Trace Comprehension Operators for Executable DSLs

Dorian Leroy, Erwan Bousse, Anaël Megna, Benoit Combemale, Manuel Wimmer

Trace Comprehension Operators for Executable DSLs

Dorian Leroy[1], Erwan Bousse[2], Anaël Megna[3,4], Benoit Combemale[3], and Manuel Wimmer[2]

[1] JKU Linz, Linz, Austria
[2] TU Wien, Vienna, Austria
[3] University of Toulouse, CNRS IRIT, Toulouse, France
[4] Safran SA, Paris, France

Abstract. Recent approaches contribute facilities to breathe life into metamodels, thus making behavioral models directly *executable*. Such facilities are particularly helpful to better utilize a model over the time dimension, *e.g.,* for early validation and verification. However, when even a small change is made to the model, to the language definition (*e.g.,* semantic variation points), or to the external stimuli of an execution scenario, it remains difficult for a designer to grasp the impact of such a change on the resulting execution trace. This prevents accessible trade-off analysis and design-space exploration on behavioral models. In this paper, we propose a set of formally defined operators for analyzing execution traces. The operators include dynamic trace filtering, trace comparison with *diff* computation and visualization, and graph-based view extraction to analyze cycles. The operators are applied and validated on a demonstrative example that highlight their usefulness for the comprehension specific aspects of the underlying traces.

Keywords: Model Execution; Domain-Specific Languages; Executable DSL; Execution Trace; Trace Analysis

1 Introduction

A large amount of DSLs are used to represent behavioral aspects of systems in the form of *behavioral models* (*e.g.,* [1,2,3,4,5]). To better appreciate how such models unfold over the time dimension, a lot of efforts have been made to facilitate the design of so-called *executable DSLs* (*e.g.,* [6,7,8,9,10,11,12]), which enable the execution of conforming models using *execution semantics*. Two approaches are commonly used to define the execution semantics of an executable DSL, namely operational semantics (*i.e.,* interpretation) and translational semantics (*i.e.,* compilation). We focus in this paper on executable DSLs defined with operational semantics, and more precisely with discrete-event operational semantics.

Executing a model gives the possibility to observe the evolution of its *state* over time, *i.e.,* the *trace* of the execution [13]. Once an execution trace has been captured (*e.g.,* by instrumenting the model interpreter), it can be exploited in

several development contexts, such as providing prompt feedback to the modeler, understanding the fault revealed by a failed test case, or performing complex automated dynamic analyses of the considered traces. In particular, in the context of design-space exploration, *trade-off analyses* of different design choices can be achieved by comparing the traces resulting from executing different variants of the model. This is especially important as even the smallest change in the model (*e.g.,* changing a guard on a transition), in the considered execution scenario (*e.g.,* exchanging the order of two signals sent to an UML state machine), or in the language definition (*e.g.,* semantic variation points) can lead to a completely different execution trace.

However, it remains difficult for a modeler to grasp the impact of a design change on the resulting execution trace. Comparing execution traces is often hampered by noisy or redundant data captured in execution traces, which leads to finding irrelevant differences between the traces. In particular, it is often compulsory to filter out extraneous dynamic information from an execution trace, in order to focus only on the changes occurring in a relevant subset of the model state. In addition, due to the sequential nature of a trace, it is laborious to visualize which model states were explored multiple times during the execution of a behavioral model, and between which states the model may have oscillated, *e.g.,* in order to discover potential cycles or bottlenecks.

To address these problems, we propose in this paper a set of formally defined *trace comprehension operators*. These operators can be used for dynamic information filtering, trace comparison with *diff* computation and visualization, and graph-based views extraction to analyze cycles. Some operators can be combined for better results, *e.g.,* to extract a graph-based view out of filtered traces. We provide a formalization of each operator, and we implemented them as part of the GEMOC Studio[5], an Eclipse-based language and modeling workbench. We validate the approach by demonstrating the relevance of the operators for model variants conforming to a State Machines DSL inspired by UML State Machines.

The remainder of the paper is structured as follows. Section 2 presents the scope of considered DSLs and execution traces, and a motivating example. Section 3 shows an overview of the proposed solution. Section 4 describes the formalization of the trace comprehension operators. Section 5 presents the implementation and the validation of our approach with a use case based on a State Machine DSL. Section 6 presents related work. Finally, Section 7 concludes.

2 Background and Motivating Example

In this section, we first precisely scope the executable DSLs considered in our approach, *i.e.,* metamodel-based DSLs with discrete-event operational semantics. We then present the considered execution traces, and finally we present a motivating example based on a State Machines DSL.

[5] https://eclipse.org/gemoc

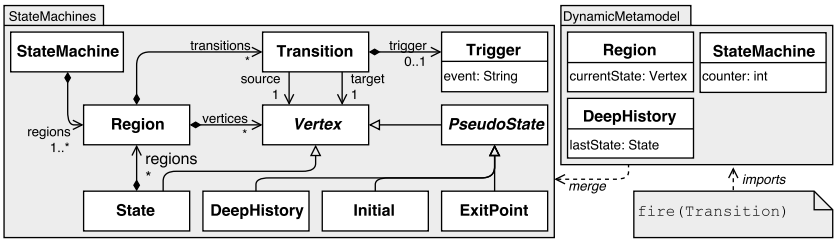


Fig. 1: The State Machines executable DSL.

