation in memory in order to validate whether the text was written according to the language grammar. It is also important to note that, although a language has only one grammar, i.e. only a single concrete syntax, it can have different AST representations due to ambiguities generated in the process of matching the text with the concrete syntax.

Model transformation is a technique extensively explored in MDE context that derives from Compiler Theory [Vogel et al. 2011, Favre 2011]. **Model transformation**

[3]We will use the term '*object of interest*' to denote any system where the term 'system' is not limited to computer systems.

is the process of automatic converting a source model into another target model by obeying a *transformation definition*. This definition is a set of *transformation rules* that describe how to convert one or more constructs from the source model into one or more constructs of the target model [Mens and Van Gorp 2006]. A model transformation between models expressed in the same language are named *endogenous transformations* or *rephrasing* whereas transformations performed between models of different languages are named*exogenous transformations* or *translation* [Visser 2001, Mens and Van Gorp 2006]. From Formal Methods and Compiler Theory we also have model checkers. A **model checker** is a software that is capable of verifying whether the models are compatible with a formal description of requirements using formal methods [Holzmann 1997].

In this sense, ExM import two techniques from MDE and Compiler Theory to achieve executability, namely: translation and interpretation [Ciccozzi et al. 2019]. In **translation strategy** a model written in a non directly executable notation is converted into an model capable to be executed or directly into machine instructions to be executed by the computer [Aho 2006, Ciccozzi et al. 2019]. In the **interpretation strategy** the model is read and executed by a software capable of interpreting the model and execute it [Aho 2006, Ciccozzi et al. 2019]. In the translation strategy, the general term for translation strategy is model transformation and the software that performs it is called *model transformer*. When the translation happens from textual model into machine instructions, the process is called *compilation* and the software that performs it is called *compiler*. When the translation happens between textual models written in different languages, the process is called *transpilation* and software that performs it is called *transpiler* [Aho 2006, Andres and Perez 2017, Ciccozzi et al. 2019].

Yet in the context of M&S, a **static modeling language** is a modeling language that statically addresses structural and behavioral views of a problem that does not vary along the time [Gomaa 2011]. A well-known example of a static model would be UML [ISO/IEC 19505-2:2012 2012]. A **static model** is a model described by a static modeling language. A UML class diagram model would be an example of a static model. Finally, by combining M&S and Compiler Theory, we can have ExM. An **executable model language** (ExML) is a modeling language that offers executable constructs or *execution semantics* that can be used to define and execute the model. Those constructs species the behavior of the models during execution [Hojaji et al. 2019]. A **executable model** is a model defined by an executable modeling language. The model contains aspects of the behavior of the system needed to be executed [Levis and Wagenhals 2000, Hojaji et al. 2019].

There are three different approaches for defining executable semantics: denotational semantics, translational semantics and operational semantics [Sun et al. 2008, Hojaji et al. 2019]. A **denotational semantics**, also known as *mathematical semantics*, describes mathematical constructs [Sun et al. 2008, Hojaji et al. 2019]. Each construct is associated to a mathematical object by mapping functions. A **translational semantics** defines the model transformation rules needed to convert the model into another executable model language [Hojaji et al. 2019]. An **operational semantics** defined the executable behavior of each construct in a virtual machine or interpreter [Sun et al. 2008, Hojaji et al. 2019].

Static models are produced during MDE process [Blair et al. 2009]. They are ”software models” that live above the code level and abstract low-level concerns. They are systematically employed and transformed during the software development [Blair et al. 2009, Vogel et al. 2011]. ExM, on the other hand, are considered to be **run-time models**, sometimes also referred as *models@run.time*. They are casually connected (i.e. not always connected) self-representation (i.e. up-to-date representation) of the associated system [Blair et al. 2009]. They emphasize in the structure, behavior and goals of the system from a *program space perspective* (i.e. not concerned with the physical implementation aspects) [Blair et al. 2009, Vogel et al. 2011].

