Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

TOWARD A LANGUAGE FOR THE FLEXIBLE OBSERVATION OF SIMULATIONS

Tobias Helms
Jan Himmelspach
Carsten Maus
Oliver Röwer
Johannes Schützel
Adelinde M. Uhrmacher

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

ABSTRACT

Simulation studies typically imply the generation and interpretation of data. Collecting, storing, and filtering data can be expensive. Therefore, it is important to allow a user to specify these processes flexibly depending on the modeling language, the model, and the objective of the simulation study. An instrumentation language is presented and applied to collect, aggregate, store, and filter data generated during experimentation with models specified in ML-Rules, a rule-based multilevel modeling language for cell biological systems.

1 INTRODUCTION

A simulation study can induce the need to execute hundreds of model configurations, with the need to execute thousands of replications for each configuration. To explicitly select the data to be stored can reduce the storage consumption and can speed up the process of storing and analysis. This is of even more importance if aggregations of the values are already required during the computation of replications, e.g., for deciding whether a computation can be stopped because a certain state has been reached.

A common view on runtime efficiency is the efficiency of different computation algorithms/protocols, e.g., Wang et al. (2009), or of data structures, such as differences in event lists (Vaucher and Duval 1975). This view is an important first step but for real life applications data collection may add a significant penalty to the execution time. Data collection is mandatory and an efficient realization can help to reduce the overall time needed for computation and the capacity of the storage devices needed. In existing software for modeling and simulation (M&S) data collection is either turned on by default, e.g., in SDML (Moss et al. 1998), or it can be specified in the model, e.g., in BioNetGen (Blinov et al. 2004). Some software packages provide visual means for selecting properties to be observed, but depending on the model this procedure is often not feasible as a model can comprise thousands of entities. In addition, in the case of models with a dynamic structure, e.g., Uhrmacher (2001), new model entities can be generated and should be observed as well. To open an editor to let a user select what to observe would hamper any experiment execution, especially if thousands of model entities appear and vanish in the course of a single computation. Thus a convenient specification of advanced filtering and aggregation functions is needed, and the following questions arise:

1. How could a general language for observation look like, if it is possible to create such a language?
2. Are there specific elements induced by the modeling formalism which play a role in observation?

978-1-4673-4781-5/12/$31.00 ©2012 IEEE

These questions are examined using JAMES II and the rule-based multilevel modeling language ML-Rules (Maus et al. 2011). This language has been created for models in the realm of systems biology and allows a rule-based definition of models which can be nested and exhibit dynamic structures. JAMES II is an open source M&S framework, based on the plug'n simulate architecture (Himmelspach and Uhrmacher 2007), not restricted to any modeling means. It has been developed based on a strict separation of concerns and already provides a basic mechanism for instrumentation (selecting what to observe) and observation (determine what to do with the observed values). However, up to now, to use these mechanisms implies using Java.

Another question is whether data collection belongs to the model or to the experiment the model is used in. What can be observed depends on the model to be observed, what we are interested in depends on the experiment. Observation is intrinsically bound to the model (as model attribute names have to be used), but for a model more than one observation definition might exist. This is due to the fact that a model might be reused in different experiments. However, an observation definition might be reused in different experiments as well and thus we conclude that the observation definition should be separated from model and experiment.

In the next section we give an overview of related work on observing data in scientific computing. Section 3 describes the requirements analysis done based on a questionnaire, existing solutions, and personal experiences of the development team. In Section 4 a general language for instrumentation is presented. Section 5 demonstrates the usage of the language on experiments with models described in the ML-Rules language. The subsequent sections sketch the realization which executes instructions specified in the instrumentation language and illustrate some performance insights. We conclude with a discussion of the achieved solution.

2 RELATED WORK

Solutions to the problem of selecting data to be observed are available for many M&S software products, but to our best knowledge there is not much scientific work dealing with specification or efficiency considerations of data collection.

A subset of the software packages for M&S provides visual instrumentation/observation means. For example, Snoopy (Rohr et al. 2010) and COPASI (Hoops et al. 2006) allow a user to select in a check list box the variables to be plotted. The latter also supports to create individual and reusable report definitions by choosing diverse kinds of model as well as simulator objects to be observed and included in the report. Such graphical approaches may not scale well in case of models with thousands of entities. Other packages, e.g., BioNetGen (Blinov et al. 2004), allow to describe what to observe using BioNetGen expressions in the model code. Thereby, model entities may be filtered and aggregated according to certain attributes. So far the observation definitions are placed in the model file and they are restricted to BioNetGen models. A comparable approach is made by MatLab/SimuLink which provides special export elements which can be placed into the model description and thus observation gets also part of the model description. DEUS (Amoretti et al. 2009) integrates observation functionality by extending the model as well: logging events are scheduled as "normal events" but if they are executed they do not modify the model but record the model's state.

