Back-annotation framework for Simulation Traces of Discrete Event-based Languages

Ábel Hegedüs, István Ráth, and Dániel Varró

Budapest University of Technology and Economics, Hungary
{hegedusa,rath,varro}@mit.bme.hu

Abstract. Back-end analysis tools aiming to carry out model-based verification and validation of dynamic behavioral models frequently retrieve sequences of simulation steps (called execution traces) as their output. Unfortunately, as each back-end tool typically has a different representation of such traces, even the simple replay of an execution trace within a modeling environment (i.e. outside the analysis tool) needs significant investment. In the paper, we present a generic handling for recording execution traces. Our approach complements static and dynamic metamodels by introducing a generic execution trace metamodel, which can be specialized to dedicated execution trace information for individual DSMLs. Furthermore, we present techniques to (i) automatically generate execution traces from changes of the underlying model, and (ii) to drive simulation according to an execution trace model. Our approach will be exemplified by the modeling language and trace information of the SAL model checker.

1 Introduction

Model-driven analysis aims at revealing conceptual flaws early in the design process. In the typical approach, high-level design models (UML, BPEL [16], SysML, etc.) are automatically transformed into mathematical models (e.g. Petri nets, transition systems, process algebras) to carry out analysis by formal methods. The results of the analysis are then attempted to be back-annotated to the original source model. In case of modeling languages with dynamic behavioral semantics (e.g. statecharts, workflows, etc.), the target formal analysis tools frequently carry out simulation or model checking to ensure the (functional) correctness of the design or validate its non-functional characteristics (e.g. performance).

In such dynamic scenarios, these back-end analysis tools typically retrieve an execution trace (run) of the system as a positive or negative example (see Fig. 1). Unfortunately, in most cases, a variety of back-end analysis tools are used, all of which have different textual representation of their execution traces. As a result, even the simple replay of an execution trace within a modeling environment (like Eclipse) needs significant investment, which has to be repeated for each individual analysis tool (not to mention further back-annotation problems).

In the paper, we aim at providing a generic handling for execution traces in dynamic modeling languages with a specific focus on those retrieved by model

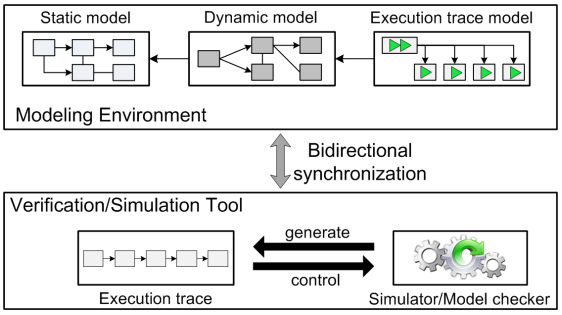


Fig. 1: Modeling approach overview

checkers and simulation tools. First, (1) we propose a metamodel for execution traces, which complements traditional static and dynamic metamodels (Fig. 1). Based on this metamodel, (2) we define operations to support the replay of such traces within a general purpose modeling environment (outside the original analysis tools). Finally, (3) we investigate how execution trace models can be produced and consumed using live (event-driven) model transformations (such as how changes of dynamic models drive the automated creation of execution trace models). Our techniques will be exemplified on the language and execution traces provided by the SAL model checker [4].

The rest of the paper is structured as follows. Sec. 2 provides a brief introduction to the language aspects of the SAL model checker. In Sec. 4, we present a metamodel of execution traces. Sec. 5 discusses how an execution trace model can be replayed to update the dynamic model, while Sec. 6 describes how execution trace models can be created. Finally, related work is discussed in Sec. 7 and Sec. 8 concludes our paper.

2 Background

We provide an informal introduction to the language of the SAL model checker, which serves as the running example of the paper (Sec. 2.1). Then we discuss a way how dynamic SAL models can be integrated in a modeling framework using dynamic metamodeling techniques (Sec. 2.2).

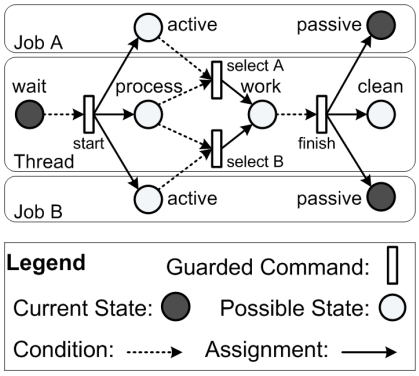
2.1 The SAL Language

Symbolic Analysis Laboratory (SAL) [4] is a framework for combining different tools to calculate properties of concurrent systems and it includes a simulator and advanced tools for symbolic and bounded model checking. These tools are used on input models captured as a transition system using a language also called SAL. Models written in the SAL language consist of three parts: the variable type definitions, the module specifications and the requirements.

The *variable types* can be finite types (e.g. boolean, tuple), infinite types (e.g. numbers), or subtypes. For the current paper, we will restrict our examples to tuples where the type declaration defines a finite number of possible *values*. The *specification of a module* consists of state variable *declarations*, variable *initializations* and the *transitions* part. The state of the system model is defined by the current value of the state variables, while the evolution of the system is specified by transitions.

For variable initialization, SAL uses *definitions*, which are of the form $x = expression$ or $x \in set$ (nondeterministic choice). The x' form can refer to the new value of variable x in a transition. The initialization of variables is given as a combination of definitions [4]. Transitions are *guarded commands* defined in the form $g \rightarrow S$ where g is a boolean guard and S is a list of definitions (assignments). A guarded command is *enabled* if the boolean guard evaluates to true based on the actual state of the system. The executed command is chosen from the set of enabled commands nondeterministically. The execution consists of applying the definitions in S by setting the new value of the contained variables.