2.1 Considered Executable DSLs

An executable DSL is composed of both an abstract syntax defining the concepts of the domain, and an execution semantics defining how these concepts are executed. In this paper, we focus on executable DSLs whose abstract syntax is a *metamodel*, and whose execution semantics is an *operational semantics*. A metamodel is a class-based object-oriented model defined using a metamodeling language (*e.g.,* MOF [14] or Ecore [15]) composed of a set of metaclasses. A metaclass is composed of a set of *properties*, each either an attribute (typed by a datatype, *e.g.,* , integer) or a reference to another metaclass.

The left part of Fig. 1 shows the abstract syntax of an example of State Machines DSL, which is directly inspired from UML State Machines. A StateMachine contains at least one Region. A Region contains Vertex elements, which can be State elements or PseudoState elements. PseudoState elements are further refined into Initial, ExitPoint and DeepHistory elements. Finally, a Region also contains Transition elements which point to a `source` and a `target` Vertex. Transition elements contain Trigger elements, which may each possess an `event`.

Next, we decompose the *operational semantics* of an executable DSL in two parts: a data structure representing the state of the executed model, and a model transformation altering the model state. We define this data structure as a metamodel which extends the abstract syntax with new dynamic metaclasses and properties using *package merge*. Executing the model consists in applying the model transformation of the semantics, which performs an endogenous, possibly in-place, transformation on the model state. We do not make assumptions on the language used to define the model transformation, nor on the content of the transformation. Instead, we only consider that it produces a sequence of changes in the state of the model, each change caused by a given *observable event* (*e.g.,* a rule call for rule-based languages, or a method call for imperative languages).

The upper right part of Fig. 1 shows the metamodel of the operational semantics extending the abstract syntax with new dynamic properties: the `currentState` property in Region is used to track the current state of the Region during the execution, the `lastState` property in DeepHistory stores the last visited state in the owning Region element, and finally the `counter` counts the number of fired transitions during the execution. Finally, we consider that the

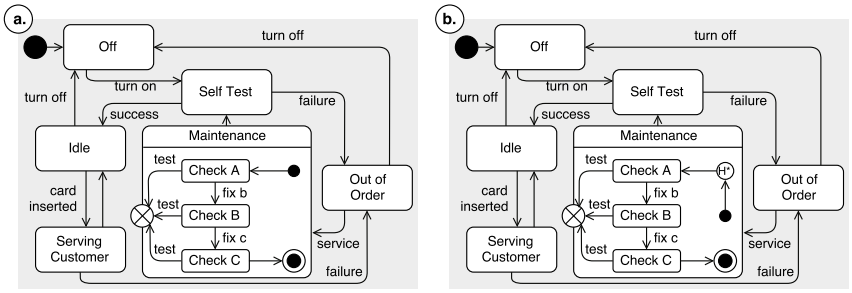


Fig. 2: Two ATM state machine variants: *a* without history and *b* with history.

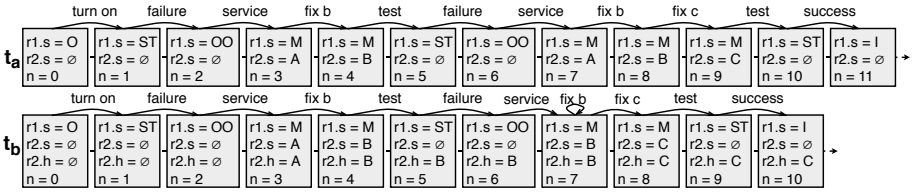


Fig. 3: Traces from the example models: *a* is without history, *b* is with history.

only observable event of the DSL is the firing of Transition elements. Thus, the execution semantics only defines a `fire` event.

2.2 Considered Execution Traces

At runtime, an executed model is composed of a set of objects, each object being an instance of a metaclass of the DSL. An object assigns one value per property of the corresponding metaclass. A *model state* is a recording of all values assigned to dynamic properties—*i.e.,* the properties added by the operational semantics—of the executed model at a certain point of the execution.

We call *execution trace* a sequence containing all model states reached during an execution and all observed execution events. A trace is obtained by recording the execution of a behavioral model. In a trace, we call *dimension* the sequence of all values assigned to a specific dynamic property by an object over the execution. As our previous work [13,16], considering a trace as a set of dimensions is central to our approach, as it gives the possibility to efficiently manipulate the parts of a trace related to specific dynamic properties. We present examples of traces in the following subsection.

2.3 Motivating Example

Fig. 2 depicts two models conforming to the State Machines DSL shown in Fig. 1. Both models represent the behavior of a *cash dispenser*, also called *ATM*, with

states such as *Idle* or *Serving Customer*. Transitions represent how the ATM switches mostly between idling, maintenance and service states. The difference between the two models lies in the added deep history pseudostate in the region of the *Maintenance* state. The semantics of the deep history pseudostate is that it stores the last visited state of its containing region and, when targeted by a transition, restores this state as the current state of the region. Adding such a pseudostate can thus affect greatly how the execution unfolds, and predicting the impact of such a change can be difficult.