Some software packages allow to setup observation via coding. Dalle and Mrabet (2007) presented OSIF, an approach where via aspects, observation code is weaved into the models implementation at the places one wants to observe data from. They argue that model and observation code should be separated to keep the model readable and to increase the performance if some observations are not required in a model use. Although of interest this approach has only a restricted general usage as it assumes that models are coded in a programming language for which an aspect-oriented compiler is available. But if this holds true it is the most efficient variant as the code for observation is only embedded if it is really required. The authors do not define an extra language to specify which aspects shall be weaved into the code. This has to be done with the normal means of the language used.

In JAMES II instrumentation and observation is based on the plug-in schema and the observer pattern (Dalle, Ribault, and Himmelspach 2009). What shall be observed is defined by specifying (coding) an instrumenter which takes care of attaching observers to the observable entities a user is interested in. This is comparable to the approach of Dalle and Mrabet (2007). In contrast to the aforementioned approach the observer pattern adds a performance penalty induced by the observation code: calling an update method if an observable entity has changed costs some time, independent from existing observers.

The last category, e.g., represented by SDML (Moss et al. 1998), automatically records every state change without any chance to select what shall be observed.

Thus, the approaches are either based on coding in a programming language (e.g., Java), are restricted to specific model types (e.g., BioNetGen), are part of the model description (e.g., BioNetGen, MatLab, DEUS) or they collect all data (e.g., SDML).

3 REQUIREMENTS — OBSERVING MODELS AND COMPUTATION ALGORITHMS

The requirements collected stem from a small questionnaire, existing solutions, and personal experience collected over several years of M&S software building and use. To get better feedback on requirements imposed by potential users a small questionnaire has been developed and different users (biologists, simulationists, visualization experts, computation algorithm/environment developer, and modeling means developer) have been asked what they need. In summary the participants of this small survey found it important to collect

- state changes from specified model entities,
- data from computation algorithm entities,
- counts of model entities,
- reactions/events/state transitions executed,
- structural changes in models,
- simple statistics (like averages, etc.) from specified model/computation algorithm entities, and
- all changes, changes every n steps, every n time units, or when a specific expression becomes true.

These observations are needed to store or visualize trajectories of models, to observe data to steer experiments (e.g., optimizations), to compute stopping criteria (e.g., compute until a certain state has been reached), to control the number of replications to be computed, and to get data for an automated selection of computation algorithms (e.g., to speed up the computation of the upcoming replications). Some parts will be explicitly specified for an experiment (e.g., data to be collected due to the objective of the simulation study) whereby another part might be implicitly specified by needs arising from using certain stopping or replication criteria or by the optimization algorithm used. This implies that we need to allow multiple instrumentations — those explicitly defined by users and means arising out of the experiment.

Further requirements stem from the environment the solution is developed for: JAMES II is not restricted to any modeling language/formalism, and thus the instrumentation/observation should be usable for any of those; JAMES II aims at an efficient execution of models at "any size", and thus the instrumentation/observation should strip away unwanted data as early as possible, and data observation should not be too expensive; JAMES II may be equipped with modeling means which allow to describe systems which change their structure over runtime, where model entities might be added or removed; and thus the instrumentation has to be able to observe changes in new model entities as well.

To maximize re-usability of models and observations the separation of model, experiment, and observation specification is recommendable.

4 AN INSTRUMENTATION LANGUAGE

To reflect the identified requirements above, an instrumentation language is designed which is inspired by the structured query language (SQL). The grammar of the language is given in Figure 1.

The `INSTRUMENT` keyword allows to select what to instrument (either model or simulator) with an observer. With the `OBSERVE` keyword a user can define *w*hat he would like to observe. Observation can be limited by conditions which have to be fulfilled before a new observation is made and by the frequency the observations shall be made, i.e., only observe every n-th step or after n simulation time units.

In the specification (cf. Figure 1) the parts that are modeling means dependent constructs are put into angle brackets. The language constructs depending on the specific means are the `what` statement, which addresses the requirement to provide means for aggregation and statistical measures, the condition which allows us to introduce advanced filtering mechanisms based on expressions, and the language specific terms which generate keys to be used to identify group memberships.

Thus, most of the language is independent of the concrete modeling means the model to be observed is realized in. However, as a prototype the language has been realized so far only for ML-Rules. In the next section the ML-Rules specific replacements of the three constructs are introduced.

```
File             ::= Header Queries
Header           ::= "VERSION" number
Queries          ::= { Query ";" }
Query            ::= "INSTRUMENT" Target ObserveStatement WhereStatement GroupByStatement Trigger
Target           ::= "MODEL" | "SIMULATOR"
ObserveStatement ::= "OBSERVE" What ["," What]
WhereStatement   ::= "WHERE" (WhereCondition | "TRUE")
GroupByStatement ::= "GROUP BY" <language specific term>
Trigger          ::= "EVERY" When
What             ::= ( "COUNT" | (("SUM" | "AVG" | "MIN" | "MAX") "[" NumberType "]") )
                     "(" <reference.to.target> ")"
NumberType       ::= "DOUBLE" | "INTEGER"
WhereCondition   ::= <reference.to.target> "=" Value | "IN" "{" Value {"," Value} "}"
                     {"AND" WhereCondition}
Value            ::= Constant | <reference.to.target>
Constant         ::= String | Double | Integer
When             ::= (Integer "STEPS") | (Float "T")
```

Figure 1: Grammar of the instrumentation language.