Example SAL transition system We introduce a simple SAL example modeling a thread that may perform one of two jobs during its run. Fig. 2 shows a possible graphical notation of the SAL system on the left and the real textual syntax of the same example on the right.




```
ThreadExecution : context = begin
  threadType : type = {wait,process,work,clean};
  jobType : type = {active,passive};

TwoJobs : module = begin
  local thread : threadType
  local jobA, jobB : jobType
INITIALIZATION
  thread = wait; jobA = passive; jobB = passive;
TRANSITION[
  start : (thread = wait) -->
     thread '= process; jobA '= active; jobB '= acitve; []
  selectA : (thread = process AND jobA '= active) --> thread '= work; []
  selectB : (thread = process AND jobB '= active) --> thread '= work; []
  start : (thread = work) -->
     thread '= clean; jobA '= passive; jobB '= passive;]
end
end;
```

Fig. 2: Example transition system

The **Thread** starts in the *wait* state while both **Job A** and **Job B** are initialized in the *passive* state. The guard of the **work** command requires that the thread has to be in the *wait* state, while the assignments set the jobs in the *active* state and the thread in the *process* state. There are two commands **select A** and **select B** with similar guards (one of the jobs in the *active* state and the thread in the *process* state) and the same assignments (set the thread in the *work* state). Finally, the guard of the **finish** command requires the thread

to be in the *work* state and its assignments set the the jobs back to *passive* state and the thread in the *clean* state.

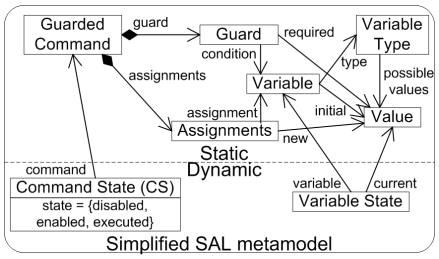
2.2 Dynamic metamodeling for behavioral models

Dynamic metamodeling (DMM) [11] aims at specifying the dynamic behavior of executable modeling languages by combining metamodeling with rule based formalisms to capture operational semantics.

In DMM, the *static metamodel* describing the structure of models in the target language is extended with a *dynamic (execution) metamodel* capturing the dynamic state of models during execution or simulation. Then, the dynamic (behavioral) semantics of the language is defined by transformation rules that modify the instances of the dynamic metamodel. These operational rules are frequently formally defined by graph transformation techniques [9]. As a result, DMM specifications are suitable for describing a complete execution semantics of dynamic (behavioural) modeling languages.

Furthermore, DMM also enables to simulate dynamic models using existing transformation engines. For this purpose, the applicability of each simulation rule is first checked, for instance, by graph pattern matching techniques. Then a rule is applied for a selected match (if any exists), which updates the underlying dynamic model to result in a new (dynamic) state. This selection can be nondeterministic or user-driven as well. Simulation rules can be fired as long as an enabled rule is found. This form of simulation is widely used in graph transformation tools.

Dynamic metamodeling example The dynamic metamodeling is illustrated to define semantics for transition systems of SAL. The left part of Fig. 3 shows a (simplified) metamodel for SAL including both static and dynamic aspects.




```
rule executeCommand() = seq{
 choose Cmd with
   find EnabledCommand(Cmd) do seq{
  call changeCommandStateToExecuted(Cmd);
  forall Asnt with
   find CommandsAssignment(Cmd,Asnt) do
    choose Var,Value with find
      AsntsVarAndValue(Asnt,Var,Value) do
     call setVariableValue(Var,Value);
  forall Cmds with find Command(Cmds) do
   call calculateCommandState(Cmds);}}
```

Fig. 3: SAL system model and command execution transformation rule

The Static Metamodel contains **Guarded Commands** which have a **Guard** and an **Assignments** contained element and **Variables** which have a **Variable Type** defined in the declaration part of the model. Their **initial** state is one from

the **possible values** of their type. The guards contain several variable-value conditions declaring the **required** value for the given variable. The assignments also contain variable-value assignments, defining the **new** value for the variables.

The Dynamic Metamodel contains **Command State** elements, which store the dynamic state of the *command*. A Command State can be *disabled* (when the guard condition is false), *enabled* (when the guard condition is true), or *executed* (to denote that the command has just fired). The **Variable State** element records the *current* values of the corresponding variable.

Command execution rule The execution of a command can be defined in a transformation rule based on the semantics of the SAL system when firing a guarded command. First one command (**Cmd**) is chosen non-deterministically from the enabled commands (pattern matching returns a match). Then the command state of **Cmd** is set to *executed* (the attribute value is modified) and all the assignments of **Cmd** (**Asnt**) are processed (all matches of a pattern) by updating the current value of variables to the state defined by **Asnt** (change target of the *current* relation). Finally the state of every command (all found matches) is refreshed based on the new variable states (by calling an evaluator rule).

Dynamic models without dynamic semantics Note that dynamic models (i.e. an explicit representation of execution-time information) are highly relevant in themselves, i.e. they do not depend on the actual formalism we use for defining the dynamic semantics of the language. For instance, when the SAL model checker retrieves a counterexample, the existence of a precise dynamic semantic specification is not compulsory to replay the same scenario within a general purpose modeling tool. In fact, most results of the paper are independent of the underlying simulator, and only depend on the dynamic model itself.

3 Generic back-annotation framework for dynamic modeling languages

Back-annotation is the reverse model transformation problem to derive corresponding source design model information from the results of a formal analysis carried out on a target model. In case of *discrete event-based dynamic analysis languages*, these results are typically represented as an *(execution) trace* obtained from a simulation run or as a counter-example (i.e. sequence of steps leading to the violation of a requirement) of a model checker. Therefore the back-annotation in this case consists of deriving a trace of the source model from a given trace of the target model.

In the paper, we first propose a generic framework for the back-annotation of simulation traces. In this framework, we assume the existence of the following traditional modeling and transformation concepts:

– **static, dynamic and trace metamodels** for both the source and target domains to precisely define the taxonomy for instance models;

- **operational semantics** to specify the dynamic behaviour of the target analysis model during simulation;
- **structural model transformation** which derives a (static) target model from an arbitrary source model;
- **traceability links** created by the model transformation to store the structural correspondence between the elements of the source and target models.

Using these concepts back-annotation necessitates the definition of precise transformations for:

1. **trace generation rules** derived from the operational semantic rules of the model to record a simulation run as a trace model observing the changes of the dynamic model.
2. **trace processing rules** to replay the execution of a simulation run on the dynamic model using an arbitrary trace of the same model.
3. **trace mapping (back-annotation) rules** to derive a source trace model from the target trace model using traceability links existing between the models.

In this paper we focus on the problem of generating a trace from the output of the simulation and processing the persisted traces.

3.1 Metamodels of dynamic modeling languages

We assume the existence of various metamodels in the context of a *dynamic modeling language (DML)*, which are exemplified together with main relationships in Fig. 4.