ExM can also be used for simulating the system or real world physical elements. A **simulation model** is a type of executable model that will be executed by a simulation engine in order to study a problem with precise control and timing [Nutaro 2011, de França and Travassos 2015]. **Digital Twin** Also known as *computational megamodel*, *device shadow*, *mirrored system*, *avatar* or a *synchronized virtual prototype*, it is a type of ExM that is a virtual representation of a physical asset generated through data extracted from the real physical asset and simulation models in order to make predictions in real-time, optimization, monitoring, controlling and improved decision making [Rasheed et al. 2020].

While an ExM is executing some information can be stored for later analysis, a **model execution trace** captures all relevant information about the execution of a model [Hojaji et al. 2019]. This information can be any that occurs during the execution of the model such as execution states, events that occurred, state changes, processed inputs, and produced outputs.

4. Discussion

ExM, as any other research area, has its root in other well-established CS disciplines. Many of those origins are not so clear-cut and well-accepted. Besides, some concepts may exist in different disciplines meaning different things. From our previous SMS, we were able to observe an overload of terms derived from PLT, MDE and M&S. At the same time, we also observed an absence of agreement in their definitions. Terms such as language, notation and formalism were used without a clear distinction from each other.

The concept of models were also very ambiguous and broad. Some authors suggest they could be anything capable of answering questions regarding the object of interest [ISO/IEC/IEEE:42010 2011] while other aimed to be more precise and supported the idea of raising the level of abstraction [Selic 2008]. When we are dealing with SE, the end goal is often to ease the development process. Therefore, we support the notion of models as tools for raising the level of abstraction. We also envision ExM as a mean to increase the degree of computer-based automation for the software development process [Selic 2008].

From our SMS, we were only able to identify three definitions for ExM [Levis and Wagenhals 2000, Dahmann et al. 2017, Hojaji et al. 2019]. Many studies did not present any definition for ExM and simply assumed the readers knew what they were. Levis and Wagenhals is the older definition for ExM we found [Levis and Wagenhals 2000]. They do not present a precise definition but rather aim to create a notion of what ExM can be and their possible usages for architectural evaluation. Dahmann et al. also aimed to present their take on what ExM is and what it

can do. They do that by presenting a working example of a ExM used by DARPA[4] [Dahmann et al. 2017]. Hojaji et al. was the only study identified in our SMS that tempted to present a precise definition for ExM [Hojaji et al. 2019] and it was adopted herein for that precise reason. They define ExM in terms of an executable modeling language that offers execution semantics to be used to define the behaviors of the system being modeled.

Despite the best efforts, that definition also has some shortcomings. One may argue that translational semantics may represent a problem for the definition. Should a model translated to a third-generation language such as C++ or C# be considered ExM as they required an additional translation for low-level instructions to be executed? What could we said about directed-executed third-generation language such as javascript or python that they do not required additional translation to be executed? In our opinion, ExM should be, at most, directly converted once to be considered an ExM. Besides, they should also provide means to keep the model and the generated code synchronized. Thus, both answers should be 'no' since simple translation does not qualify as ExM. This also clearly separate ExM from traditional MDE approaches that offer only one-way translation.

Lastly, some studies mention that ExM can be used for simulation [Hojaji et al. 2019]. However they do not present the direct correlation between ExM and simulation models. For models@runtime and digital twin studies, we also noticed an absence in this correlation. Despite the fact they all clearly fit the definition for ExM.

Limitations of the Study The results presented herein were extracted from relevant studies from important authors we obtained from a SMS. However, this study is not an exhausted list of concepts related to ExM. Besides, someone may diverge from our perspective on which concepts should be considered relevant to ExM or the methodology we used. Therefore, in order to reduce bias, two experienced researchers in the area of MDE and ExM participated in the development of this study and revised the obtained results.