Looking at our requirements we see that we capture with this language the collection of state changes from specified model entities, data from computation algorithm entities, counts of model entities, reactions/events executed, simple statistics (like averages, etc.) from specified model/computation algorithm entities, and to record all changes, changes every n steps, or after n time units. Further on, the modeling means specific parts have been identified and thus the language copes with the requirement of JAMES II that it should be usable for different means. The set of user supplied queries can be extended by any number of additional (generated) queries, i.e., queries generated by the experiment to get the data needed for optimizations etc.

A model/simulator might define what can be observed to ease the instrumentation process. An editor created for this language might then make use of this knowledge to support users in defining instrumentation queries. Checking whether the instrumentation created in the language can be applied to the real objects to be observed is done while the instrumentation is performed, again editors might provide advanced means for checking this in before.

5 INSTRUMENTING ML-RULES

ML-Rules is a recently developed modeling language for describing quantitative multilevel models in the field of systems biology (Maus et al. 2011). The language design is based on a reaction-centric paradigm and supports hierarchically nested variable model structures. Thereby, each level in the hierarchy may has its own state and behavior. The state of model entities (called species) is characterized by attributes (which may be of a string, boolean, or numerical type) and their enclosed content, i.e., a multiset of species (called solution). Dynamics of a model are specified in terms of rule schemata, which are constrained by

attributes and amounts of species. The stochastic rate with which a concrete rule instantiation will fire is exponentially distributed, i.e., the semantics of ML-Rules grounds on continuous time Markov chains. We do not want to go into more detail here and would like to refer the interested reader to the original paper by Maus et al. (2011) instead. However, the following examples should give an impression about the heterogeneity of produced data and the need for flexible instrumentation and observation of ML-Rules models.

5.1 Adapting the Instrumentation Language to ML-Rules

Adapting the generalized instrumentation language to ML-Rules requires to define the statements in angle brackets from Figure 1 for this particular modeling formalism. Thereby, language specific constructs like attributed species, nested model structures, and rule schemata need to be considered.

Different statements require the description of a language specific `<reference.to.target>` to access the desired information. For example, `species.name` references the name of a species and `species.quantity` retrieves the species amount. Other targets denote specific attributes of a species, which have a defined order in ML-Rules and can thus simply be referenced by `species.attribute(n)`, where n is an integer number. An example of a simulator target is `matchings.propensity`, which retrieves the propensity of a rule schema matching.

For the identification of data in the produced output and for data aggregation, the `<language specific term>` being part of a `GroupByStatement` defines a language specific method for generating defined keys. In the ML-Rules realization this could be, e.g., `species.name`, but also `species.name attribute aware` and `species.upwardhierarchy` which take attributes of species or their localities within the hierarchical model structure into account when generating keys.

5.2 Example 1

This first example is a very simple one with the only purpose to give a brief introduction to some basic principles and syntactical notations of ML-Rules and to provide first simple queries for instrumenting such models. The example model comprises two species ("A" and "B") and two rules, which describe how the amounts of species change while the simulation executes:

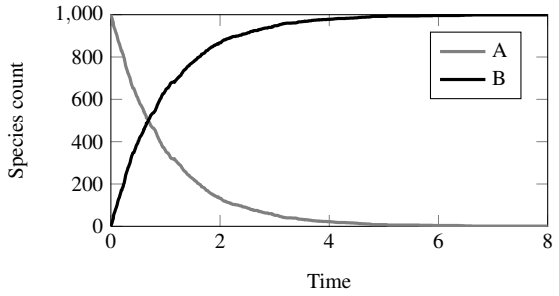
$$A^a \xrightarrow{\#a} B(1)$$

$$B(x)^b \xrightarrow[x<4]{0.3\#b} B(x+1)$$

The first rule describes a conversion from species "A" to species "B". The latter is characterized by an attribute "1", which is written between round brackets behind the species name. This attribute will be incremented when the other rule fires. This second rule is specified in terms of a rule schema, i.e., depending on the matched species different concrete rules may be instantiated. However, the rule is constrained to only fire as long as the attribute of "B" is smaller than 4. The stochastic firing rate of a rule depends on its transition propensity. In the example above, the propensities of both rules are adjusted during runtime as they are proportional to the amount of reactants, i.e., "A" and "$B(x)$", which may change over time and can be accessed by using the "#" operator in combination with an identifier a reactant species is assigned with (superscripts "a" and "b").