Fig. 3 shows two execution traces resulting from the execution of the models in Fig. 2 with the following sequence of stimuli: *turn on*, *failure*, *service*, *fix b*, *test*, *failure*, *service*, *fix b*, *fix c*, *test*, *success*. In these traces, `r1` refers to the Region element owned by the state machine and `r2` to the Region element owned by the *Maintenance* state. The `s` property refers to the current state of a Region element, and the `h` refers to the last state of a DeepHistory element. Finally, the `n` property refers to the counter of fired transitions of the state machine. The name of the states are abbreviated for space reasons.

By looking at the execution traces shown in Fig. 3, we can glimpse that if we were ignoring the dimensions `counter` and `lastState`, then many similarities between the two traces could be found. For instance, the states $\langle$*Maintenance*, *Check C*, 9$\rangle$ and $\langle$*Maintenance*, *Check C*, *Check C*, 8$\rangle$ would then be equivalent. Even in a single trace, the value of the `counter` is different in each model state, which make it hard to identify possible cycles encountered in the states of the State Machine. For instance, t_a features a cycle that can only be detected if the `counter` dimension is ignored: $\langle$*Maintenance*, *Check B*, 4$\rangle$ and $\langle$*Maintenance*, *Check B*, 8$\rangle$ are two states part of this cycle.

In summary, even with small models with little dynamic information, and with only small changes between model variants, it is already challenging to understand and to compare the resulting execution traces. A similar observation could be made if small changes were made to the semantics of the considered DSL, or to the stimuli of the considered execution scenario. In this context, to ease the task of understanding execution traces, our contribution is a set of four *trace comprehension operators*. We give a brief presentation of these operators in Section 3, before defining them formally in Section 4.

3 Approach Overview

In this section, we present an overview of the contribution of this paper, *i.e.,* four different and complementary trace comprehension operators. Fig. 4 summarizes the application context including the inputs and outputs of the different operators. On the left, two behavioral models named A and B are shown, and a trace is obtained for each of their executions. Then, four different operators can be used to manipulate the obtained traces:

– The *Filter* operator takes an execution trace as input and produces a refined version of the input execution trace as output. It removes a selected set of

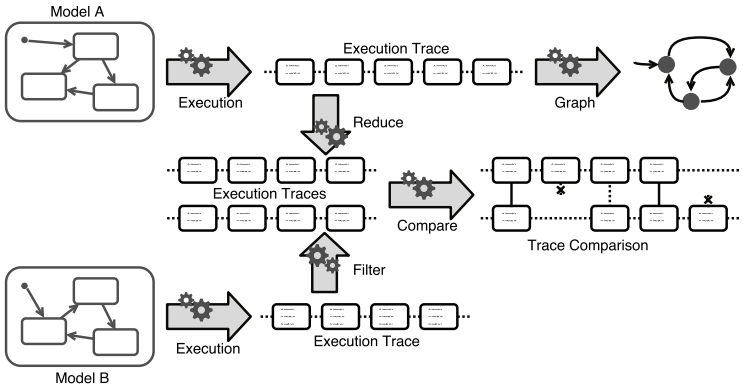
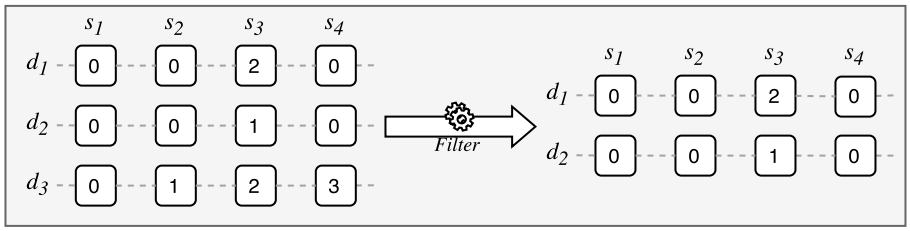


Fig. 4: Overview of a possible workflow using all four proposed operators.

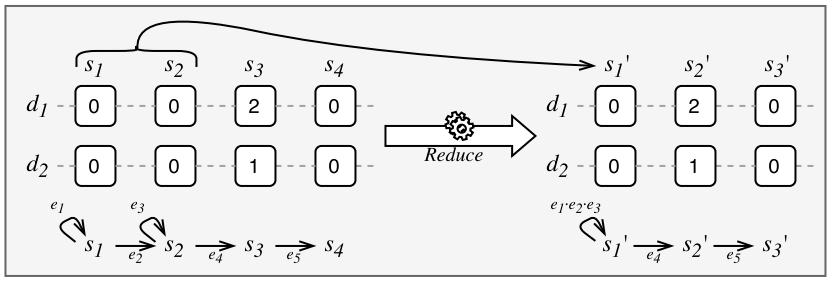
dimensions (see Section 2.2 for the definition of dimension) from the input trace, which results in a simplified trace that only reflects the evolution of a subset of the model state. Note that *Filter* does not change the amount of model states in the trace, and only changes the content of each model state.