First, a **static metamodel** MM_{stat} defines the static structure of a language including possible types of model elements, their main attributes and relations with other model elements. An instance of this metamodel is called the **static model** (M_{stat}), e.g. a concrete Petri net structure.

Next, a **dynamic metamodel** MM_{dyn} uses and extends the static metamodel MM_{stat} for storing information related to dynamic behaviour (e.g. current state, value, configuration) of a structural element. The **dynamic model** (M_{dyn}) is an instance of the MM_{dyn}, e.g. the current marking of a given Petri net place.

Finally, a **trace metamodel** (MM_{trc}) is defined for the language to represent simulation runs of the M_{dyn}. The MM_{trc} *uses* the MM_{dyn} for recording how the dynamic model changed and the MM_{stat} for describing which static element is concerned. A **trace model** (M_{trc}) is an instance of the MM_{trc}, e.g. the sequence of fired transitions of a Petri net. The M_{trc} describes the changes of M_{dyn}, therefore it is represented as a *change model* in terms of [5].

While execution traces are traditionally interpreted as a sequence of elementary operations, in the current paper, we use hierarchical trace models consisting of *micro steps* (atomic operations on M_{dyn}) and *macro steps* (complex operations on M_{dyn}), which is compliant with recent approaches [12] to define semantics for big-step DMLs like statecharts.

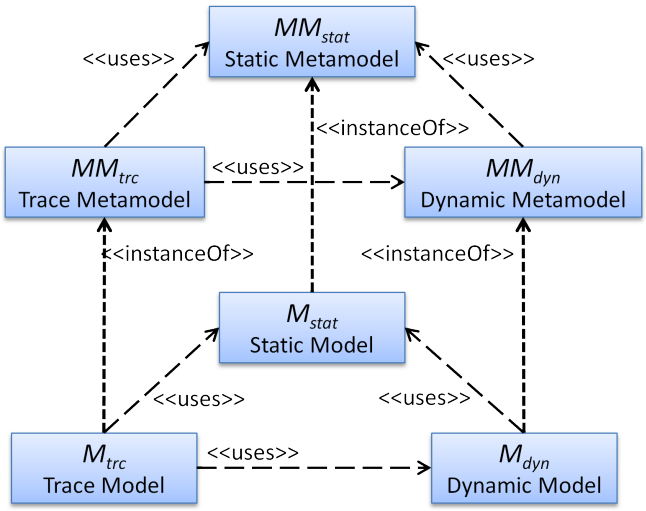


Fig. 4: Metamodels and instances of a dynamic model

Given the set of micro steps as terminal symbols T, the hierarchy of macro steps as non-terminal symbols NT, and a grammar Gr implied by the operational semantics description of the language (see below), a trace model can be formally interpreted as the abstract syntax tree AST of a well-formed sentence of the grammar GR.

3.2 Operational semantics and traces for dynamic models

The simulation of a DML is performed in accordance with the *operational semantics* of the language defined by simulation rules. In our framework we assume that simulation rules are defined as intra-model transformations illustrated in Fig. 5 (see also [10,7,21]).

The execution of a rule in the transformation $MT_{sym} : (M_{stat}, M_{dyn})\Delta M_{dyn}{}'$ modifies the M_{dyn} by also taking into account M_{stat} and results in a new $M_{dyn}{}'$. During a simulation run, the changes of the dynamic model are recorded as a sequence of micro steps as part of the derived trace model M_{trc}. Furthermore, the hierarchy of macro steps in M_{trc} is in direct correspondence with the transformation rules fired during the simulation run. Therefore, we assume that the simulation rules MT_{sym} imply a grammar GR, which defines well-formed execution traces.

The M_{dyn} is used as an input to create the trace model (M_{trc}) for a specific simulation run using a **trace generating transformation** ($CDT_{gen} : CM_{dyn} \mapsto M_{trc}$) which can be *automatically generated* from MT_{sim}. The created M_{trc} contains the micro steps of the change in a macro step hierarchy.

The M_{trc} can be used to replay the execution of a specific simulation run. The **trace processing transformation** ($CDT_{proc} : M_{trc} \mapsto CM_{dyn}$) generates

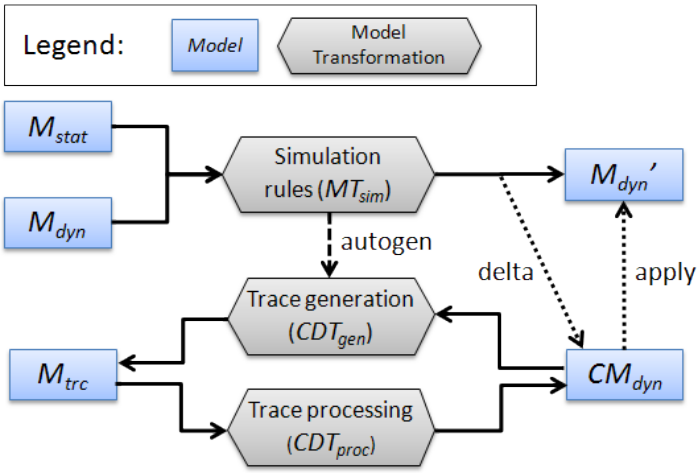


Fig. 5: Simulation and trace generating transformation

dynamic model changes that are applied to M_{dyn}, after which the model state (M_{dyn}') will be the same as after the execution of a simulation rule.

3.3 Forward model transformation with traceability links

We assume the existence of a unidirectional structural model transformation $MT_{src2trg}$ (see Fig. 6) which generates a static target model M^{trg}_{stat} from a given static source model M^{src}_{stat}. This MT is also responsible for deriving the initial state of M^{trg}_{dyn} from the initial state of M^{src}_{dyn}.