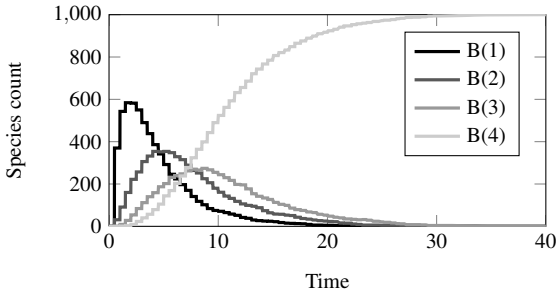
Two different instrumentation statements are given in Figure 2. The results of the instrumentation and observation (trajectories) have been used to generate charts. In both cases, the model is instrumented to count the quantity of species (`OBSERVE COUNT(species.quantity)`). However, while the left instrumentation example observes *each* species of the model (`WHERE TRUE`), the other one filters the data to observe only species "B" (`WHERE species.name = 'B'`). The `GROUP BY` statement is also different in both instrumentations: in the left example observation results are simply grouped by species names and

```
INSTRUMENT MODEL
  OBSERVE COUNT(species.quantity)
  WHERE TRUE
  GROUP BY species.name
  EVERY 1 STEPS;
```



(a) Aggregated species counts; high resolution.

```
INSTRUMENT MODEL
  OBSERVE COUNT(species.quantity)
  WHERE species.name = 'B'
  GROUP BY species.name attribute aware
  EVERY 0.5 T;
```



(b) Counts of differently attributed B species; low resolution.

Figure 2: Different queries and results from instrumentation Example 1.

in the right example keys are generated according to the attributes of observed species, i.e., a distinction between different attributes of "*B*" is achieved. Finally, the last statement in both instrumentations refers to the trigger how often the data are observed, i.e., at each simulation step (`EVERY 1 STEPS`) and every 0.5 time units (`EVERY 0.5 T`). As can be seen in Figure 2, the latter may reduce the amount of data recorded.

5.3 Example 2

In the previous section the general usage of a concrete realization of our instrumentation language is shown with the help of a rather simple model. Now we would like to give a more comprehensive and realistic example of a biological model that motivates the need for flexible instrumentation of ML-Rules.

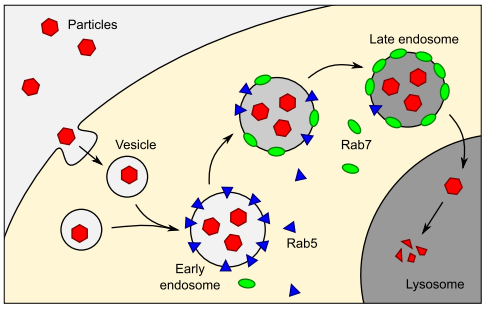


Figure 3: Illustration of the endocytosis and endosome maturation model.

The model illustrated in Figure 3 describes a process called endocytosis, that is the process by which living cells engulf molecules or other particles from their environment. Thereby, the particles are wrapped by vesicles which may subsequently fuse with other membrane-bound compartments to form early endosomes. An endosome represents a sorting compartment that identifies the correct destination of absorbed particles within the cell. For example, endocytosed receptor molecules may need to be recycled and therefore sorted out early. Other molecules need to be directed to the Golgi apparatus and deleterious particles like toxins and bacteria are transported to the lysosome compartment where they will be degraded. The latter transport process, where endosome maturation plays a crucial regulating role, is subject of our example model. During maturation, the pH of an endosome decreases significantly and a conversion of bound Rab proteins has been observed (Conte-Zerial et al. 2008; Poteryaev et al. 2010).

$$Part^p + Cell[C?]^c \xrightarrow{k_{endo}\#c\#p} Cell[Ves[Part] + C?] \tag{1}$$

$$2\,Ves[V?]^v \xrightarrow{\binom{\#v}{k_{fuse}}} Endo(2.0, \text{'early'}, 8.0)[2\,V?] \tag{2}$$

$$Ves[V?]^v + Endo(vol, \text{'early'}, pH)[E?]^e \xrightarrow{k_{fuse}\#e\#v} Endo(vol+1, \text{'early'}, \frac{pH \cdot vol + 8}{vol+1})[V? + E?] \tag{3}$$

$$Endo(vol_1, state, pH_1)[E_1?]^{e_1} + Endo(vol_2, state, pH_2)[E_2?]^{e_2} \xrightarrow{k_{fuse}\#e_1\#e_2} Endo(vol_1+vol_2, state, \frac{pH_1 \cdot vol_1 + pH_2 \cdot vol_2}{vol_1+vol_2})[E_1? + E_2?] \tag{4}$$

$$Endo(vol, \text{'early'}, pH)[Rab5^{r_5} + Rab7^{r_7} + E?]^e \xrightarrow[\#r_7 > 2\#r_5]{\#e} Endo(vol, \text{'late'}, pH)[Rab5 + Rab7 + E?] \tag{5}$$

$$Endo(vol, \text{'late'}, pH)[E?]^e + Lyso[L?]^l \xrightarrow{k_{fuse}\#e\#l} Lyso[E? + L?] \tag{6}$$