- The *Reduce* operator also takes an execution trace as input and produces a refined version of the input execution trace as output, where each subsequence of successive identical model states is merged into a single model state. *Reduce* is particularly useful when applied after the *Filter* operator when the only differences between the states of a sequence of successive states were found in the dimensions that were filtered out.
- The *Compare* operator takes two execution traces as input and produces a trace difference model as output. This difference model highlights all the changes that occurred between the first trace and the second one: which states were added, or removed, or substituted by other states. Such comparison can be used to better understand the impact of a design change on the trace resulting from the execution.
- The *Graph* operator takes an execution trace as input and produces a state graph as output. This state graph is a representation of all different model states reached during the execution. Among other benefits, such higher-level view provides a better global understanding of the execution, and can highlight cycles and bottleneck states.

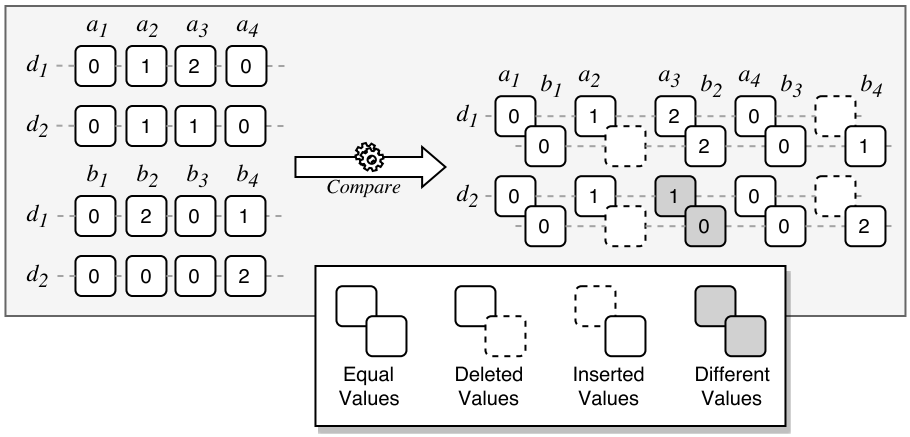
The middle and right part Fig. 4 show a typical workflow where the traces obtained from the models are simplified through the use of the *Filter* and *Reduce* operators, before being used as input for the *Graph* and *Compare* operators. In the following section, we provide a formal specification of these four operators.



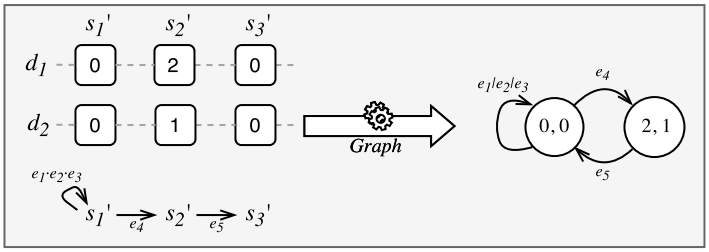
(a) *Filter* operator, which removes dimensions from a trace.



(b) *Reduct* operator, which factorizes redundant states.



(c) *Compare* operator, which produces a trace difference.



(d) *Graph* operator, which extracts a state graph from a trace.

Fig. 5: Graphical summary of all four trace comprehension operators.

4 Operators for Execution Trace Comprehension

In this section we present our contribution, *i.e.,* a set of four trace comprehension operators. Fig. 5 summarizes graphically all proposed operators using abstract examples, and will be used throughout the section to illustrate the operators. In what follows, we first formally define what is a execution trace, then we provide a formal definition of each trace comprehension operator.

4.1 Execution Trace Formalization

In order to give a formal definition of operators that manipulate execution traces, we must first formally define the concept of trace. In the remainder of the paper, we denote T the set of all execution traces.

Definition 1 (Trace)**.** A trace is a tuple $\langle S, D, E_<, val, step\rangle$ where:

- S is the set of model states of the execution trace.
- D is the set of dimensions of the execution trace.
- $E_< = (E, <_E)$ is the totally ordered set of events that occurred during the execution where, $\forall e_1, e_2 \in E, e_1 <_E e_2$ if e_1 happens before e_2.
- $val : (S \times D) \to V$ is the function mapping a model state and a dimension to a value. Using val, we define a state equivalence relation $Eq \subseteq S \times S$ as $(a, b) \in Eq \Leftrightarrow \forall d \in D, val(a, d) = val(b, d)$, denoted $a \equiv b$.
- $step : E_< \to (S \times S)$ is the function mapping an event to a starting an ending state. Note that an event *can* have the same starting and ending state, which means that the model state did not change due to the event occurrence. We denote:
  - $a \xrightarrow{e} b$ the fact that $step(e) = (a, b)$,
  - $a \xrightarrow{*} b$ the fact that *step* can lead from a to b with a sequence of events, *i.e.*:

$$\exists e \in E_<, a \xrightarrow{e} b \vee \exists n \in \mathbb{N}, \exists e_1, ..., e_n \in E_<, \exists s_1, ..., s_{n-1} \in S,$$
$$a \xrightarrow{e_1} s_1 \xrightarrow{e_2} ... \xrightarrow{e_{n-1}} s_{n-1} \xrightarrow{e_n} b \wedge \forall i \in]1; n], e_{i-1} <_E e_i,$$

