Unification or integration? The Challenge of Semantics in Heterogeneous Modeling Languages

Gabor Karsai

Institute for Software-Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
gabor.karsai@vanderbilt.edu

Abstract. Model-driven software development and systems engineering rely on modeling languages that provide efficient, domain-specific abstractions for design, analysis, and implementation. Models are essential for communicating ideas across the engineering team, but also key to the analysis of the system. No single model or modeling language can cover all aspects of a system, and even for particular aspects multiple modeling languages are used in the same system. Thus engineers face the dilemma of either defining a unifying semantics for all models, or finding a solution to the model integration problem. The talk will elaborate these problems, and show two, potential solutions: one using a model integration language (for the engineering design domain) and another one using explicit and executable semantics (for the domain of distributed reactive controllers).

The problem

Engineered systems are increasingly built using model-driven techniques, where models are used in all phases of the system's lifecycle, from concept development to product to operation. Models are built for everything: from the smallest part to the entire system, and models are used for all sorts engineering activities: from design to analysis and verification, to implementation and manufacturing. Engineering models are often based on domain-specific abstractions of reality; for example a finite-element model represents a 3D shape and an engineering assembly drawing represents how those shapes need to be joined together by some manufacturing steps to form an assembly. While there is a multitude of domain-specific models used in the design of a complex system, somehow these models have to 'fit together' because (1) they are describing the same, single system, and (2) they need to be combined to allow cross-domain, system-level analysis of the design. Models created in (domain-specific) isolation are necessary and very useful, but insufficient when the larger system is considered - in the larger systems subsystems and their components interact, and these interactions have to be expressed as well.

Obviously, the same applies to software systems: multiple, often domain-specific models are used to describe a complex system. Somewhat differently from conventional engineering, where a multiple, (physical) domain modeling tools are used, in software we tend to use multiple domain-specific modeling languages. Arguably, every modeling tool has a 'language' (explicitly defined or not) and a modeling language without a supporting tool is only partially useful; hence in this paper we will focus on the issue of the modeling languages and their semantics. Semantics is a central question in language engineering: how do we specify what 'sentences' of an artificial, engineered language mean? Fortunately, in the theory of computer languages there has been many decades of research that produced techniques for specifying semantics of languages. However, these specifications rarely span multiple, potentially different languages.

The problem at hand is stated as follows: How can we integrate heterogeneous, domain-specific modeling languages so that the instance models can be linked together and system-level analysis can be performed on these models? It is easy to see that this problem has multiple facets. Naturally, one problem is that of semantics: how do we 'integrate the semantics' of multiple modeling languages? Say, if we have a model $M_{L_1}(A)$ of a subsystem A in a modeling language L_1, and another model $M_{L_2}(A)$ of the same A in a modeling language L_2, what does the composition $M_{L_1}(A) \times M_{L_2}(A)$ mean (if it is meaningful at all)? Similarly, what if we compose the models of two different components, say A and B, and ask the same question about $M_{L_1}(A) \times M_{L_2}(B)$? Clearly, the semantics of composition has to be defined. Another problem is more operational: how do we manage complex 'model repositories' where the models of the system (or systems) are being kept? If changes are made in one model, what is the impact of these changes on the dependent and related models? Building and managing such model repositories brings up many deep technical and pragmatic problems.

The problem stated above has a close relationship to systems engineering. One of the main tasks in systems engineering is to discover, understand, and manage cross-domain and cross-system interactions that make the engineering of complex so difficult. Solutions, like SysML are certainly a good step in the right direction, but they are rather limited as far as semantics is concerned, and more research is needed to place them on a solid theoretical foundation.

Unification or Integration?

There seem to be (at least) two approaches to solving the problem. One can be called as 'unification', where we design a universal modeling language that magically unifies all existing modeling languages. Domain-specific models will then be translated into this unified modeling language, and analysis and verification will happen on the unified models. The semantics of all domain-specific modeling languages would have to be re-expressed in the unified language (i.e. in a common semantic domain). However, this approach seems very unrealistic: the domain-specific languages are typically rich, so coming up with a language that unifies them all is extremely difficult, their semantics sometimes does not align well, and creating a grand unified language does not seem feasible. Additionally, the set of languages to be integrated is changing from project to project, so a unified language will have to be extremely large. Arguably, the only 'language' that is common across domain-specific modeling is that of mathematics but this on such a high level that it is not pragmatic due to the loss of domain-specificity.

The other approach can be called as 'integration' where the focus is on integrating models: a model integration language (MIL) with 'sparse' semantics is used. The semantics of the model integration language is for capturing the cross-domain interactions in terms of the structure of the system. This model integration language is lightweight and (potentially) evolvable, so that new domain-specific modeling languages can be added to the suite as necessitated by the development project. In this approach domain-specific models 'stay' in their own modeling tools, and the integration models are reflections of these domain models. The integration models thus capture the interfaces of the domain models relevant for analyzing the interactions.The interfaces of the component (or subsystem) models in the MIL are 'rich' in the sense that they are multi-domain and their connectivity will allow the analysis of interactions throughout the system.