5. Final Remarks

This study contributes to ExM research area with a collection of relevant concepts and their definitions. The concepts were obtained from a SMS and a list of studies and books from important authors were used in the synthesis of the results. We also elaborated a concept map where we establish the relationships between concepts while presenting them divided by CS disciplines. As future work, we intend to conduct a survey with specialists in the field to evaluate the concept map represented in this paper. We also intent to evaluate the use of the concept map in disciplines of introduction to the topic.

References

Aho, A. V. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

Andres, B. F. and Perez, M. (2017). Transpiler-based architecture for multi-platform web applications. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*. IEEE.

[4]Defense Advanced Research Projects Agency

Blair, G., Bencomo, N., and France, R. (2009). Models@ run.time. *Computer*, 42:22 – 27.

Ciccozzi, F., Malavolta, I., and Selic, B. (2019). Execution of uml models: a systematic review of research and practice. *Software & Systems Modeling*, 18(3):2313–2360.

Crystal, D. (2008). *A dictionary of linguistics and phonetics*. Blackwell Publlishing,Credo Reference, Malden, Massachusetts Oxford England Boston, Massachusetts.

Dahmann, J., Markina-Khusid, A., Doren, A., Wheeler, T., Cotter, M., and Kelley, M. (2017). Sysml executable systems of system architecture definition: A working example. pages 1–6.

de França, B. B. N. and Travassos, G. H. (2015). Experimentation with dynamic simulation models in software engineering: planning and reporting guidelines. *Empirical Software Engineering*, 21(3):1302–1345.

Dias-Neto, A. C., Spinola, R., and Travassos, G. H. (2010). Developing software technologies through experimentation: experiences from the battlefield. In *XIII Ibero-American Conference on Software Engineering*.

Favre, J.-M. (2011). Towards a basic theory to model model driven engineering. pages 1–8.

Gomaa, H. (2011). *Static Modeling*, page 94–114. Cambridge University Press.

He, X., Ma, Z., Shao, W., and Li, G. (2007). A metamodel for the notation of graphical modeling languages. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 219–224.

Hojaji, F., Mayerhofer, T., Zamani, B., Hamou-Lhadj, A., and Bousse, E. (2019). Model execution tracing: a systematic mapping study. *Software and Systems Modeling*, 18(6):3461–3485.

Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295.

ISO/IEC 19505-2:2012 (2012). *Information technology — Object Management Group Unified Modeling Language (OMG UML) — Part 2: Superstructure*, 2000.

ISO/IEC/IEEE:42010 (2011). Iso/iec/ieee systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46.

Krahn, H., Rumpe, B., and Völkel, S. (2007). Integrated definition of abstract and concrete syntax for textual languages. In Engels, G., Opdyke, B., Schmidt, D. C., and Weil, F., editors, *Model Driven Engineering Languages and Systems*, pages 286–300, Berlin, Heidelberg. Springer Berlin Heidelberg.

Levis, A. H. and Wagenhals, L. W. (2000). C4isr architectures: I. developing a process for c4isr architecture design. *Systems Engineering*, 3(4):225–247.

Mens, T. and Van Gorp, P. (2006). A taxonomy of model transformation. volume 152, pages 1–17.

Nutaro, J. (2011). Building software for simulation: Theory and algorithms, with applications in c++. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*.

Rasheed, A., San, O., and Kvamsdal, T. (2020). Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE Access*, PP:1–1.

Reghizzi, S. (2013). *Formal languages and compilation*. Springer, London.

Rozenberg, G. (1997). *Handbook of formal languages*. Springer, Berlin New York.

Selic, B. (2008). Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3):379–391.

Sun, Y., Demirezen, Z., Lukman, T., Mernik, M., and Gray, J. (2008). Model transformations require formal semantics. pages 1–4.

Visser, E. (2001). A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109 – 143.

Vogel, T. and Giese, H. (2010). Adaptation and abstract runtime models. pages 39–48.

Vogel, T., Seibel, A., and Giese, H. (2011). The role of models and megamodels at runtime. In Dingel, J. and Solberg, A., editors, *Models in Software Engineering*, pages 224–238, Berlin, Heidelberg. Springer Berlin Heidelberg.