$$Endo(vol, state, pH)[E?]^e \xrightarrow{k_{acid}\#e} Endo(vol, state, pH - log(pH)10^{-3})[E?] \tag{7}$$

$$Endo(vol, state, pH)[E?]^e + Rab7^r \xrightarrow{k_{bind}\#e\#r} Endo(vol, state, pH)[Rab7 + E?] \tag{8}$$

$$Endo(vol, state, pH)[E?]^e + Rab5^r \xrightarrow{k_{bind}\#e\#r} Endo(vol, state, pH)[Rab5 + E?] \tag{9}$$

$$Endo(vol, state, pH)[Rab5^R + E?]^e + Rab5^r \xrightarrow[pH > T^{pH}]{k_{bind}\#e\#r\#R10^{-2}} Endo(vol, state, pH)[2\,Rab5 + E?] \tag{10}$$

$$Endo(vol, state, pH)[Rab5^r + E?]^e \xrightarrow{k_{unbind5}\#e\#r} Endo(vol, state, pH)[E?] + Rab5 \tag{11}$$

$$Endo(vol, state, pH)[Rab7^r + E?]^e \xrightarrow{k_{unbind7}\#e\#r} Endo(vol, state, pH)[E?] + Rab7 \tag{12}$$

$$Lyso[Rab5^r + L?]^l \xrightarrow{k_{recycle}\#l\#r} Lyso[L?] + Rab5 \tag{13}$$

$$Lyso[Rab7^r + L?]^l \xrightarrow{k_{recycle}\#l\#r} Lyso[L?] + Rab7 \tag{14}$$

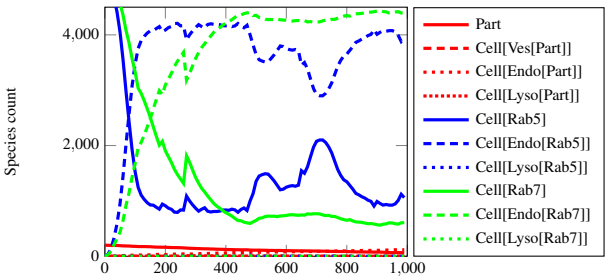
$$Lyso[Part^p + L?]^l \xrightarrow{k_{degrade}\#l\#p} Lyso[L?] \tag{15}$$

Figure 4: ML-Rules model of endocytosis and endosome maturation. An informal description is given in the text. Parameters used for simulation: $k_{endo} = 10^{-3}s^{-1}$, $k_{fuse} = 2 \times 10^{-3}s^{-1}$, $k_{acid} = 10^{1}s^{-1}$, $k_{bind} = 10^{-3}s^{-1}$, $T^{pH} = 3$, $k_{unbind7} = 10^{-3}s^{-1}$, $k_{unbind5} = 10^{-2}s^{-1}$, $k_{recycle} = 1s^{-1}$, $k_{degrade} = 10^{-2}s^{-1}$, initial solution $S = [200\,Part + 1\,Cell[1\,Lyso + 5000\,Rab5 + 5000\,Rab7]]$.

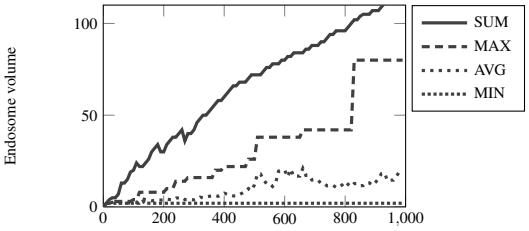
The ML-Rules model description in Figure 4 makes use of the capabilities of ML-Rules for modeling variable nested structures. For example, Rule 1 in Figure 4 describes the process of endocytosis, where a particle ("*Part*") enters the cell and thereby is wrapped by a vesicle ("*Ves*"). Nesting is described by putting the enclosed content within a pair of square brackets, i.e., "$Cell[Ves[Part]]$" describes a particle that is wrapped by a vesicle which in turn is enclosed within a cell. Similarly, Rab proteins may also change their location. The "?" operator is used to define a special variable, e.g., "$C?$" in Rule 1 or "$E?$" in Rule 5, that binds the not explicitly mentioned remaining content when performing the rule schema matching. Most of the species do not have attributes, however, endosome species are characterized by three attributes describing their volume, a discrete state ('early' or 'late'), and the pH. The values of these attributes may frequently change, e.g., due to fusion with vesicles or other endosomes. Without going more in detail, it should be obvious that the example model may produce a bunch of heterogenous data from which the extraction of relevant information may be challenging. Again, we would like to present an exemplary utilization of our instrumentation (Figure 5).

In the first example, the amounts of species filtered by a list of names are counted, similar to what has already been presented in the previous section. However, here we are interested in the location of species and thus the observations are grouped by the hierarchy above (`GROUP BY species.upwardhierarchy`).

