Abstract
Back-end analysis tools aiming to carry out model-based verification and validation of dynamic behavioral models frequently produce sequences of simulation steps (called execution traces) as their output. In order to support back-annotation of such traces, we need to store and replay them within a modeling environment (outside the analysis tool). In the paper, we present a technique for replaying recorded execution traces of dynamic modeling languages. Our approach complements static and dynamic metamodels by introducing a generic execution trace metamodel which is used to replay completed executions of a simulation directly over the dynamic model. Furthermore, we present a technique to drive a simulation according to execution trace models. Our approach will be exemplified by the modeling language and trace information of the SAL model checker and BPEL business processes.

Keywords
execution traces · simulation · dynamic modeling languages

1 Introduction
Model-driven analysis aims at revealing conceptual flaws early in the design process. In the typical approach, high-level design models (UML [31], BPEL [28], SysML [30], etc.) are automatically transformed into mathematical models (e.g. Petri nets [36], transition systems [49], process algebras [21]) to carry out system analysis by formal methods. The results of the analysis are then back-annotated to the original source model to highlight flaws directly in the design models.

In case of dynamic modeling languages (e.g. statecharts, workflows, live sequence charts [26]), the back-end formal analysis tools frequently carry out simulation or model checking to ensure the functional correctness of the design using analysis models like Petri nets, process algebras or labeled transition systems. As a result, back-end analysis tools produce an execution trace of the system as a designated or counter example.

However, in order to support the back-annotation of a complex counter example generated by an analysis tool, the corresponding execution trace needs to be replayed within a modeling environment (like Eclipse). Unfortunately, each back-end analysis tool uses a different, tool-specific textual trace representation, which requires a significant development effort for trace integration.

In the paper, we provide a generic replay mechanism for execution traces in dynamic modeling languages with a specific focus on those traces created by model checkers and simulation tools. We assume that a dynamic modeling language is defined by a combination of static, dynamic and (execution) trace metamodels while the availability of precise operational semantics is not required. This metamodeling approach was first introduced in the book presenting the results of the Sensoria project [16]. In the current paper, we extend upon the concept of a generic execution trace metamodel [19] and define high-level and elementary operations to support the replay of such traces within a general purpose modeling environment (i.e. outside the original analysis tool).

Our techniques will be first exemplified on the language and execution traces of the SAL model checker [5] then we show how the same technique can be applied to replaying execution...
of BPEL business processes (first demonstrated as a tool [20]).

The paper is structured as follows. First, related work is discussed in Sec. 2 and we give a conceptual overview of our approach in Sec. 3. Sec. 4 provides a brief introduction to the language of the SAL model checker and to static, dynamic and execution traces metamodels. Sec. 5 discusses how an execution trace model can be replayed to update the dynamic model. Sec. 6 illustrates the approach on BPEL processes, while Sec. 7 lists limitations. Finally, Sec. 8 concludes our paper.

2 Related work

Traces have been extensively researched in previous years as a means to represent and store information regarding (i) the dynamic behavior of a system or (ii) correspondences between models. To separate the models of these significantly different concerns, we refer to execution traces in the first case and traceability connections in the second case. Note that the current paper focuses execution traces and their replaying, therefore related work regarding traceability is not detailed. Approaches regarding traceability models [11, 39, 48] generally define static traceability models which record the correspondence between various model structures and suggest techniques, methods and tools for generating, managing or processing such models.

Problem-specific execution traces Execution traces are used in many cases, for understanding distributed systems [27], recovering behavior [17] and improving performance [35]. Dynamic traces were defined for individual languages such as UML sequence diagrams [44], UML Activity Diagrams [33], Concurrent Object-Oriented Petri Nets [32]. These approaches are usually developed for a single language or system and offer detailed representation and generation capabilities. Since they are highly specialized for a given domain, it would be difficult to apply them to a different domain. In the current paper, we define a generic, domain-independent representation for execution traces and a replaying framework for traces stored in this representation.

Recording and visualizing execution traces M3Actions [31] is a framework to develop execution semantics for MOF metamodels. It consists of a graphical editor for defining the structure and behavior of models, a generic interpreter and debugger for executing them and a trace recorder for storing executions. The framework focuses on support for modeling operational semantics and the recorded traces are low-level.