  - $a \to b$ the fact that $a \neq b \wedge \exists e \in E_<, a \xrightarrow{e} b$, *i.e., a* directly precedes b,
- for any two states a and b, there is an ordered sequence of events that lead from a to b or from b to a, *i.e.,* $\forall a, b \in S, a \xrightarrow{*} b \vee b \xrightarrow{*} a$.
- the total order on events $<_E$ combined with the *step* function create a total order $<_S$ over the states defined as: $\forall a, b \in S, a <_S b \Leftrightarrow a \xrightarrow{*} b$ We denote $s = S_i$ the fact that $|\{s' \in S : s' <_S s\}| = i$

Example 1. Using only natural integer values (*i.e.,* $V = \mathbb{N}$), and events e_i ordered by their index i, let t_{ex} be an execution trace conforming to Definition 1:

$$t_{ex} = \langle \{s_1, s_2, s_3, s_4\}, \{d_1, d_2, d_3\}, \{e_1, e_2, e_3, e_4, e_5\}, val, step\rangle$$
$$\text{where } val(s_1, d_1) = 0 \quad val(s_2, d_1) = 0 \quad val(s_3, d_1) = 2 \quad val(s_4, d_1) = 0$$
$$val(s_1, d_2) = 0 \quad val(s_2, d_2) = 0 \quad val(s_3, d_2) = 1 \quad val(s_4, d_2) = 0$$
$$val(s_1, d_3) = 0 \quad val(s_2, d_3) = 1 \quad val(s_3, d_3) = 2 \quad val(s_4, d_3) = 3$$
$$\text{and } s_1 \xrightarrow{e_1} s_1 \quad s_1 \xrightarrow{e_2} s_2 \quad s_2 \xrightarrow{e_3} s_2 \quad s_2 \xrightarrow{e_4} s_3 \quad s_3 \xrightarrow{e_5} s_4$$

4.2 Dimension Filtering

When an operational semantics introduces a large amount of dynamic properties, or when the executed model is very large, an execution trace may contain a large amount of dimensions to grasp. Yet, understanding specific aspects of the behavior might only require looking of a specific subset of dimensions of interest. For this purpose, our first operator is called *Filter* (see Fig. 5a), and aims at removing dimensions out of a trace in order to simplify it. This operator is in fact an abstraction operator on the model states contained in the trace.

Definition 2 (*Filter*)**.** Given an input trace $\langle S, D, E_<, val, step\rangle$ and an input set of dimensions I, the *Filter* operator is defined as:

$$\begin{array}{rccc} Filter: & (T \times \mathcal{P}(D)) & \rightarrow & T \\ & (\langle S, D, E_<, val, step\rangle, I) & \mapsto & \langle S, D', E_<, val', step\rangle \end{array}$$

where $D' = D \setminus I$ and $val' : S \times D' \rightarrow V$ is defined as $val'(s, d') = val(s, d')$.

Example 2. We apply *Filter* to the trace t_{Ex} and dimension d_3 from Example 1:

$$Filter(t_{Ex}, \{d_3\}) = \langle \{s_1, s_2, s_3, s_4\}, \{d_1, d_2\}, \{e_1, e_2, e_3, e_4, e_5\}, val', step\rangle$$

$$\begin{array}{rl} \text{where } & val'(s_1, d_1) = 0 \quad val'(s_2, d_1) = 0 \quad val'(s_3, d_1) = 2 \quad val'(s_4, d_1) = 0 \\ & val'(s_1, d_2) = 0 \quad val'(s_2, d_2) = 0 \quad val'(s_3, d_2) = 1 \quad val'(s_4, d_2) = 0 \\ \text{and } & s_1 \xrightarrow{e_1} s_1 \quad s_1 \xrightarrow{e_2} s_2 \quad s_2 \xrightarrow{e_3} s_2 \quad s_2 \xrightarrow{e_4} s_3 \quad s_3 \xrightarrow{e_5} s_4 \end{array}$$

Note that $s_1 \equiv s_2$ and $s_1 \rightarrow s_2$, *i.e.,* two successive model states are identical. The next operator will enable the merging of these states to obtain a more compact trace, *i.e.,* where a state is always different from the preceding state.

4.3 Trace Reduction

When using a trace recorder that always records the model state at each occurring observable event without checking if the state has changed, or when using the *Filter* operator introduced above, a trace may contain successive equivalent states which are redundant and can be considered as superfluous data. This phenomenon is also known as stuttering [17]. To simplify such traces, we propose an operator *Reduce* (see Fig. 5b) which merges such successive equivalent states while preserving the behavior depicted by the trace.

Definition 3 (*Reduce*)**.** The *Reduce* operator is defined as:

$$\begin{array}{rccc} Reduce: & T & \rightarrow & T \\ & \langle S, D, E_<, val, step\rangle & \mapsto & \langle S', D, E_<, val', step'\rangle \end{array}$$

where:

– S' is the set of sets of successive equivalent states of S, *i.e*:

$$S' = \{s \in \mathcal{P}(S) : \forall a \in s, \forall b \in S, a \equiv b \wedge a \rightarrow b \Rightarrow b \in s\}$$

– $step' : E_< \rightarrow (S' \times S')$ is defined as: $step'(e) = \langle A, B\rangle \Leftrightarrow step(e) \in (A \times B)$
– $val' : (S' \times D) \rightarrow V$ is defined as $val'(B, d) = val(a, d)$ for any $a \in B$

Hence, each output state of S' is composed of (and thus replaces) a set of equivalent successive states of S, and both $step'$ and val' are adjusted accordingly.