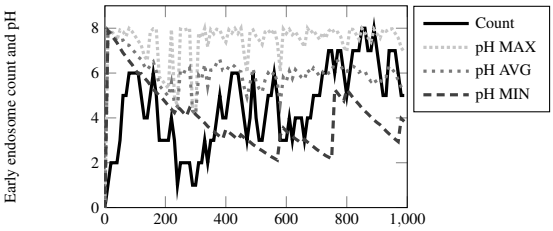
```
INSTRUMENT MODEL
  OBSERVE COUNT(species.quantity)
  WHERE species.name
        IN {'Part', 'Rab5', 'Rab7'}
  GROUP BY species.upwardhierarchy
  EVERY 10.0 T;
```




```
INSTRUMENT MODEL
  OBSERVE SUM[DOUBLE](species.attribute(1)),
          MAX[DOUBLE](species.attribute(1)),
          AVG[DOUBLE](species.attribute(1)),
          MIN[DOUBLE](species.attribute(1))
  WHERE species.name = 'Endo'
  GROUP BY species.name
  EVERY 10.0 T;
```




```
INSTRUMENT MODEL
  OBSERVE COUNT(species.quantity),
          MAX[DOUBLE](species.attribute(3)),
          AVG[DOUBLE](species.attribute(3)),
          MIN[DOUBLE](species.attribute(3))
  WHERE (species.name = 'Endo' AND
         species.attribute(2) = 'early')
  GROUP BY species.name
  EVERY 10.0 T;
```




```
INSTRUMENT SIMULATOR
  OBSERVE SUM[DOUBLE](matchings.propensity)
  WHERE TRUE
  GROUP BY matchings.name
  EVERY 10.0 T;
```

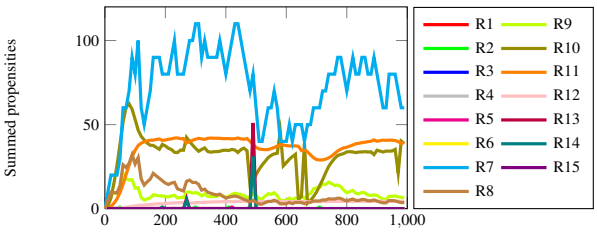


Figure 5: Different queries and results from instrumenting example 2. In all charts the x-axis describes simulation time.

The usage of statistical aggregation functions over attribute values is shown in the next two example queries. One leads to the observation of the summed, maximal, average, and minimal volume (`species.attribute(1)`) of endosome species (`WHERE species.name = 'Endo'`). The other aggregates over the third attribute (`species.attribute(3)`) of endosomes denoting their pH. In addition, the amount of species is counted again and all observations are filtered by the second attribute, i.e., only early endosomes are observed (`species.attribute(2) = 'early'`).

The last query in Figure 5 gives an example of an instrumentation of the simulator (`INSTRUMENT SIMULATOR`) rather than the model. The example shows the observation of sums of rule propensities (`matchings.propensity`), which reveals, for example, that Rule 7 (decreasing pH of endosomes) is the most active rule schema with the highest propensity among all rules most of the time.

6 IMPLEMENTATION ASPECTS

The language allows to specify any number of observation queries and thus the question arises how this can be implemented in an efficient manner.

The structure of the objects used by an observer are shown in Figure 6. An observer is attached to a model/simulator by the instrumentation procedure and the language defined above is used to derive the instances of the classes working together to filter and aggregate the data.

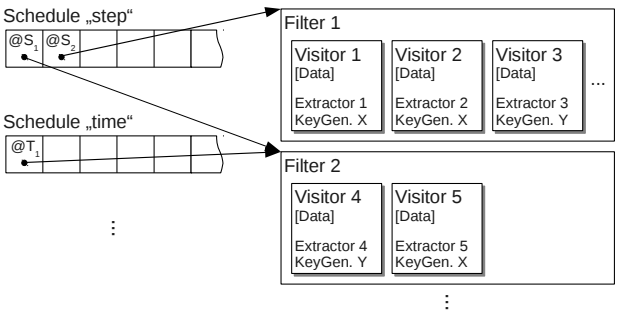


Figure 6: Schematic figure showing the relation between schedules and filters as well as the encapsulation of filters, visitors, data extractors and key generators in an observer.

The responsibility of the schedules in an observer is to control the execution of the observation. If an observation is scheduled the filters will be used to check whether the information at the current time has to be processed further. If the filters let data through the visitors will be applied to aggregate the data according to the specification. Each visitor (see Gamma et al. (1995) for a description of the visitor pattern) comprises a key generator, which effectively groups the data, and an extractor, which extracts the data to be aggregated from the observed entities. This schema allows to group queries by their triggers so that those to be executed are grouped and it makes sure that the model data structures will only be iterated once per trigger event (independent from the number of queries provided).

To use schedules, filters, and visitors to realize the software executing the instrumentations defined via the language, allows to reuse filters and visitors for different observation purposes and is the pre-condition for a flexible instrumentation language which can then easily be extended to support further aggregation techniques.

7 PERFORMANCE ASPECTS

To aggregate data over the model structure may consume a considerable amount of time. The worse it might get

- the more frequent we do this observation,
- the more aggregated data we are interested in, and
- the more complex our filter expressions are.