Traviando [23] is a tool package for analyzing and visualizing traces exported from a number of supported tools (e.g. Möbius). It supports model checking (using LTL properties) on imported traces and is able to display traces as Message Sequence charts or a tree-type visualization for investigating state information. Contrary to our method, this tool represent traces as simple sequences (as opposed to our hierarchical approach) and does not contain any replay capabilities.

Harel [26] represent traces for state-based models and reactive systems as scenarios which include atomic model changes similarly to delta steps in the generic trace metamodel described in the current paper. It also supports generation, analysis, visualization and interaction through the live sequence chart formalism. The approach focuses on reactive systems and their execution traces, while in the current paper we focus on dynamic modeling languages and an alternative approach to generic replaying of traces.

A recent approach [2] builds on the Metaviz trace visualization framework to provide model-based definition on creating high-level views from complex execution traces created during validation. The main motivation for the approach is to improve the practical usage of model validation tools.

It is common in these approaches that they focus on recording and visualizing runtime information of programs or dynamic models into execution traces, while in the current paper we use existing trace models to replay the dynamic behavior of models. Furthermore, traces recorded by these methods could be mapped into our generic trace metamodel thus adding trace replaying to their capabilities.

Metamodels for execution traces Alawneh [3] introduces metamodels for execution traces (as a standalone domain) to record runtime information of program executions. They propose to build the metamodel on KDM [29] and identify several trace types on the programming language level. Similarly to this approach, we argue for a metamodel for execution traces to represent the dynamic behavior of modeling languages.

The objective of [38] is to define a Tool-Independent Performance Model for mapping design and architectural models to performance models (used for design-time analysis of system performance). The introduced workbench is designed to include simulation and analysis capabilities and to derive execution sequences (scenarios) from UML activity diagrams for driving the simulation. This approach also shows that it is important to introduce a generic method that is usable for a particular task (e.g. performance analysis) with different domains. We describe a similar technique using a generic trace replaying framework for dynamic modeling languages.

The main contribution of our approach in comparison to existing work is that the proposed execution trace models are independent from the underlying simulation tool. Therefore, the execution of the analysis or simulation that creates traces can be completely separated from processing and evaluating these traces. Furthermore, persisted execution traces can be replayed in a modeling environment without using (external) simulators and model checkers.

3 Execution Traces in DMLs

Our overall goal is to provide a generic framework for replaying an execution trace, generated by a back-end analysis tool, within a general modeling framework (e.g. EMF [45] or VIATRA2 [47]). The replay mechanism is generic enough to be reusable and easily adaptable for various discrete event-based dynamic modeling languages (DML) used in analysis tools. The
trace replaying framework would also significantly reduce the cost of back-annotation for different pairs of source and target languages as demonstrated in our previous papers [16][19].

Metamodels for dynamic languages In our framework, we assume the existence of various metamodels in the context of a DML, which are exemplified in Fig. 1.

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 ($M_{stat}$).

Next, a dynamic metamodel $MM_{dyn}$ uses and extends the static meta-model $MM_{stat}$ for storing information related to the dynamic behavior (e.g. current state, value, configuration) of a structural element. The dynamic model ($M_{dyn}$) is an instance of $MM_{dyn}$.

This way, a trace metamodel ($MM_{trc}$) is defined for the language to represent simulation executions of $M_{dyn}$. $MM_{trc}$ uses $MM_{stat}$ for recording how the dynamic model changed and $MM_{stat}$ for describing which static element is concerned. A trace model ($M_{trc}$) is an instance of $MM_{trc}$, e.g. the sequence of execution steps.

Operational semantics for dynamic models The simulation of a DML is performed in accordance to the operational semantics of the language, defined by simulation rules. In our framework we assume that simulation rules are defined as intra-model transformations (see also [9][13][34]).