Example 3. Resulting trace from $Reduce(Filter(T_{Ex}, \{d_3\}))$.

$$Reduce(Filter(T_{Ex})) =$$
$$\langle \{s'_1 = \{s_1, s_2\}, s'_2 = \{s_3\}, s'_3 = \{s_4\}\}, \{d_1, d_2\}, \{e_1, e_2, e_3, e_4, e_5\}, val', step'\rangle$$
$$\text{where } val'(s'_1, d_1) = 0 \quad val'(s'_2, d_1) = 2 \quad val'(s'_3, d_1) = 0$$
$$val'(s'_1, d_2) = 0 \quad val'(s'_2, d_2) = 1 \quad val'(s'_3, d_2) = 0$$
$$\text{and } s'_1 \xrightarrow{e_1} s'_1 \quad s'_1 \xrightarrow{e_2} s'_1 \quad s'_1 \xrightarrow{e_3} s'_1 \quad s'_1 \xrightarrow{e_4} s'_2 \quad s'_2 \xrightarrow{e_5} s'_3$$

The soundness of *Reduce* can easily be proven, *i.e.,* the fact that two successive states of S' cannot be equivalent, and that one state of S is only mapped to a single state of S'. These properties can be rephrased as two theorems:

Theorem 1. $Reduce(T) = \langle S', \_, \_, \_, \_\rangle \Rightarrow \forall s_1, s_2 \in S', s_1 \rightarrow s_2 \Rightarrow s_1 \not\equiv s_2$

Theorem 2. $Reduce(T) = \langle S', \_, \_, \_, \_\rangle \Rightarrow \bigcap_{s \in S'} s = \emptyset$

4.4 Trace Comparison

As understanding a single execution trace is already a difficult task, grasping the differences between two execution traces is even more challenging and error-prone. To address this problem, we propose a *Compare* operator that produces a *trace difference* showing the similarities and dissimilarities between two traces. Note that since traces may come from different models (*e.g.,* an original and a revised one), each trace may possess its own set of dimensions, hence *Compare* requires an explicit mapping between the dimensions of the first and second traces.

Our comparison procedure relies on the notorious *Levenshtein distance* [18], which is an operator counting the minimal number of insertion, deletion or substitution operations required to transform one string into another. For instance, the Levenshtein distance between "*S*TR**ING**" and "TR**A**C**E**" is four, which is computed by summing the number of insertions in *italics* and of substitutions in **bold**. While the output of the *Levenshtein distance* is an integer, computing this distance requires computing all the distances between all the possible prefixes of the input strings (*i.e.,* substrings starting with the first character). It is then possible to infer from all these distances the exact set of insertions, deletions or substitutions required to transform the first string into the second string, which is the kind of information we require to construct a trace difference. For our work, we adapted the Levenstein distance to compare traces instead of strings, where model states play the role of characters, which can be compared using the equivalence relation.

Definition 4 (Levenshtein distance on traces)**.** The Levenshtein distance between two traces $T_1 = \langle A, \_, \_, \_, \_ \rangle$ and $T_2 = \langle B, \_, \_, \_, \_ \rangle$ is given by $lev_{T_1,T_2}(|A|, |B|)$ where:

$$lev_{T_1,T_2}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{T_1,T_2}(i-1,j) + 1 \\ lev_{T_1,T_2}(i,j-1) + 1 \\ lev_{T_1,T_2}(i-1,j-1) + 1_{A_i \not\equiv B_j} \end{cases} & \text{otherwise} \end{cases}$$

Where $1_{A_i \not\equiv B_j}$ equals 0 when $A_i \equiv B_j$, and equals 1 otherwise.

As we can see, to obtain the Levenstein distance $lev_{T_1,T_2}(|A|, |B|)$, we rely on a recursive operator $lev_{T_1,T_2}(i,j)$ which computes the distance between the subsequence of states $[0,i]$ of T_1 and the subsequence of states $[0,j]$ of T_2. These distances can be used to infer the insertions, deletions and substitutions required to go from the first trace to the second. In that goal, we define the following notations on top of *lev*:

- $in_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i,j-1) + 1$,
- $del_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i-1,j) + 1$,
- $subst_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i-1,j-1) + 1$.

Using this $lev_{T_1,T_2}(i,j)$ through these notations, we can define the *Diff* operator which produces a unique set containing states of T_1 that were deleted, states of T_2 that were inserted, and pairs of states from T_1 and T_2 that were substituted.

Definition 5 (*Diff*)**.** We define the union set of inserted, deleted and pairs of substituted states identified as part of a Levenshtein distance computation as $Diff_{T_1,T_2} = DiffRec_{T_1,T_2}(|A|, |B|)$, where:

$$DiffRec_{T_1,T_2}(i,j) = \begin{cases} DiffRec_{T_1,T_2}(0,0) = \emptyset & \\ DiffRec_{T_1,T_2}(i,j-1) \cup B_j & \text{if } in_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j) \cup A_i & \text{if } del_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j-1) \cup \{A_i, B_j\} & \text{if } subst_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j-1) & \text{otherwise} \end{cases}$$

Finally, we define *Compare* as a trivial projection of the output of the *Diff* operator into a tuple that separates insertions, deletions and substitutions in three different sets.