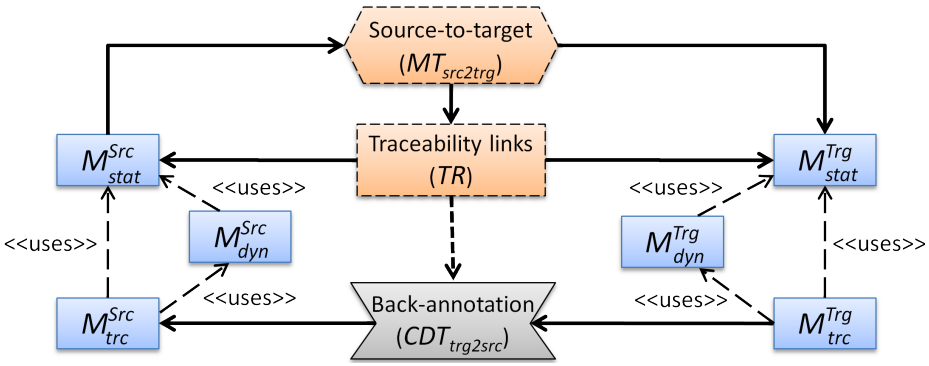


Fig. 6: Forward model transformation and back-annotation

We also assume that this transformation generates **traceability links** (TR) between the source and target models in order to record the structural corre-

spondence between the model elements. As we only rely upon these traceability links for back-annotation, any kind of forward model transformation approaches and tools can be used.

3.4 Back-annotation of dynamic execution traces

In the paper, back-annotation of a target DML to a source DML is defined as a transformation $CDT_{trg2src}$ which is able to generate the M^{src}_{trc} from an arbitrary M^{trg}_{trc} if such source trace exists. The $CDT_{trg2src}$ makes use of the TR to identify corresponding elements in source and target models.

Given that the traces contain model changes, we propose to define the $CDT_{trg2src}$ as a change driven model transformation [22].

In many practical cases, no formal operational semantics is available for the source DML (e.g. in case of UML or BPEL). Or in other terms, their formal semantics is defined in denotational way by mapping them to a formal target DML like Petri nets [26,14]. It is worth pointing out that our back-annotation framework only assumes that

- a target trace M^{trg}_{trc} is made available by some analysis tool, which is compliant with the formal operational semantics GR^{trg} of the target (analysis) DML,
- the macro steps of the source (design) DML can be identified based on an informal behavioural description.

4 Execution trace models

4.1 Capturing execution traces in dynamic models

Our overall goal is to provide a generic framework for replaying an execution trace retrieved by a back-end analysis tool within a general modeling framework (e.g. EMF). The replay mechanism is generic enough to be reusable and easily adaptable for various dynamic modeling languages. Such an investment would also significantly reduce the cost of back-annotation for different pairs of source and target languages.

For that purpose, we first introduce a *generic execution trace metamodel*, (Sec. 4.2) which complements dynamic metamodels by providing an abstraction how a counter example or simulation trace is captured in various analysis tools. Obviously, this metamodel need to be refined when integrating a specific analysis tool (like SAL), but generic replaying operations for an execution trace will be captured already on this abstract level (Sec. 4.3).

Then we define a bidirectional synchronization technique between execution trace models and dynamic models. In other terms, we demonstrate how a step in the execution trace model can be replayed by updating the dynamic models appropriately (Sec. 5). Furthermore, we also describe how an execution trace model can be automatically derived during a simulation run by observing only the changes of the underlying dynamic model (Sec. 6).

4.2 Abstract execution trace metamodel

Essentially, an execution trace model captures the changes between two subsequent states of the dynamic model. This way, the execution trace metamodel (see Fig. 7) complements the existing static and dynamic metamodels as well.

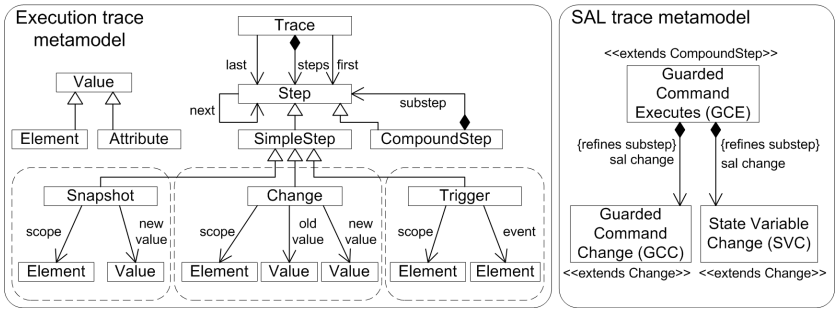


Fig. 7: Execution trace metamodels

Trace is the root element of the execution trace model, which contains the (top-level) *steps* of the recorded execution. The *last* relation specifies the last step that was executed in the simulation (i.e. the last changes occurred). The *first* relation defines the beginning of the trace (wrt. a specific run).

Step is an abstract representation of one or more dynamic model changes which occur within the same atomic transaction. The sequence of changes happening after each other define an ordering between the steps represented by the *next* relation (where the source step precedes the target in the trace).

As we observed in the traces retrieved by various back-end analysis tools, steps are frequently organized into a step hierarchy. As a consequence, we also distinguish between *SimpleSteps* (i.e. elementary changes in the dynamic model), *CompoundSteps*, which themselves contain several lower-level steps (as represented by *substeps* aggregation).

Representations for simple steps *SimpleSteps* record elementary changes specific to a certain model element in the underlying dynamic model (called the *scope* of the step) as retrieved by the model checker or simulator in an execution trace. However, different back-end tools record such information in different style. In fact, different bits of information within a trace can be represented differently even within a single tool.

We found three essentially different ways how transitions of the dynamic model are persisted in an execution trace in different tools, which we call (1) *snapshot* (which store everything for the new state), (2) *change* (which store the delta between the old and the new state), and (3) *trigger* (which saves the event caused the transition in the dynamic model).

Snapshot This simple representation is used when every step in the trace stores all the dynamic information required to describe the new state of the model element. A single snapshot element therefore records a reference to a dynamic model element (*scope* relation) and the value representing its dynamic state after the step (*new value* relation). This approach is usually taken in analysis tools when significant changes are possible in two neighboring states in the execution trace of the dynamic model (e.g. parallel workflows). It is easy to implement but results in complex (and verbose) execution trace representations.

Change This simple step represents only the modification (delta) between two subsequent states of a dynamic model element. A change element stores the reference to the model element (*scope* relation) and both the dynamic state before the modification (*old value* relation) and after (*new value* relation). Recording the modifications of the dynamic model is effective when the number of updated dynamic elements in a given step is limited (as it stores more information for a model element compared to snapshots). For example transition systems or Petri nets.