The execution of a rule $r$ in the transformation $MT_{sym} : (M_{stat} , M_{dyn}) \rightarrow M'_{dyn}$ modifies $M_{dyn}$ by also taking into account $M_{stat}$ and results in a new $M'_{dyn}$ as illustrated in Fig. 1. During a simulation execution, the changes of the dynamic model are recorded as a sequence of execution steps as part of the derived trace model $M_{trc}$. Furthermore, the complex manipulation steps in $M_{trc}$ are in direct correspondence to the transformation rules fired during the simulation execution.

Replaying execution traces of dynamic languages In our proposed framework, the execution traces of analysis models are persisted in a modeling environment using the output generated by back-end simulator or model checker tools (see Fig. 2). The model $M_{trc}$ can be used to replay the execution of a specific simulation execution.

The execution of step $s$, in the trace replaying transformation $MT_{rep} : (M_{dyn}, M_{trc}) \rightarrow M_{dyn}$ modifies the $M_{dyn}$, after which the model state ($M'_{dyn}$) will be the same as after the execution of a simulation rule $r$. The persisted traces can be replayed in the modeling environment using generic replaying operations through a trace manipulation interface. However, the main advantage of providing trace replay functionality appears when analysis traces are back-annotated into a source (design or engineering) model where a simulator may not be available. The back-annotated trace can also be replayed by the same generic replay framework.

In the current paper we exclusively focus on replaying simulation traces persisted as trace models, while the back-annotation of execution traces is discussed in our other papers [18][19].

4 Definition of Dynamic Modeling Languages

We provide a brief introduction to the language of the SAL model checker, which serves as the running example of the paper (Sec. 4.1). Then we discuss how dynamic SAL models can be integrated in a modeling framework using dynamic metamodeling [14] techniques (Sec. 4.2). Finally, we specify an execution trace metamodel (Sec. 4.3).

4.1 The SAL language

Symbolic Analysis Laboratory (SAL) [5] 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. Fig. 3 shows a simplified $MM_{stat}$ and $MM_{dyn}$ for SAL on the left and an example SAL system (in the textual syntax) on the right.

The SAL structure (Static Metamodel) 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 (see lines 2-3). The specification of a SAL module consists of state variable declarations (see lines 5-6), variable initializations and the transitions part. The state of the system model is defined by the current value of the 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 refers to the new value of variable $x$ in a transition. The initialization of variables (see line 8) is given as a combination of definitions [5]. Transitions are guarded commands defined in the form $g \rightarrow S$ where $g$ is a boolean guard (see line 10) and $S$ is a list of definitions (assignments, see line 11).

The SAL Dynamic Metamodel 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 referenced variables. In the metamodel we define 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),

Replaying Execution Trace Models for Dynamic Modeling Languages
4.2 Dynamic metamodeling for behavioral models

Dynamic metamodeling (DMM) [14] aims at specifying the dynamic behavior of executable modeling languages by combining metamodeling with rule based formalisms to capture operational semantics. In DMM, 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 formalized by graph transformation (GT) techniques [12].

In GT, **graph patterns** [46, p. 218] represent conditions that have to be fulfilled by a part of the model, this part is called a **match**. **GT rules** are specified by a **precondition** (or left-hand side - LHS) pattern determining the applicability of the rule and a **postcondition** (or right-hand side - RHS) pattern that specifies the result model declaratively. In the paper, we use the transformation language of **ViATRA2** [46] which essentially follows the single-pushout approach with injective matches.

The applicability of each GT rule is first checked by graph pattern matching techniques. Then a rule is applied for a selected match (if any exists), which updates the underlying $M_{dyn}$ to result in a new (dynamic) state. This selection can be non-deterministic or user-driven. Simulation rules can be fired as long as an enabled rule is found. This form of simulation is widely used in graph transformation tools (such as **Agda** [43], **AToM3** [8], **VMTS** [24], or **ViATRA2** [15]).

**Simulation rule example** The dynamic metamodeling is illustrated by describing the semantics for transition systems of SAL using graph transformation rules. The execution of a command can be defined in a transformation rule using the transformation language of the **ViATRA2** framework (left part of Fig. 4) based on the semantics of the SAL system when firing a guarded command. The right part shows a graph transformation rule for applying an assignment definition.