Although the implementation already merges the jobs to be done so that only a single pass of the model's data structures is required we expect to observe an overhead. The question we try to answer here is how expensive are the filters and the aggregation functions already implemented per execution?

For our experiments we use the second example model and we used four different sets of queries (see Figure 7). Query 1 contains a simple count statement with no filtering, Query 2 a count statement with filtering and Query 3 contains three aggregations with filtering. Query 4 is a combination of Queries 1-3. Further on we let the computation run without any observation at all to get the pure computation time. The machine configuration used is an Intel 990X, Hyperthreading activated, 3.47GHz with 24 GB of RAM (DDR3, PC3-10700), Windows 7 64Bit operated, Java 1.7 64Bit, James II 0.8.6 with an Dhrystone index (Java) of 198.19GPIS. Data is written to two 2TB discs combined using a MARVELL RAID-0 VD (ca. 209 MB/s).

Query 1:

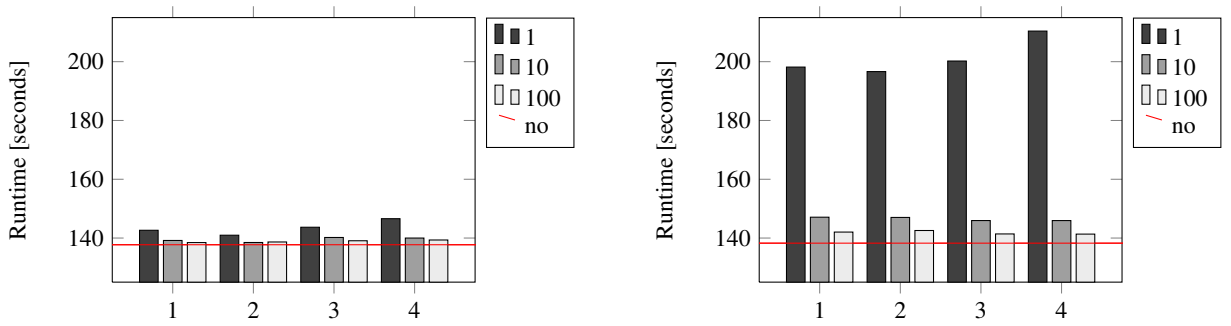
```
INSTRUMENT model OBSERVE COUNT(species.quantity) WHERE TRUE GROUP BY species.name EVERY n STEPS;
```

Query 2:

```
INSTRUMENT model OBSERVE COUNT(species.quantity) WHERE (species.name = 'Endosome' AND
  species.attribute(2) = 'late') GROUP BY species.name EVERY n STEPS;
```

Query 3:

```
INSTRUMENT model OBSERVE SUM[DOUBLE](species.attribute(1)), AVG[DOUBLE](species.attribute(1)),
  MIN[DOUBLE](species.attribute(1)), MAX[DOUBLE](species.attribute(1))
  WHERE species.name = 'Endosome' GROUP BY species.name EVERY n STEPS;
```



(a) Runtime of observation code without data storing.

(b) Runtime of observation code including time for direct writing of the observations to the disc.

Figure 7: Test queries (top) and runtime results (bottom). Presented are the minimal runtime values for each of the setups. The 4th setup comprises all three queries in the same run and the red line indicates the computation time without observation. The instrumentation interval n is coded in the colors.

In our first experiment we turned off the writing of observed data to the disc as we are interested in the pure overhead induced by observation. In our second experiment we turned on data storing to show that the overhead shown in the first experiment gets worse the more to be done. Data to be written is proportional to the frequency with which the data is collected. In our example to collect data after every step leads for Query 4 to approx 300 MB, every 10th step to 30 MB and every 100th step to 3 MB of data on the disc per single computation.

For Figure 7(a) we used the minimal values for the computations among the replications as these are the closest approximation of the theoretical speed. As expected, the effort to collect data for each single step is higher than the effort to collect data only for each 10th or 100th step. The overall effort is too low to identify any additional speedup between 10 and 100 in the example we used.

Figure 7(b) shows the results with activated writing to a disc. The time differences already identifiable in Figure 7(a) become more prominent in this second experiment. We have not executed further experiments with larger data sets as the overhead gets already visible using this rather "small" amount of data written.

The most complex query (Query 4) is still only a subset of what could be queried from a model. This short performance analysis shows for a single example model that the number of entities to be observed, and even more important, the frequency at which they are observed play an important role for the overall performance. Thus the performance study shows that collecting data can be expensive and that it is better to carefully select what to observe and when to observe.