Trigger The third step type can be used for event-driven languages (like statechart simulators) where the update of the dynamic model is *triggered* by an event (e.g. receiving a message). In this case, the step only records the event itself (*event* relation) instead of all the effects of triggered by the event. Furthermore, the dynamic model element directly targeted by the event (e.g. the receiver of the message) is stored (*scope* relation). This representation is advantageous when effects of the event can be (forward or backward) simulated knowing the current state of the dynamic model and the trigger itself. As a result, only this third step type relies upon the existence of dynamic operational semantics to drive simulation over the dynamic model.

It is worth highlighting that a trace information may contain a combination of these step representations for the different dynamic elements, which means that our trace model is highly adaptable.

Navigation Our step representation also respects navigation constraints, i.e. it is possible to navigate in the trace both forwards and backwards without having to traverse the whole trace model to find the required modifications of the dynamic model. Therefore the presented simple steps ensure that only the current and either the next or previous step is required to navigate in that direction.

Dynamic model elements The relations existing between the execution trace metamodel and the dynamic execution model have two kind of targets. *Elements* of the dynamic model are one, while *values* may be either model elements or *attributes* (e.g. string, integer, boolean, double, float). In the current paper, we assume that elements of the static metamodel are left unaltered during the replay of an execution trace model, e.g. we do not create or delete guarded commands, for instance.

4.3 Tool-specific execution trace metamodels

When integrating the trace representation of a specific tool (e.g. SAL in our case), the abstract execution trace metamodel can be refined and adjusted to needs of the tool. For this metamodel refinement, we rely upon standard generalization between classes and relations (as available in MOF or in ontologies).

In case of the SAL language, the top-level simulation steps are the execution of guarded commands while the modifications are variable assignments. Furthermore, the dynamic model stores the actual state of transitions as well, which changes along with the variable state changes. The additionally defined types are shown on Fig. 7 and detailed in the following. An example for the instance model can be seen on Fig. 9.

Guarded Command Execution is the elementary simulation step in SAL transition systems. It is specialized from Compound Step and contains the dynamic model changes caused by the execution of the command. The *substeps* aggregation is also refined as *sal_change* which targets the new change types.

State Variable Change is specialized from *Change* step and describes the modification of SAL variable during the execution of the commands variable assignments. The *scope* of the change is the variable while the *values* target the scalar elements representing the state of the variable before and after the command execution.

Guarded Command Change is also specialized from the Change step and represents that the execution of the command may change the state of other commands as well (enabling or disabling them) and can be used also to store which command was executed. Note that in the example, *selectA* and *selectB* are different commands with equal effects thus the step has to record which was executed. The *scope* of the change is the command and the *values* are the state before and after the step.

5 Replaying Execution Traces

Execution trace models record scenarios retrieved by a run of an external simulator or model checker (e.g. SAL) in a form which is independent of the back-end analysis tool and compatible with an underlying modeling framework.

Now, we show how such execution trace models can be replayed within a modeling framework. For this purpose, we distinguish between two cases, (1) when only the dynamic model of the language is available without operational simulation rules (e.g. Fig. 3), and (2) when operational rules are also available in addition to the dynamic model itself. In this latter case, simulation rules can be modified directly. Howver, since the first case handles a more generic situation, our investigations will be primarily directed that way.

So, in the more general first case, replaying the trace requires the processing of the subsequent step in the execution trace model, and a direct update of the underlying dynamic model accordingly. We use model transformations for this

purpose (Sec. 5.1).Furthermore, we also propose a simple interface providing basic operations to drive the replay of execution trace models within the modeling framework (Sec. 5.2).

5.1 Processing execution trace models to update dynamic models

In general, in order to update the underlying dynamic model, the replayer tool needs to have access to the current state of the dynamic model and the next step in the execution trace, which captures how the current state needs to be changed. This is again illustrated using our SAL transition system as an example. Fig. 8 demonstrates how the execution trace model (top) is used for stepping forward (imitating the execution of a guarded command) and modifies the dynamic model. Note that the illustration is simplified by leaving out most relations defined in the metamodel between the execution trace model and the dynamic model. These are similar *thread_clean* to for each substep. Also, the substeps of *selectA_executes* are hidden for clarity.

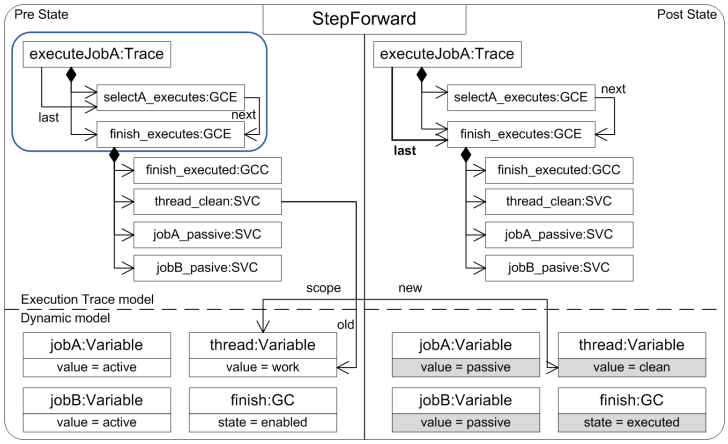


Fig. 8: (Forward) Replaying execution traces

First the last processed step in the trace (*selectA_executes* of type *Guarded Command Executes*) and the subsequent step (*finish_executes*) are retrieved along the *last* and *next* relations. Then the *State Variable Change* elements (*thread_clean*, *jobA_passive*, *jobB_passive*) are processed and the corresponding variables (*thread*, *jobA* and *jobB*) are updated in the dynamic model. Finally the *Guarded Command Changes (GCC))* are processed and their state modified to *executed* using the *new value* relations. The update of dynamic models can be automated using transformation rules.

5.2 Trace manipulation interface

In our prototype implementation, operations of the trace manipulation interface are implemented by transformations over the generic execution model, i.e. they are highly domain-independent. However, for space considerations, we only informally describe the main tasks carried out by (1) *complex interface operations* for traces, which are assembled from (2) *elementary trace manipulation operations*.

Interface for trace manipulation Our initial prototypical implementation[1] contains the following three high-level trace manipulation operations, which are directly available from the graphical user interface to navigate in an execution trace model, and keep the dynamic model synchronized with the actual position in the trace.