First, one command $Cmd$ is chosen non-deterministically from the enabled commands (where pattern matching returns a match). Then, all the assignments $Asnt$ of $Cmd$ are enumerated (as defined by all matches of $CmdsAsnt$ pattern) by modifying the current value relation of variables to the state defined by $Asnt$. The $applyAsnt$ transformation rule (right part of Fig. 4) is applied on a match of the LHS pattern and changes the target of the current relation of the corresponding $Variable$, as defined by the RHS pattern.

4.3 Execution Trace Models

An execution trace model captures the changes between two subsequent states of $M_{dyn}$. This way, the execution trace metamodel (see left part of Fig. 5) complements the existing $MM_{stat}$ and $MM_{dyn}$ as well.
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 change that occurred). The first relation defines the beginning of the trace (wrt. a specific execution).

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 defines an ordering between the steps represented by the next relation (where the source step precedes the target in the trace).

Traces created by various back-end analysis tools are frequently organized into a step hierarchy. As a consequence, we distinguish between CompoundSteps, which represent complex model manipulations and contain further steps (as represented by substep aggregation) and SimpleSteps representing elementary changes (i.e., the dynamic state before and after the modification denoted by the old value and new value relations, respectively) specific to a certain model element in $M_{dyn}$ (called the scope of the step) as recorded by the model checker or simulator in an execution trace. This representation is similar to change operations used in change-driven model transformations [6, 35].

Dynamic model elements The relations existing between the execution trace metamodel and the dynamic execution model have two kinds of targets. Either they are elements of the dynamic model, or values which may be either model elements or attributes (e.g., string, integer, boolean, double, float).

Trace model example A concrete trace model instance is shown in the right part of Fig. [5]. The selA_fired compound step contains the atomic step thread_work which has variable TVS as a scope, and process and work as old and new values.

The trace metamodel in Fig. [5] was derived based on our investigation of the following analysis tools: GROOVE [37], SPIN [22], UPPAAL [4], INA [42], SAL [40], Möbius [10], and LTSA [25]. Each tool has either simulation or verification capabilities that provide execution traces. We also examined the BPEL Designer [1] as a design tool and explored other languages (e.g., UML statecharts).

4.3.1 Trace model level of detail

In the generic trace replaying framework, trace models store each atomic model manipulation in order to include all required information to replay the execution trace without the original analysis tool or simulator. Thus, it is possible to replay traces of dynamic modeling languages where precise operational semantics are not available. For example, the execution trace models of such languages can be generated by model transformations using traces created by formal analysis or simulation of an other language [19].

Note that an execution trace could be replayed without storing atomic modifications if the executed simulation rule is identifiable and its internal behavior is completely determined by the input parameters. However, there are languages that do not meet this criteria. For example, the simulation rule may include random choices and variable value assignments depending on the exact environment of the tool (e.g., current time). In such cases it is insufficient to store only the executed rule and the parameters to generate a replayable trace and each atomic model manipulation should be recorded instead. However, as in the case of SAL, the stored trace model can contain the information about the executed rules in addition to the atomic model manip-
4.3.2 Extendible trace metamodel

The presented generic trace metamodel is able to store execution traces of discrete event dynamic modeling languages, where the simulation primarily alters parts of the dynamic model. However, some languages include (a) additional model manipulations during simulation, for example model elements may be created or deleted during the execution or (b) timing characteristics which should be taken into consideration during replay (e.g. for animation).

In order to support such languages additional extensions can be easily incorporated into the generic replay framework by (1) specializing the types of the metamodel (e.g. Step, SimpleStep or CompoundStep), (2) defining the necessary attributes and relations for such specialized types and finally, (3) providing specific handlers for these step types to be used by the framework during replaying.

Actually, for supporting element creation and deletion, change operations \[35\] can be used as special SimpleSteps and change commands \[6\] as special CompoundSteps. For supporting timing, it is possible to add timing related attributes to the Step type both for representing the exact time of the model manipulation (i.e. a timestamp) and the duration of the simulation rule.

5 Replaying Execution Trace Models

Execution trace models record scenarios generated by an execution 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 define an approach for replaying persisted execution traces directly over the dynamic model, without relying on simulation rules (e.g. Fig. 4). Existing simulators of dynamic languages use dedicated, tool-specific support for replaying traces and they are implemented as closed technology. Furthermore, many dynamic design languages completely lack simulator support.

