Software and Systems Modeling (2020) 19:527–528
https://doi.org/10.1007/s10270-020-00793-7

EDITORIAL

Modeling dynamic structures

Jeff Gray[1] · **Bernhard Rumpe**[2]

Published online: 4 April 2020

David Harel once said during a talk, "Bridges are made to stand and software is there to do." This is a very appropriate analogy, because it shows that software is about behavior. As a consequence, many software modeling techniques supported by languages also allow the description of behavior. However, behavior usually is embedded in some structure. In object-oriented systems, this is typically the object, where a system is composed of many object instances, with the class as the describing artifact that defines the blueprint. In many forms of complex or distributed systems, the notion of "component" or "assembly" is also used in various forms to describe structure.

At the programming level, there are usually only two forms of structural definitions. On one hand, we have a fully dynamic version, such as classes and the possibility to instantiate and link these classes in the form of objects. Furthermore, an object can be "rewired" dynamically with other objects building dynamic structures at run time. On the other hand, there is also a completely static definition, where all structural and connection aspects of components are defined at design time. This static structure is then replicated exactly as is and never changed at run time. For example, this has been the case in automotive, avionics and other safety critical systems. The strict static structure has advantages, because it allows us to better assess and analyze the behavior attached to the structure. It also has disadvantages, because it hampers reuse, dynamic updates, configurability, and other advantages that software engineers desire.

Due to static versus dynamic structure of the system, we also (should) use very different modeling languages for those structures. For example, UML class diagrams model principal structures that allow various forms of dynamic instantiation. Correspondingly, UML object diagrams allow the description of specific "instantiated" situations. On the other side, there are, e.g., SysML internal block diagrams (IBDs) that describe static structures (with some extensions). It can also be observed that often modeling languages for static structures do not have an elaborated type system, because each component exists only once. A distinction between class (as type and blueprint) and instantiable object is not possible. (Ok, SysML IBD do borrow a type system from the UML class diagrams.)

Software architects might need something in between: they need to be able to model a controlled form of dynamics that allows them to understand the system, but also does not bind them to a specific static structure. This is definitely a challenge when designing a good modeling language for architectures that captures both. The need for a balance between the relationship of dynamic adaptation and fixed static structures in software and systems design leads us to the following interesting questions:

(1) What are the allowed forms of changes? Particularly: is the set of possible "configurations" finite or infinite?
(2) Can new kinds of components be added to an already existing system?
(3) When can a change happen?
    (a) while the system is running (e.g., dynamic class loaders)
    (b) in stand-by mode (e.g., software update in some modern electric cars)
    (c) when configuring a system (e.g., individual for each car during assembly)
    (d) when designing a product (e.g., developer choosing from a product line)

The latter two are not very "dynamic," but interesting quality and modeling aspects surprisingly often need to be handled the same way.

(4) Who is deciding the change?
    (a) the component internally

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

Jeff Gray
jeff.gray@sosym.org

[1] University of Alabama, Tuscaloosa, AL, USA

[2] RWTH Aachen University, Aachen, Germany

(b) user or developer
(c) the adverse environment
(d) a supervising, orchestrating component

There are several techniques that offer benefit when describing controlled dynamics, but to our knowledge, none covers a broad range of the above questions. The existing techniques are more focused at the programming and calculus level, rather than focused within the context of modeling languages. The current mechanisms to support controlled dynamics can be classified into the following forms:

(1) Change of structure is defined by explicit, imperative operators. Typically, the structure starts from a minimal configuration (e.g., empty object-system, core feature set) with explicit mechanisms to instantiate and connect structural elements; capabilities to reconnect or delete/kill structural elements are also provided. Often, these operators are rather basic mechanisms ("assembler like"), but they can also be aggregated into larger and more meaningful forms that are consistency-preserving, such as semantically valid transformations or delta-operations.
(2) The other alternative is mainly to use a general constraint-like diagram, such as UML class diagrams that describe the set of possible structures, using cardinalities and other structural constraints. Class diagrams define type-like elements (e.g., classes, associations) that can be instantiated dynamically. They provide flexible description techniques, but a rather limited power of control. Of course, we can add OCL as a more fine-grained logic language that allows the set of possible configurations in class diagrams to be constrained.

As noted, behavioral descriptions are usually attached to the existing structure: the structural description is the "master." However, it is also possible to use the behavioral description, especially state machines, to describe the dynamic structure in a controlled way: the behavior description becomes the master. If so, this is done in two ways: (1) in descriptive form by attaching a specific configuration to each state, or (2) in a more imperative form by attaching a structure transformation to each transition. We observe as an aside: The use of behavioral modeling languages in combination with structural modeling languages makes it evident that such a language combination must be dependent on the form of use and not(!) a fixed combination. Most tools do not cover these needs.

It seems that the complexity of the systems in our world could be described and modeled better if we had a flexible combination of modeling techniques to (1) describe a static core structure, (2) allowing the instantiation of additional structural elements in a controlled way, and (3) also defining, respectively, constraining the possible sets of structures using behavioral or logic languages. We definitely need a deeper understanding of the techniques and tools that could better assist software and systems engineers in describing dynamically changing structures. This could lead to additional opportunities and benefits that can be realized from either new combinations of modeling languages or appropriately defined new DSLs. It will be beneficial to see more investigation into these techniques in the future.

Content of this issue

1. **BPMDS 2017 Special Section**
   Guest Editors: Selmin Nurcan and Rainer Schmidt
2. **Theme Section on Model-Based Engineering of Smart Systems**
   Guest Editors: John Fitzgerald, Fuyuki Ishikawa, and Peter Gorm Larsen
3. **Regular Papers**
   - "Benchmarking bidirectional transformations: theory, implementation, application, and assessment" by Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zuendorf
   - "Systematic review of matching techniques used in model-driven methodologies" by Ferenc Somogyi and Mark Asztalos
   - "Transitive-closure-based model checking (TCMC) in Alloy" by Sabria Farheen, Nancy Day, Amirhossein Vakili, and Ali Abbassi
   - "Ark: a constraint-based method for architectural synthesis of smart systems" by Milena Guessi, Flavio Oquendo, and Elisa Yumi Nakagawa
   - "Using empirical studies to mitigate symbol overload in iStar extensions" by Enyo Gonçalves, Camilo Almendra, Miguel Goulão, Joao Araujo, and Jaelson Castro

We hope that you develop new insights and observations when reading the articles in this issue.