Step forward This function finds the last step in the trace (either by the *last* or *first* relation) and if there exists a next step in the execution trace then that step is processed and every modification represented by substeps is carried out on the dynamic execution model. The *last* relation is updated to target the processed step.

Step backward One of the advantages of the execution trace model is the ability to navigate in either direction along the execution. This function can be used to revert the modifications of the actual step on the dynamic model. First the last step is retrieved using the *last* relation, then the modifications are processed depending on the step type used. Snapshot steps are reverted by finding the previous step and updating the dynamic model based on its substeps. Change substeps can be rolled back by using the *old value* relation to set the scope element in its previous state. However, reverting a Trigger step is not always feasible. Finally, the *last* relation is set to target the previous step, if there is one, otherwise the relation is deleted.

Reset This function can be used to roll back the execution to the beginning of the trace. It can be implemented by (1) collecting the initial values from static model or (2) storing the initial state in the first step of the trace. In both cases, the initial state of the dynamic model is resumed and the *last* pointer is deleted to signal that the execution is in its initial state.

Skip to end TODO: Maybe, we should also introduce fast forward to the end of the trace, if this is not too complex

These functions provide the most useful functionality required for a user to replay and simulate the execution stored in the execution trace model. Furthermore, they also enable automated animation by calling the interface repeatedly using short time intervals between calls.

[1] More information available at http://viatra.inf.mit.bme.hu

Elementary trace manipulation operations In order to provide these high-level user interface operations, elementary operations have also been defined to manipulate and traverse execution trace models. To increase generality, these operations were defined directly over the generic execution trace metamodel.

- **firstStep(): Step** Find the first step of the trace to resume simulation.
- **lastStep(): Step** Find the last step of the trace to resume simulation.
- **nextStep(): Step** Traverse the trace horizontally to find the next step from the *last* position.
- **previousStep(): Step** Traverse the trace horizontally to find the previous step from the *last* position.
- **unfoldStep(Step): Step*** Traverse the trace vertically to find substeps of a given step and ensure that every modification is processed.
- **getDynamicElement(Step): Element** Return the corresponding dynamic model element for a given simple step.
- **executeStep(Step** Modify the dynamic execution model using Snapshot and Change step representation types.

Processing Trigger steps. The processing of a trigger step is carried out differently as it only records the event, which triggered the execution of a macro step (sequence of elementary steps, as in case of statecharts), but not the actual modifications themselves. For this reason, in this case we assume the existence of operational semantic rules for processing the events within the modeling framework. As a consequence, a trigger step can be processed by (1) accessing the event of the trigger step, (2) offering it to the operational semantic rules, when (3) the update of the dynamic model is carried out by these simulation rules.

5.3 Elaboration on the generic trace handling

Traces persisted with the generic trace metamodel can be replayed without defining a completely new transformation for every specific language. In this section we show how the low-level operations and high-level functions of the trace manipulation interface are implemented in VIATRA2.

Horizontal traversing of a trace We define graph patterns for traversing the trace on a given hierarchy level. Listing 1.1 shows the patterns for finding the last executed step in the trace (*lastStep*), the step following a given step (*nextStep_*) and the step following the last executed step (*nextStep*). Note that the *nextStep* pattern is used to return the previous step as well by finding a match for *Step* with the *NextStep* known.

```
pattern lastStep(Step,Trace) = {
 trace(Trace);  step(Step);
 trace.last(R,Trace,Step);
}
pattern nextStep_(Step, NextStep) = {
 step(Step); step(NextStep);
 step.next(R,Step,NextStep);
```

```
}
pattern nextStep(NextStep,Trace) = {
 step(NextStep);
 find lastStep(LastStep,Trace);
 find nextStep_(LastStep,NextStep);
}
```

Listing 1.1: Last and Next step patterns

Vertical traversing of a trace The substeps of a step are retrieved in order when traversing the trace vertically. Listing 1.2 shows the graph pattern that searches for substeps in a higher-level *Step*. When looking for the first substep, the *nextStep_* pattern is used to ensure that the selected substep has no preceeding step. Otherwise, the same pattern is used to find the next substep.

```
pattern unfoldStep(Step, LastSubstep, Substep) = {
 // first substep
 check(LastSubstep == undef);
 step(Step); step(Substep);
 step.substep(R,Step,Substep);
 neg find nextStep_(BeforeFirst,Substep);
} or {
 // more substeps
 step(Step); step(Substep); step(LastSubStep);
 step.substep(R,Step,Substep);
 step.substep(R2,Step,LastSubstep);
 find nextStep_(LastSubstep,Substep);
}
```

Listing 1.2: Unfold step pattern

Forward stepping Listing 1.3 shows the generic implementation of the forward stepping function in the VIATRA2 transformation language. First, the *Step* following the *last* executed step of the trace is found. Then the *last* relation is updated to record that the stepping in the trace. Next the substeps of *Step* are retrieved in order and executed.

```
rule stepForward() =
 choose Step with find nextStep(Step) do seq{
  call setLastRelation(Step);
  iterate choose Substep with
     find unfoldStep(Step,LastSubstep,Substep) do seq{
   call executeStep(Substep);
   update LastSubstep = Substep;
  }
 }
```

Listing 1.3: Forward stepping

Executing steps The simple step types *Snapshot* and *Change* both refer to a model element and a value corresponding to the element. Listing 1.4 shows the graph patterns defined for retrieving this information from the persisted *Step*.

```
pattern stepScope(Step, Scope) = {
 step(Step);
 element(Scope);
 step.scope(_,Step,Scope);
}
pattern stepValue(Step,Value) = {
 change(Step);
 value(Value);
 step.newValue(_,Step,Value);
} or {
 snapshot(Step);
 value(Value);
 step.value(_,Step,Value);
}
pattern elementValueRel(Element,ValueRel) = {
 element(Element);
 value(Value);
 element.value(ValueRel,Element,Value);
}
```