Therefore, we decided to make two general assumptions on supported dynamic modeling languages when specifying our replaying approach. Trace replaying has to be feasible for languages that (1) have no operational semantics (simulation rules) specified or (2) the existing simulation tools cannot be modified to support replaying.

In this general 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 propose a simple interface providing an informal description on basic operations to drive the replay of execution trace models within the modeling framework (Sec. 5.1). Next, we precisely specify these operations using graph patterns and transformation rules (Sec. 5.2). Then, we illustrate the application of our approach on SAL traces (Sec. 5.3). Finally, we give a short description of the implemented replaying tool (Sec. 5.4).

5.1 Overview of trace replaying interface

We informally describe the main tasks carried out by (1) complex interface operations for traces, which are assembled from (2) elementary trace manipulation operations. Operations of the trace manipulation interface are then specified by graph patterns and GT rules over the generic execution trace model.

Interface for trace replaying The trace replay interface contains four 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 operation finds the last executed step in the trace and if there exists a next step then it is processed and every modification represented by substeps is carried out on the dynamic execution model.

Step backward One of the advantages of the execution trace model is the ability to navigate in either direction along the execution. This operation can be used to revert the modifications on the dynamic model by retrieving the last executed step and the processing its substeps (using the old values).

Jump to start This operation 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 dynamic model or (2) storing the initial state in the first step.

Jump to end This operation can be used to reach the last step of the trace without stepping through them all. It is advantageous when a recorded simulation execution is continued from a state persisted earlier in a trace.

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. In fact, these operations resemble the debugging interface of the Eclipse framework (e.g. Step Over, Step Into, Step Return) in that it is possible to navigate in the replaying without additional instrumentation.

Elementary trace manipulation operations In order to provide these high-level user interface operations, elementary operations (listed in Table 1) are also defined to manipulate and traverse execution trace models. To increase generality, these operations are defined directly over the generic trace metamodel.

5.2 Specification of 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 can be specified by graph patterns and GT rules in VIATRA2.

![Next step pattern](image1.png) ![Unfold step pattern](image2.png)

(a) Next step pattern (b) Unfold step pattern

Fig. 6. Horizontal and vertical traversal

**Horizontal traversal of a trace** We define graph patterns for traversing the trace on a given hierarchy level. Fig. 6a shows the pattern `nextStepInTrace` for finding the next step S2 following the last executed step S1 in the trace T.

**Vertical traversal of a trace** The substeps of a step are processed in order when traversing the trace vertically. Fig. 6b shows the graph pattern that searches for substeps in a higher-level Step. When looking for the first substep, a negative application condition pattern is used to ensure that the selected substep SS has no preceding step BSS. Otherwise, the second pattern is used to find the next substep from a given step LastSS.

**Step forward** Fig. 8 shows the generic implementation of the forward stepping function defined as abstract state machines in VIATRA2 transformation language. First, the Step following the last executed step of the trace is found. Then the last relation is updated to record forward stepping in the trace. Next the substeps of Step are processed in order and executed.

**Executing steps** The simple steps refer to a model element and a value corresponding to the element. The right part of Fig. 7 shows the graph pattern defined for retrieving this information from the persisted Step. When executing a step (see left part of Fig. 7), the action depends on the type of the Step. Compound steps are unfolded and their substeps are executed in order. Simple steps are executed by first retrieving the scope Sc and value V elements from the Step and the relation between them from the model (CRel). Then the target of the relation is replaced with the value persisted in the step. Note that if the executed step should be handled by a domain-specific extension (see Sec. 4.3.2) then the appropriate handler StepTypeHandler is called first.

5.3 Execution trace replaying example

We use our example SAL transition system (see Fig. 3) to illustrate the replaying of a persisted execution trace (see Fig. 5) with the defined generic operations.

The top part of Fig. 9 demonstrates how the execution trace model is used for stepping forward (imitating the execution of a guarded command) and how a simple step is executed by modifying the dynamic model (bottom part).