Definition 6 (*Compare*)**.** Given two traces $T_1 = \langle A, \_, D_1, \_, \_ \rangle$ and $T_2 = \langle B, \_, D_2, \_, \_ \rangle$ and a mapping $M \subseteq \mathcal{P}(D_1 \times D_2)$, the *Compare* operator is defined as:

$$\begin{array}{rcl} Compare : T \times T \times (D_1 \times D_2) & \to & \mathcal{P}(B) \times \mathcal{P}(A) \times \mathcal{P}(A \times B) \\ (T_1, T_2, M) & \mapsto & \langle In, Del, Subst \rangle \end{array}$$

where $In = Diff_{T_1,T_2} \cap B$, $Del = Diff_{T_1,T_2} \cap A$ and $Subst = Diff_{T_1,T_2} \cap (A \times B)$.

Note that while the comparison results are unordered, this does not prevent the presentation of the comparison result in a human-readable way. This can be done by iterating over the states of both traces in parallel, and looking for them in the trace difference. For instance, Fig. 5c was obtained from the trace difference obtained with *Compare* containing $\langle\{b_4\}, \{a_2\}, \{\langle a_3, b_2\rangle\}\rangle$ using the following reasoning:

- a_1 and b_1 are absent from the result: hence all their values are equal (first column).
- a_2 is not contained in a pair: hence it has been deleted from t_1 (second column).
- a_3 and b_2 are contained in a pair: hence some values are different from a_3 to b_2. These values can be identified by iterating over the dimension pairs that are part of the provided matching (third column).
- a_4 and b_3 are absent from the result: hence all their values are equal (fourth column).
- b_4 is not contained in a pair: hence it has been inserted in t_2 (fifth column).

4.5 State Graph Extraction

For each model state in an execution trace, there may be other *equivalent* model states scattered over the trace, which means that the execution is going back to this state several times during the execution. However, the sequential nature of a trace makes it difficult to grasp such information, and to understand the possible *cycles* in the execution trace. To provide a better understanding of the encountered model states, we propose the last operator called *Graph* (see Fig. 5d), which creates a directed graph from a trace, where each vertex is mapped to a set of equivalent states of the trace, and each event adds an edge between the vertexes containing its source and target states, if such an edge does not already exists. These edges also carry the set of events that caused their existence.

Definition 7 (*Graph*)**.** Let G be the set of all directed graphs. The *Graph* operator is defined as:

$$\begin{array}{rccc} \mathit{Graph}: & T & \to & G \\ & \langle S, D, E_<, \mathit{val}, \mathit{step}\rangle & \mapsto & \langle V, A\rangle \end{array}$$

where:

- V is the set of vertices, with $V = \{s \in \mathcal{P}(S) : \forall a, b \in s, a \equiv b\}$
- A is the set of directed edges, with $A = \{\langle v_1, \mathit{Events}, v_2\rangle \in V \times \mathcal{P}(E) \times V : \forall e \in \mathit{Events}, \exists a \in v_1, \exists b \in v_2, a \xrightarrow{e} b\}$

Example 4. Resulting graph from $\mathit{Graph}(\mathit{Filter}(T_{Ex}, \{d_3\}))$

$$\begin{aligned} \mathit{Graph}(\mathit{Filter}(T_{Ex})) = \langle\{&v_1 = \{s_1, s_2, s_4\}, v_2 = \{s_3\}\}, \\ &\{(v_1, \{e_1, e_2, e_3\}, v_1), (v_1, \{e_4\}, v_2), (v_2, \{e_5\}, v_1)\}\rangle \end{aligned}$$

5 Implementation and Evaluation

In this section, we first explain how we implemented the operators as part of the GEMOC Studio. We then present how we validate the approach using the motivating example conforming to the State Machine DSL from Section 2.3.

5.1 Implementation within the GEMOC Studio

We implemented the four operators within the GEMOC Studio, an open source (EPL 1.0) Eclipse package atop the Eclipse Modeling Framework. The GEMOC Studio includes a language workbench to implement executable DSLs, and a modeling workbench to create, execute and debug conforming models. We implemented a set of graphical views to display both execution traces and operators outputs (*i.e.,* traces, *diff* models and graphs) in a human-readable way.

At runtime, traces commonly reach a large amount of states that must be stored in memory. Therefore, while this is out the scope of this paper, our implementation aims at limiting the amount of memory required by the trace comprehension operators. Most notably, when both the input and the output of an operator are traces (*i.e.,* with *Filter* and *Reduce*), and when the output is significantly similar to the input, we produce a *virtual trace* that contains links to the concrete input trace instead of containing values, along with information on filtered dimensions (for *Filter*) or regrouped states (for *Reduce*).