Listing 1.4: Execute step patterns

When executing a step, the action depends on the type of the *Step*. Compound steps are unfolded and their substeps are executed in order. Snapshot and Change steps are executed by first retrieving the scope and value elements from the *Step* and the relation between them from the model (VR). Then the target of the relation is replaced with the *Value* persisted in the step.

```
rule executeStep(in Step) = seq{
 if(find CompoundStep(Step) seq{
  // execute substeps
  iterate choose Substep with
     find unfoldStep(Step,LastSubstep,Substep) do seq{
   call executeStep(Step);
   update LastSubstep = Substep;
  }
 } else
 if(find Change(Step) || find Snapshot(Step))
  // find Scope, Value and relation
  choose Scope with find stepScope(Step, Scope) do
   choose Value with fing stepValue(Step,Value) do
    choose VR with find elementValueRel(Step,VR) do
     if(find Element(Value))
      setRelationTo(VR,Value);
 else if(find Trigger(Step))
  call DSMTraceProcessor.executeTrigger(Step);
}
```

Listing 1.5: Execute step rule

Finally, for *Trigger* steps an external transformation is invoked that implements the handling of trigger steps for a given DSM. This solution ensures that the generic transformation can be used for an arbitrary DSM which has it's trace persisted in model conforming to the generic trace metamodel, even if it specialises it for domain-specific Triggers.

6 Creating execution traces during simulation

In order to non-intrusively record the model changes performed by any simulator, we propose an approach which uses the *live transformation* support of

VIATRA [20]. Live transformation rules run in the background, and fire whenever the models are changed (either by a user, the simulator, or by any other means). Thus, the execution trace can be generated by creating trace model elements corresponding to change operations (e.g. create, update, delete) – analogously to the change history model generation approach elaborated in [22]). These special transformation rules are *independently* specified and executed from the simulator, therefore this approach can also be used for custom simulators (e.g. native Java implementations), as long as they operate on models stored in a modeling framework.

6.1 Execution trace generation

The execution of a simulator can be divided into three phases, which are repeated iteratively: (a) *evaluation*, where the possible actions are calculated, (b) *decision points*, where nondeterminism is resolved (e.g. by selecting the action to be executed), and (c) *application*, where the model is modified.

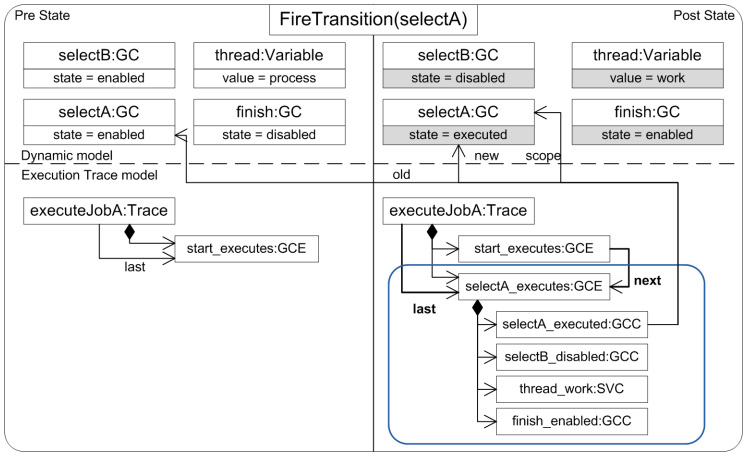
In order to create a complete execution trace, the decisions are recorded in the model at every decision point. However, the decisions in themselves are only understandable if the semantics of the target model is given. Therefore, the modifications of the application phase are recorded as substeps of the decisions. In our example, the decision point can be recognized by the modification of the state of an enabled guarded command to the *executed* state.

Creating the step hierarchy. Every decision point in the simulation corresponds to a top-level *step* in the trace. The bounded parameters can be recorded either as attributes of the step (by creating relations to the selected model elements), or as substeps of the decision. The top-level step is connected to the preceding step in the trace and the *last* relation of the trace is updated to the new step to record which step is built at the moment. The modifications are stored as *substeps* of the decision step.

Trace and runtime model connections. Both the bounded parameters and the modifications refer to the elements of the dymamic model. This correspondence is recorded in the trace model, with relations created in the substeps and pointing at the appropriate elements.

6.2 Trace generation example

We again use the context of the transition system example of Sec. 2.1. The decision phase in the simulation means that an enabled command has to be selected. This decision is stored as a step in the trace model using the *Guarded Command Executes* element introduced in the SAL-specific trace metamodel. Fig. 9 shows both the runtime (up) and trace (bottom) models as they change as follow-up to the execution of the command by the simulator. The illustration is simplified by leaving out most of the relations between the execution trace model and the dynamic model. These are similar *select_executed* to for each substep. Also, the substeps of *start_executes* are hidden for clarity. As the states of dynamic model




```
1 @Trigger
2 rule variableStateChange() = {
3  precondition pattern pre(VS,V) = {
4   VariableState(VS);
5   Variable(Var);
6   VariableState.var(_,VS,Var);
7   Value(V);
8   VariableState.cur(_,VS,V);
9   check(V == cur(VS));
10  Trace(T);
11  GCEx(GCE);
12  Trace.last(_,T,GCE);
13 }
14  action {
15   new(VarStCh(VSC));
16   new(GCEx.sal_ch(_,GCE,VSC);
17   new(VarStCh.var(_,VSC,Var);
18   new(VarStCh.old(_,VSC,cur(VS));
19   new(VarStCh.new(_,VSC,V);
20   update cur(VS) = V;}}
```

Fig. 9: Execution trace generation

elements are updated, these changes are recorded by event-driven transformation rules as *Guarded Command Change* substeps, while variable value changes are persisted as *State Variable Change* substeps. Furthermore, the *last* relation is updated in the trace.

The bottom part of Fig. 9 shows a sample event-driven transformation rule. The `precondition` of the rule describes the expected event (in the form of a graph pattern), while the `action` is performed in case of an event (defined as model manipulations). This rule activates when the *Value* (line 7) of a *Variable* (line 5) changes (line 9). Since the *current* value of every variable is stored during the execution (line 20), the value change (line 15) can be recorded as a substep (*VarStCh* in line 16) of the last guarded command execution (*GCEx* in line 11) in the trace model (line 10).

7 Related Work

While many trace model exists in related work such as [6,29,3,24,2,8]. These approaches generally define static trace models which record the correspondence between various model structures. The current paper focuses dynamic (execution) traces created for sequences of steps.

Execution traces are used in many cases, for understanding distributed systems [15], recovering behaviour [13], improving performance [19]. Dynamic traces were defined for individual languages such as UML sequence diagrams [27], UML Activity Diagrams [23], Concurrent Object-Oriented Petri Nets [18]. However, these solutions lack a generic, domain-independent representation for traces.