When stepping forward in the trace, the framework selects the next compound step `finish_executes` to execute, since the last processed compound step in the trace was `selectA_executes` (represented by the last relation) that has a `next` relation targeting `finish_executes`. During the application of rule StepForward, the substeps of the step are executed and the `last` relation is set to step `finish_executes`.

The execution of the SimpleStep `thread_work` is performed by finding the current value of the corresponding variable state TVS, and updating it in the dynamic model. The new value for TVS is selected by navigating through the `new` relation of step `thread_work`.

5.4 Implementation

The metamodels for the SAL language, as well as the trace generator and replay transformations are implemented in the VIATRA2 model transformation framework, which also supports the development and execution of simulation rules. VIATRA2 uses textual languages for defining both metamodels and transformations, thus their complexity can be illustrated with the number of lines for each definition. The static metamodel of SAL is over 1000 lines of code (LOC) and includes over 100 elements each with several relations, while both the SAL dynamic metamodel and the generic trace metamodel are under 100 LOC defining around 20 elements and relations. The SAL trace generator transformation processes a text-based trace is around 1000 LOC with 38 patterns and 11 complex rules, while the replay
transformation is a few hundred LOC with around 20 patterns and 10 rules.

We also developed a tool for importing counter-examples of the SAL model checker to trace models in the ViTrax2 framework. Furthermore, we used the proposed approach for replaying execution traces of Petri Nets as well.

The trace metamodel is designed to allow the implementation of a trace replaying transformation that requires only neighboring steps at a given time (due to persisting both old and new values of a model element). Therefore, replaying is independent of the size of traces (which can be well over 100 steps).

6 Replaying BPEL business process execution

The trace replay framework is mainly a generic tool for replaying execution traces that were originally recorded from analysis tools or simulation (see Sec. 5). However, it is also possible to replay traces for high-level design languages that lack formal semantics or simulation tools. In this section we describe how generic replaying was used for business processes defined in the Business Process Execution Language (BPEL) [28].

6.1 Execution traces for BPEL

In order to support the replaying of BPEL process executions with the proposed generic framework, we first have to define the dynamic metamodel for BPEL and show that the generic trace metamodel defined in Sec. 4.3 is capable of representing the execution traces of BPEL.

The complete static metamodel of BPEL contains a high number of types for different activities, events and information representation. For the purposes of the paper only a small fragment is relevant (illustrated in Fig. 7). Elements of the static metamodel are all specialized from ExtensibleElements with Process representing the business process itself containing an Activity. Activity types, among others, include Sequence and Receive. The process also contains Variables which are accessed and manipulated by activities.

In order to model process instances in execution we define additional dynamic information for BPEL elements. Activity State is associated with activities and has a current state. This state can be either startable, runs and executed for all activities, but further refinement is possible with additional states for complex structures (such as scopes). Similarly, Variable State is associated with variables and has a current state that can be uninitialized, correct and faulty.

A small BPEL execution trace model is shown in the right side of Fig. 1 where the first compound step is the start of the process (process_starts) and the second step is the execution of a receive activity (receive_runs). This step also includes a substep for setting the state of the input variable, from uninitialized to correct, representing the storing of the received message. Since the BPEL trace can be modeled using the generic execution trace metamodel (discussed in Sec. 4.3), the traces can be replayed in the proposed framework without any additional development effort.
Mapping non-sequential BPEL activities  The structural activities defined in the BPEL language often represent non-sequential execution where the control flow of different process instances can differ based on the particular execution. For example, a conditional decision may have multiple branches where the actually executed branch is selected based on the current value of the process variables. Similarly, a looping activity (e.g. the updateDesired? cycle in Fig. 12) can be executed more than once. However, during the execution of the BPEL process, the steps corresponding the execution of these structures will be sequential in the stored trace. Consider the updateDesired? cycle in the example, every time the cycleCore activity becomes executed, the condition is checked whether to make it startable again or change updateDesired? to executed. Finally, in case of parallel execution in a flow activity, the execution of the contained activities (e.g. the Balances and Security sequences) may overlap, but they can be represented as a sequence of simple steps as well. Details on how to handle overlapping and other mismatches between the granularity of BPEL and SAL traces can be found in our SEFM paper [19]. Thus, non-sequential execution is also mapped into sequential steps in the execution trace, where each step will have at most one corresponding next step. When such activities are present in the process during trace replaying, their activity state is set in the same way as done with sequential activities.