8 CONCLUSION

Collecting data is of central importance in executing simulation studies. Therefore, flexible means are required. We identified a set of requirements for such means. To those belong, the ability to work on

selected changes from specified model entities, selected data from computation algorithm entities, counts of sets of selected model entities, data about sets of selected reactions/events executed, variable model structures, simple statistics (like averages, etc.) from specified model/computation algorithm entities, and to record all changes, changes every n steps, or after n time units. To address these requirements we have presented a first proposal of a language to describe what shall be observed during computing a simulation model. The SQL-inspired language provides basic constructs to select what to observe, when to observe, and how to observe the data. In applying the language to observe the computation of ML-Rules models, it has been shown that the language allows a compact specification of collecting, storing, aggregating, and filtering observations. Thereby, the effort to adapt the language to the needs of models defined in ML-Rules proved reasonable. Only three statements in the language need to be kept flexible so that references to models in a modeling language can be made. Our performance study indicated an overhead introduced by observation which gets worse the more to be observed and stored: it shows that the runtime overhead is mostly caused by the frequency of observation. The need for a selective instrumentation is once more illustrated by the observation of the file sizes. Future work will deal with using the language for further modeling means.

ACKNOWLEDGMENTS

This research is supported by the DFG (German Research Foundation).

REFERENCES

Amoretti, M., M. Agosti, and F. Zanichelli. 2009. "DEUS: a discrete event universal simulator". In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, 58:1–58:9. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Blinov, M. L., J. R. Faeder, B. Goldstein, and W. S. Hlavacek. 2004. "BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains". *Bioinformatics* 20 (17): 3289–3291.

Conte-Zerial, P. D., L. Brusch, J. C. Rink, C. Collinet, Y. Kalaidzidis, M. Zerial, and A. Deutsch. 2008. "Membrane identity and GTPase cascades regulated by toggle and cut-out switches". *Molecular Systems Biology* 4:206.

Dalle, O., and C. Mrabet. 2007, September. "An Instrumentation Framework for component-based simulations based on the Separation of Concerns paradigm". In *Proc. of 6th EUROSIM Congress (EUROSIM 2007)*. Ljubljana, Slovenia.

Dalle, O., J. Ribault, and J. Himmelspach. 2009, December. "Design considerations for m&s software". In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 944–955. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. Invited paper.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, USA.

Himmelspach, J., and A. M. Uhrmacher. 2007, March. "Plug'n simulate". In *ANSS '07: Proceedings of the 40th Annual Simulation Symposium*, 137–143. Washington, DC, USA: IEEE Computer Society.

Hoops, S., S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. 2006. "COPASI—a COmplex PAthway SImulator". *Bioinformatics* 22 (24): 3067–3074.

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

Moss, S., H. Gaylard, S. Wallis, and B. Edmonds. 1998. "SDML: A Multi-Agent Language for Organizational Modelling". *Comput. Math. Organ. Theory* 4 (1): 43–69.

Poteryaev, D., S. Datta, K. Ackema, M. Zerial, and A. Spang. 2010. "Identification of the switch in early-to-late endosome transition". *Cell* 141 (3): 497–508.

Rohr, C., W. Marwan, and M. Heiner. 2010. "Snoopy—a unifying Petri net framework to investigate biomolecular networks". *Bioinformatics* 26 (7): 974–975.

Uhrmacher, A. M. 2001. "Dynamic Structures in Modeling and Simulation - A Reflective Approach". *ACM Transactions on Modeling and Simulation, Vol.11. No.2*:206–232.

Vaucher, J. G., and P. Duval. 1975. "A comparison of simulation event list algorithms". *Commun. ACM* 18 (4): 223–230.

Wang, B., J. Himmelspach, R. Ewald, and A. M. Uhrmacher. 2009, December. "Experimental Analysis of Logical Process Simulation Algorithms in JAMES II". In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 1167–1179. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

TOBIAS HELMS holds a bachelor in computer science from the University of Rostock, Germany and is currently a master student. His email address is tobias.helms@uni-rostock.de.

JAN HIMMELSPACH is a post doc in the Computer Science Department at the University of Rostock. He received a diploma in Computer Science from the University of Koblenz and his doctorate in Computer Science from the University of Rostock. His research interests are in software engineering for modeling and simulation, credibility of modeling and simulation, and on efficient modeling and simulation solutions. His email address is jan.himmelspach@uni-rostock.de.

CARSTEN MAUS is a PhD student in the Computer Science Department at the University of Rostock. He received a diploma in Biology from the University of Düsseldorf. His research interests are in multilevel modeling of cell biological systems. His email address is carsten.maus@uni-rostock.de.

OLIVER RÖWER holds a diploma in computer science from the University of Rostock, Germany. His email address is oliver.roewer@uni-rostock.de.

JOHANNES SCHÜTZEL holds a bachelor in computer science from the University of Rostock, Germany and is currently a master student. His email address is johannes.schuetzel@uni-rostock.de.

ADELINDE M. UHRMACHER is professor at the Institute of Computer Science at the University of Rostock and head of the research group Modeling and Simulation. Her research interests are in developing effective modeling and simulation methods, particularly for multilevel, dynamic structure systems, and applying those methods in areas like smart environments, demography, and computational biology. Her email address is adelinde.uhrmacher@uni-rostock.de.