In [1] metamodels are introduced for execution traces (as a standalone domain) to record runtime information of program executions. They propose to build the metamodel on KDM [17] and identify several trace types on the programming language level. Unfortunately, the metamodel and the supporting techniques are wrok in progress.

The objective of [23] is to define a Tool-Independent Performance Model for mapping design and architectural models to performance models (used for analyzing system performance design-time). The introduced workbench is designed to include simulation and analysis capabilities and derive execution sequences (scenarios) from UML activity diagrams for driving the simulation.

M3Actions [25] is a framework to develop execution semantics for MOF metamodels. It consists of a graphical editor for defining the structure and behaviour of models, a generic interpreter and debugger for executing them and a trace recorder for storing execution runs. The framework focuses on support for modeling operational semantics and the recorded traces are low-level.

The main contribution of our approach wrt. related work is that the proposed execution trace metamodel and bidirectional synchronization technique are independent from the underlying simulation tool. Furthermore, persisted execution traces can be replayed in a modeling environment without the further use of (external) simulators and model checkers.

8 Conclusion

In the paper, we investigated how execution traces retrieved by model checker or simulation tools can be integrated and replayed in modeling frameworks. We proposed a generic execution trace metamodel, which complements traditional static and dynamic metamodels. Furthermore, we also discussed automated means to (1) *produce* execution trace models by processing changes of the dynamic model, and (2) *replay traces* by updating the underlying dynamic model. As a result traces can be navigated without the use of external analysis tools.

Our generic execution trace model was derived after investigating the traces retrieved by various formal analysis tools (using different modeling formalisms such as Petri nets, transition systems or process algebras). Currently, as an ongoing work, we are investigating how such execution trace models back-annotated to high-level modeling languages such as statecharts or BPEL.

Acknowledgements. This work has been partially supported by the European projects SENSORIA (IST-2005-016004) and SecureChange (ICT-FET-231101).

References

1. L. Alawneh and A. Hamou-Lhadj. Execution Traces: A New Domain That Requires the Creation of a Standard Metamodel. In *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 253–263. Springer Berlin Heidelberg, 2009.
2. B. Amar, H. Leblanc, and B. Coulette. A Traceability Engine Dedicated to Model Transformation for Software Engineering. In J. Oldevik, G. K. Olsen, T. Neple, and R. Paige, editors, *ECMDA Traceability Workshop 2008, Berlin, 12/06/08-12/06/08*, pages 7–16. Springer, June 2008.
3. S. M. Becker, T. Haase, and B. Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Software and Systems Modeling*, 4(2):123–140, May 2005.
4. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rue, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, 2000.
5. G. Bergmann, I. Ráth, A. Ökrös, and D. Varró. Change-Driven Model Transformations: Taxonomy and Language. *Journal of Software and Systems Modeling*, 2010. Submitted.
6. J. Champeau and E. Rochefort. Model Engineering and Traceability. *UML 2003 SIVOES-MDA Workshop*, 2003.
7. J. de Lara and H. Vangheluwe. Translating model simulators to analysis models. In J. L. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *LNCS*, pages 77–92. Springer, 2008.
8. N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. Engineering a DSL for Software Traceability. pages 151–167, 2009.
9. H. Ehrig, G. Engels, and H. J. Kreowski. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. World Scientific Publishing Company, 1997.
10. H. Ehrig and C. Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *ICGT*, volume 5214 of *LNCS*, pages 194–210. Springer, 2008.
11. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In A. Evans, S. Kent, and B. Selic, editors, *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.
12. S. Esmaeilsabzali and N. A. Day. Prescriptive semantics for big-step modelling languages. In D. S. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Proceedings*, volume 6013 of *LNCS*, pages 158–172. Springer, 2010.
13. A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering Behavioral Design Models from Execution Traces. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 112–121, Washington, DC, USA, 2005. IEEE Computer Society.
14. N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *Services and Formal Methods, Forth International Workshop, WS-FM 2007*, pages 28–29.
15. J. Moe and D. A. Carr. Understanding Distributed Systems via Execution Trace Data. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, page 60, Washington, DC, USA, 2001. IEEE Computer Society.

16. OASIS. Web services business process execution language version 2.0 (OASIS standard), 2007. "http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html".
17. Object Management Group. Knowledge Discovery Metamodel: KVM Version 1.1, January 2009. http://www.omg.org/spec/KDM/1.1/.
18. L. Pedro, L. Lucio, and D. Buchs. System Prototype and Verification Using Metamodel-Based Transformations. *IEEE Distributed Systems Online*, 8(4):1, 2007.
19. E. Putrycz. Using trace analysis for improving performance in COTS systems. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 68–80. IBM Press, 2004.
20. I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In *Theory and Practice of Model Transformations*, volume 5063/2008 of *LNCS*, pages 107–121. Springer Berlin / Heidelberg.
21. I. Ráth, D. Vágó, and D. Varró. Design-time Simulation of Domain-specific Models By Incremental Pattern Matching. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2008.
22. I. Ráth, G. Varró, and D. Varró. Change-driven model transformations. In *Proc. of MODELS'09, CM/IEEE 12th International Conference On Model Driven Engineering Languages And Systems*, 2009.
23. M. Sela, Aviad ands Fritzsche, A. Zherebtsov, J. Johannes, and A. Terekhov. MODELPLEX Deliverable D4.2a: Metamodels for simulation. Technical report, IBM, Decembre 2007.
24. S. M. A. Shah, K. Anastasakis, and B. Bordbar. From UML to Alloy and back again. In *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.
25. M. Soden and H. Eichler. Towards a model execution framework for Eclipse. In *BM-MDA '09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 1–7, New York, NY, USA, 2009. ACM.
26. C. Stahl. A Petri Net Semantics for BPEL. Technical Report 188, Humboldt-U. zu Berlin, Institut für Informatik, 2005.
27. K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting Sequence Diagram from Execution Trace of Java Program. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 148–154, Washington, DC, USA, 2005. IEEE Computer Society.
28. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.
29. S. Walderhaug, U. Johansen, E. Stav, and J. Aagedal. Towards a Generic Solution for Traceability in MDD, 2006.