An example for a Model Integration Language

In one of our research projects[1], we have built a model-integration language to support the design of complex cyber-physical systems (CPS). CPS are defined as engineered systems that integrate physical and cyber components where relevant functions are realized through the interactions between the physical and cyber parts. Examples include highly automated vehicles, smart energy distribution systems, automated manufacturing systems, intelligent medical devices, etc. The design of such systems involves the design of the physical, the cyber (computational and communicational), and the cyber-physical components of the system and their integration. There are a number of complex engineering tools that solve parts of the problem, e.g. CAD tools for mechanical design, Finite Element Analysis (FEA) tools for determining stresses on structural elements, simulation tools for the analyzing the dynamics of the system, modeling and synthesis tools for the design and implementation of the (cyber) hardware and software, thermal analysis tools for verifying thermal behavior of the system, and so on; but they are used in isolation, by domain engineers. Purely understood cross-domain interactions lead to expensive design iterations.

We have designed and implemented a MIL called 'Cyber-Physical Modeling Language' (CyPhyML) [1] that allows the representation of cyber-physical components and the design CPS through composition. The language is primarily structural (i.e. composition-oriented), but components (and subsystems) have rich, typed interfaces, with four categories: parametric and property interfaces (for parametrization and configuration), signal interfaces (for cyber interactions),

[1] DARPA Adaptive Vehicle Make Program, META Design Tools and Languages project at Vanderbilt University

power interfaces (for physical interactions representing the dynamics), and structural interfaces (for geometric alignment of the physical components). In CyPhyML one can represent the design of an entire CPS, but the native models of the components and subsystem are stored in the domain-specific tools - CyPhyML merely describes how they are composed. Cross-domain analysis is supported by model interpreters that assemble complex model analysis campaigns from the CyPhyML models, possibly involving multiple analysis tools.

An example for Interaction Modeling

In another research project[2] we worked on the problem of semantic integration of models representing reactive controllers. The motivating example came from a system of systems: a spacecraft and a launch vehicle, where both systems have a reactive controller that interacts with its counterpart in the other system. Two major issues were posed: (1) each reactive controller was modeled in a different variant of the Statechart notation (Stateflow and UML State machines, specifically), and (2) the controllers were exchanging messages that influenced their behavior. The goal was to verify the concrete, integrated system (where the controller models and the message exchanges were given) through model checking.

The first problem was solved by developing a framework, called Polyglot[2], to specify the semantics of Statechart variants in an executable form. The framework is built as a set of Java classes that can be specialized according to the semantic variant yielding an 'interpreter' that receives events and produces reactions to events, but whose behavior is determined by the specific model, acting as the 'program' being interpreted. The 'model' is stored as a data structure in the interpreter. We have verified the correctness of the model interpreter(s) using numerous tests exercising the various features of the modeling language. The model interpreter produced the same output sequences from the same input sequences as the code generated by the Stateflow and Rational Rose code generators, respectively. Note that the interpreter (as well as the generated code) is purely sequential: it is executed upon the arrival of input events and produces output events upon each invocation.

The second problem was addressed by providing a framework, implemented again in Java, for representing (1) how a reactive controller is wrapped into a looping process that uses some (blocking or non-blocking) 'receive' and 'send' operations to interact with its environment, and (2) how two (or more) reactive processes interact with each other via some message exchange protocol. Note that the processes are concurrent, i.e. arbitrary interleaving of the process executions is possible, hence the system has a built-in non-determinism. The main idea here was to model the interactions (thus the integration) via modeling the 'glue': the scheduling of processes and the interaction protocols. In other words, we have created a (potentially non-deterministic) scheduler that modeled the behavior of

[2] Model Transformations and Verification project, supported by NASA ARC.

the composed system. For the analysis of the composed system we have relied on the Java Path Finder tool (from NASA) that allows the byte-code based verification of Java programs, permitting non-deterministic behavior.

Lessons Learned

The main lesson we have learned was that one should focus on problem-driven integration of models, and not on some grand unification. Models are built for a purpose and when a larger system needs to be analyzed, synthesized, implemented, verified, tested, operated, or maintained one has to be very pragmatic and concentrate on what the models are for, and consider integration accordingly. Hence, one should pay attention to how effectively such model integration can be supported by tools.

Acknowledgments

The concepts, ideas, and work described in this paper are the work of many people, including Janos Sztipanovits, Daniel Balasubramanian, Ted Bapty, Abhishek Dubey, Ethan Jackson, Xenofon Koutsoukos, Zsolt Lattmann, Tihamer Levendovszky, Nag Mahadevan, Adam Nagel, Sandeep Neema, Joseph Porter, Gabor Simko and many of the colleagues at the Institute for Software-Integrated Systems. The support of Dr Michael Lowry of NASA Ames and Dr Corina Pasarenau of CMU with whom we collaborated on the NASA project is also recognized. The specific work has been supported by the DARPA AVM program (HR0011-12-C-0008), by the NSF (CNS-1035655) and NASA (NNX09AV58A). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not reflect the views of DARPA, NSF, or NASA.

References

1. Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E.: Openmeta: A model and component-based design tool chain for cyber-physical systems. In: From Programs to Systems – The Systems Perspective in Computing (FPS 2014), Grenoble, France, Springer, Springer (2014)
2. Balasubramanian, D., Păsăreanu, C.S., Karsai, G., Lowry, M.R.: Polyglot: systematic analysis for multiple statechart formalisms. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2013) 523–529