6.2 Graphical interface for replaying
We have created a graphical user interface in Eclipse to support the replaying of BPEL execution traces [20]. Fig. 11 shows the BPEL Animation Controller view, where execution traces can be opened (Load Trace), the textual file is processed, and the Viatra2 framework initializes the trace models. When the framework is ready, the navigation buttons can be used to animate the process execution. Apart from step-by-step navigation...
6.3 Visualization of dynamic state of BPEL processes

The generic replay framework works inside the model space of the Viatra2 framework. Since this representation makes it difficult to interpret BPEL traces, we also developed (see [20]) an intuitive graphical representation of execution trace replaying with a modified Eclipse BPEL Designer [1].

Fig. 12 shows the customized BPEL Designer at a given state during the trace replaying of an example BPEL process. The activities and variables of the process are colored depending on their current dynamic state. Thus the dynamic behavior of the BPEL process can be observed visually in the original design perspective used for developing BPEL processes. For the activities, light blue means startable state (e.g. addSecurityToRating), light green active (e.g. addBalanceToRating), dark green finished (e.g. Creation). For variables, yellow is uninitialized state (e.g. updateDesired), green is correct (e.g. loginData) and red is faulty.

6.4 Implementation

The execution trace of BPEL processes is created by mapping the counter-examples (traces) of the SAL model checking framework back to the context of BPEL [16]. This back-annotation transformation is part of a verification tool developed for BPEL processes using the SAL back-end tool.

The BPEL process executions can be replayed interactively using the Eclipse BPEL Designer, where the dynamic state of activities and variables are set using a service that is called by the replaying framework to export state changes for a given step and the exported state is processed by the Animation Controller.

7 Limitations of the trace replaying approach

Limitations of the approach The generic trace metamodel and replaying framework has many application possibilities, however certain limitations should be noted regarding its applicability to new DSMLs or tools.

- First, the dynamic metamodel of the DSML should represent state changes through relation or attribute manipulations in the model.
- Furthermore, integrating a new DSML (and its simulator) still requires some development effort even if the replaying is generic and the dynamic metamodel is suitable. This integration task mainly consists of creating an importer for the trace format for the given tool.
- Finally, since the trace replaying does not use the original tool that produced the original trace, the replayed execution will only represent the original at the level of detail stored in the trace.

Limitations of replaying BPEL executions The replaying of BPEL processes uses the generic trace replay framework, therefore it is limited by the factors described above. Additional limitations include:

- The traces are derived from SAL counter-examples generated through verification which only represents BPEL execution on a coarse level (i.e. simple activity states and non-interpreted variable values).
- Similarly, the trace generation options are limited as the SAL tool is not a simulator but a verification tool that produces counter-examples based on requirements.

8 Conclusion

In the paper, we investigated how execution traces retrieved by model checkers 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 replay traces by updating the underlying dynamic model. As a result, the generation and evaluation of traces can be completely separated and traces can be navigated without the use of external analysis tools.

Our generic execution trace model was actually defined based on our investigation of traces retrieved by various formal analysis tools (using different modeling formalisms such as Petri nets, transition systems or process algebras). Finally, we have illustrated by making use of a BPEL process that the replay framework can support high-level design languages as well.

Currently, as an ongoing work, we are investigating how trace generation transformations can be derived from simulator specifications. Furthermore, we plan to combine the generic trace replaying approach with design space exploration to support languages with non-deterministic simulation rules and limited execution trace generation capabilities.
Fig. 12. Animation of an execution trace

References
project on Software Engineering for Service-Oriented Computing, Springer-Verlag, 2011.


27 Moe J, Carr DA. Understanding Distributed Systems via Execution Trace Data, IWPC ’01: Proceedings of the 9th International Workshop on Program Comprehension, (2001), 60–.

28 (OASIS standard), OASIS. Web services business process execution language version 2.0, 2007. [http://docs.oasis-open.org/wsbpel/2.0/ wsbpel-v2.0.html]


42 Starke PH. Integrated net analyzer, 2003. [http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/]


