Software and Systems Modeling (2025) 24:975–976
https://doi.org/10.1007/s10270-025-01301-5

EDITORIAL

On theory and management of dependencies between models

Marsha Chechik[1] · Benoit Combemale[2] · Jeff Gray[3] · Bernhard Rumpe[4]

Published online: 20 June 2025

Software developers often need to manage dependencies. Unfortunately, software dependencies manifest themselves in various forms, and discussions about dependencies can be challenging due to the very different definitions and relationships that developers may have in mind. To reduce misunderstandings, it may be helpful to categorize the various forms of dependencies.

A dependency is a relationship between two (or more) different things. Let us exclude relationships with more than two participants and concentrate on binary relations to simplify considerations. During a typical development process, dependencies may emerge across all forms of artifacts, including requirement statements, explicit models, source code, and (readily compiled and deployable) system elements. To be precise, we distinguish development artifacts (which include, e.g., UML/SysML models and source code) and the system elements. For example, an object-oriented system consists of implemented classes and their (logical) aggregations in the form of subsystems and components. System elements are to be distinguished from models and source code, which aggregate into packages, directories, branches, or even (version-controlled) projects. Both sides, i.e., the system and the artifacts describing it, are not entirely independent of each other. Java, in particular, has done a tremendous job reliably connecting classes and their source files in an almost one-to-one relation. Colloquially, we thus do not need to distinguish between a class and its describing source file anymore. However, in this article we mention these two sides because the term "dependency" is used within both sides. Projects depend on each other; components depend on each other; and models depend on each other.

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

[1] University of Toronto, Toronto, ON, Canada

[2] University of Rennes, Rennes, France

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

[4] RWTH Aachen University, Aachen, Germany

Let us simplify considerations by concentrating on development artifacts while not considering system elements and their dependencies here.

A dependency is typically asymmetric, i.e., "A depends on B" does not imply the inverse. Circular dependencies cannot happen for certain forms of dependencies, e.g., when an artifact A is generated using an artifact B. Lack of circularity creates a dependency hierarchy (a tree or a DAG). Due to the very different forms of dependencies, we should also not mix those up: "A depends on B" and "B depends on C" do not necessarily imply any relationship between A and C. "Dependency" in general is therefore not transitive (while certain forms of dependencies usually are).

To further understand the possible forms of dependencies, it is helpful to look at a project using a typical programming language (e.g., Java), where some parts are generated from a high-level modeling language (e.g., UML Statecharts). In this scenario, development artifacts are Java source files at one level and Statechart models at another. After long and extensive discussions and examinations of projects, it is clear that we need only to distinguish between two main different and very general kinds of artifact dependencies: (a) "is derived from" and (b) "uses."

Typically, "B is derived from A" means that there is a manual activity or an automatic generator that takes all the information from artifact "B" and produces artifact "A." For example, a Java class "MyStatechart.java" is generated from "MyStatechart.model." Other examples: "C.class" is generated from "C.java" or "System.jar" is an aggregation of "C.class" files. Typically, a source "A" is no longer used in the ongoing process (unless modified and used for regeneration) because "B" encodes the information from "A," potentially in a different form. When ignoring reverse and roundtrip activities and all their peculiarities, we can see that such a dependency relationship is acyclic.

The second main form of relationship, "uses," is of a very different nature. In such a relationship, an artifact, for example, a Java class "A.java," imports "B.java" to use certain elements that have been defined in B or in its superclasses. "A.java" does not contain information encoded in "B.java,"

but relies on its elements, i.e., methods, types, constants, or externally visible attributes. In Java, such a dependency can be found through an "import" statement (or using a fully qualified name). Of course, this form of a relationship may be cyclic (even though that may from an engineering point of view result in a suboptimal design).

If more precision is desired, the encapsulated details of an artifact may be analyzed to differentiate between various kinds of uses (e.g., a method call, an attribute read or write access, a use of the imported type as an attribute type, as a method parameter type or as a local variable type, an object instantiation, or use of a constant). A well-defined Statechart model needs similar import facilities to describe the internal state, possible incoming messages, preconditions, and transition effects. Thus, Statechart models also have "uses" relationships with other artifacts. This includes further forms of "uses" between various models, for example, because a Statechart may embed another Statechart as a sub-state or interact via parallel composition.

A large modeling language, such as SysML, and in particular, SysML v2 (which is under final approvement in 2025), needs to provide quite a number of different relations between modeling elements, such as usage, composition, reference, specialization, subset definition, or general redefinition. Many of those can be applied to different kinds of modeling elements. When all of those relationships are aggregated between different models, the result is a large variety of "use" dependencies between the models. We are currently not aware of any work that has precisely addressed all of these different dependencies, neither for UML, nor for SysML, in full detail.

However, in some forms of languages/projects other forms of relationships are denoted as dependencies, such as "copy of," which might be used to describe that two identical copies of the same file are stored redundantly in different places (for whatever reason), or "tested by." Dependencies like "satisfied by" and "refined by" typically connect models and requirements and share characteristics with "derived by," but have different methodological impact.

Maybe it is time for the modeling community to have a deeper look at the dependencies or, more generally, relationships between models, potentially coming up with a "theory of dependencies," which finally might lead to more efficient and better-understood management of dependencies in supporting development tools, visualizing current dependencies, highlighting evolutionary changes in the dependency graph, or monitoring these dependencies according to a given "artifact aggregation structure" to prevent forbidden relationships.

Funding Open Access funding enabled and organized by Projekt DEAL.