Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

TOWARDS A GENERAL FOUNDATION FOR FORMALISM-SPECIFIC INSTRUMENTATION LANGUAGES

Johannes Schützel
Roland Ewald
Adelinde M. Uhrmacher

Albert Einstein Str. 22
University of Rostock
18059 Rostock, GERMANY

ABSTRACT

Experimenters need to configure the data collection performed during a simulation run, as this avoids overly large output data sets and the overhead of collecting them. Instrumentation languages address this problem by allowing experimenters to specify the data of interest. Such languages typically focus on a specific modeling formalism. While this allows for a more expressive syntax, it also prohibits their application to other formalisms. To resolve this trade-off, we propose a formalism-independent model of the instrumentation semantics and use it as a basis for developing embedded domain-specific languages (DSLs). Our instrumentation DSLs share common code, allow to add formalism-specific syntax, and are easy to extend.

1 A GENERAL FOUNDATION

At first, we sketch the 'semantic model' (Fowler 2010) that represents a user's intentions regarding instrumentation and which has been implemented as a Java class hierarchy that also could be transformed to an XML schema. It is independent of any modeling formalism and serves as a basis for our formalism-specific instrumentation languages. The model allows to specify a set of arbitrarily many *instrumentation queries*, each of which needs to specify 1) when to observe, 2) what entities to observe, 3) which values to collect from the observed entities, and 4) how to aggregate the collected values.

While some of these facets can be specified in a formalism-independent manner, others require additional knowledge on the modeling formalism at hand. For example, defining what entities to observe requires adequate means to address entities in the structure of the model. To address this issue, our semantic model supports three common model topologies: grids (e.g., for cellular automata), trees (e.g., for DEVS models), and graphs (e.g., for Petri nets). This allows to define instrumentation queries that refer, for example, to the neighborhood of an entity (grid), its parents (tree), or its neighbors (graph). For representing a query, the semantic model prescribes the following elements (brackets refer to optional elements):

```
Query = Activation + MultiActiv + [ FurtherTriggering + [Deactivation] ]
        + Filtering + Extraction + [ Partitioning + Aggregation ]
```

`Activation`, `FurtherTriggering`, and `Deactivation` define when to start the observation, with which frequency to continue it, and when to stop it. Each element can refer to a number of simulation steps, a simulation time, a wall clock time, or a condition defined on the current state of the simulation model. The latter can be useful for more complex instrumentation tasks, e.g., to first detect a model state of interest and then to start collecting more data. `MultiActiv` is a flag that defines whether the observation is re-started (after deactivation), which may be desirable for condition-based `Activation`. `Filtering` is responsible for selecting the model entities of interest. It works like a predicate defined on model entities, i.e., it defines whether an entity is considered for observation. In the `Filtering` element, multiple filters can be combined by logical operators, where filters may refer to attributes of the given entity or to attributes of other entities in its context. Which other entities are accessible depends on the model topology. There are dedicated context-based filters for grid-, tree-, and graph-based structures, e.g., to access a parent entity in a tree. `Extraction` specifies what data to collect from an entity that passes the filtering. `Partitioning` and `Aggregation` define how to aggregate collected values. Per default, every value is recorded separately. `Partitioning` can be done either by key, e.g., to group entities with the same name, or by value, resulting in a histogram if `Aggregation` is specified as counting. `Aggregation` represents a function that accepts a sequence of values and aggregates them.

2 EMBEDDED FORMALISM-SPECIFIC INSTRUMENTATION LANGUAGES

We implemented the semantic model in Java, as part of the modeling and simulation framework JAMES II (Himmelspach and Uhrmacher 2007). Now, we can develop formalism-specific embedded DSLs; queries can be mapped to the semantic model and executed by generalized instrumentation mechanisms (e.g., for trees: Helms et al. 2012). Our prototype DSLs are implemented in the Java-compatible programming language Scala (e.g., Odersky et al. 2011) and complement SESSL (Ewald and Uhrmacher 2012), a Scala DSL for simulation experiments. The DSLs share most code for query construction and all generic syntax elements. Supporting a new formalism requires only a few lines of code: the developer merely needs to add formalism-specific syntax elements and map them to (parts of) the query model. Our syntax is inspired by SQL. The general scheme is to separately define when to observe (`Activation`, `FurtherTriggering`, `Deactivation`), as this may be rather complex. Then, a specification is given that first defines what data to collect (`Extraction`, `Aggregation`) and—after the `where` keyword—from which model entities data is collected at all (`Filtering`). Figure 1 shows sample queries for cellular automata (CA) and ML-Rules, which is a rule-based language for multi-level modeling (Maus et al. 2011).

```
1 //1D Cellular automaton: count pattern after a warm-up phase
2 observe(start at (steps = 100) sample every (steps = 10)) {
3   count where pattern(1, 0, 1, 0, 0)
4 }
5 //ML-Rules: observe first attribute of certain proteins inside a vesicle
6 observe(sample every (seconds = 10)) {
7   attrib(1) where (name("Protein") and attrib(2) < 10 and parent(name("Vesicle")))
8 }
```

Figure 1: The DSL for CA instrumentation provides additional support for pattern detection (`pattern`, line 3). The DSL for ML-Rules instrumentation takes into account that species are named (`name`, line 7), can be nested (`parent`, line 7), and their attributes are accessed by indices (`attrib`, line 7).

REFERENCES

Ewald, Roland and Uhrmacher, Adelinde M. 2012. “Setting up Simulation Experiments with SESSL”. Poster, Winter Simulation Conference 2012.

Fowler, M. 2010. *Domain-Specific Languages*. 1st ed. Addison-Wesley Professional.

Helms, T., J. Himmelspach, C. Maus, O. Röwer, J. Schützel, and A. M. Uhrmacher. 2012. “Toward a language for the flexible observation of simulations”. In *Proc. of the 2012 Winter Simulation Conference*.

Himmelspach, J., and A. M. Uhrmacher. 2007. “Plug’n simulate”. In *ANSS ’07: Proceedings of the 40th Annual Simulation Symposium*, 137–143.

Maus, C., S. Rybacki, and A. M. Uhrmacher. 2011. “Rule-based multi-level modeling of cell biological systems”. *BMC Systems Biology* 5 (1): 166+.

Odersky, M., L. Spoon, and B. Venners. 2011. *Programming in Scala*. 2nd ed. Artima.