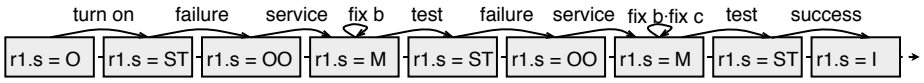
The source code of the operators can be found in the Github repository of the GEMOC execution framework[6], and more information can be found about the implementation on our companion web page[7].

5.2 Evaluation

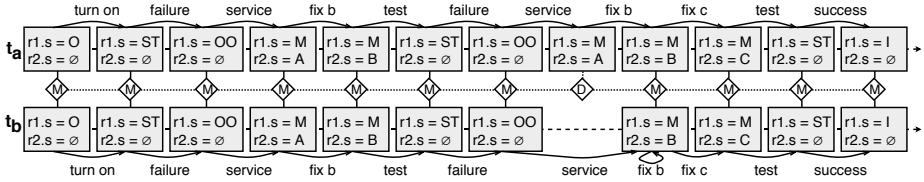
To evaluate the contribution of this paper, we demonstrate the usefulness of the proposed operators to understand the execution traces of the State Machine models previously shown as a motivation in Section 2.3. We recall that the models were depicted in Fig. 2, and the considered execution traces in Fig. 3. Fig. 6 shows different applications of the four operators to the two execution traces, most of them by combining the use of multiple operators. We explain below how the results help better understand the traces.

***Filter* and *Reduce*.** To obtain the trace shown in Fig. 6a from t_a, we first apply *Filter* on the `counter` and Maintenance.`currentState` dimensions, then we apply *Reduce*. We choose to filter the `counter` dimension because it changes at each state and thus hampers further cycle analysis or trace comparison, and the Maintenance.`currentState` in order to hide the internal working of the *Maintenance* hierarchical state. The result is a more high-level trace which only focuses on the information of interest, *i.e.,* which states of the main state machine were visited. This demonstrates that the *Filter* and *Reduce* can be used both to get

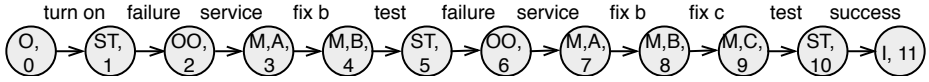
[6] https://github.com/eclipse/gemoc-studio-modeldebugging
[7] http://gemoc.org/ecmfa18



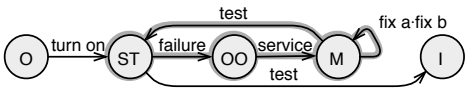
(a) *Reduce*(*Filter*(t_a, {`counter`, Maintenance.`currentState`}))



(b) *Compare*(*Filter*(t_a, {`counter`}), *Filter*(t_b, {`counter`, Maintenance.`lastState`}}))



(c) *Graph*(t_a)



(d) *Graph*(*Reduce*(*Filter*(t_a, {`counter`, Maintenance.`currentState`})))

Fig. 6: Various applications of the operators on the traces from in Fig. 3.

rid of noisy data (*e.g.,* the `counter`), and to modulate the level of detail featured in a trace my removing undesired dimensions (*e.g.,* Maintenance.`currentState`).

Compare. To obtain the trace difference shown in Fig. 6b, we first apply *Filter* on the `counter` dimension on t_a and t_b and on the Maintenance.`lastState` dimension on t_b, then we apply *Compare* on the resulting traces. The mapping of dimensions provided to *Compare* is not shown, as it is trivial except for the deep history dimension which has no match. Fig. 6b shows us that both traces align almost perfectly—except for a deleted state from one trace to the other—which was difficult to notice simply by looking at the original traces from Fig. 3. Note that this result is only possible because the traces were filtered before the comparison, since comparing unfiltered traces would not find much similarities because of the `counter` property. This demonstrates that the *Compare* operator, especially when combined with *Filter* and *Reduce*, can effectively help to understand subtle behavioral differences induced by design choices.

Graph. To obtain the graph shown in Fig. 6c, we directly applied *Graph* on the original t_a trace. We can observe that the resulting graph is of little interest as it takes the form of a sequence identical to t_a, which is mostly due to the

incremented `counter`. However, in Fig. 6d, we first applied the *Filter* operator on t_a to filter out the `counter` and Maintenance.`currentState` dimensions, followed by the *Reduce* operator. Applying *Graph* on the resulting trace shows us a better overview of the states visited during the execution. In particular, we can observe a *cycle* in the visited states, highlighted in gray. This demonstrates that the *Graph* operator, especially when combined with *Filter* and *Reduce*, can effectively help understanding which model states were visited in the execution, and which cycles can be observed between model states.

Additional material. Our companion web page[8] extends this evaluation with more complex models conforming to a real world DSL called ThingML.

6 Related Work

Several approaches rely on execution trace comparison to better understand the *semantic differences* between executable models [19,20,21,22,23]. Among the approaches closest to our work, the work done by Langer et al. [22] relies on dedicated *matching rules* to align pairs of traces in order to compute semantic differences, where a set of matching rules define how traces should be meaningfully compared in the context of a given executable DSL. In contrast, our generic approach does not require any matching rules as input, and instead relies on simplifying first the traces using *Filter* and *Reduce* in order to abstract away details that would prevent from aligning equivalent states.

Alimadadi et al. [24] propose a high-level abstraction operator that detects recurring patterns and hierarchies of patterns in sequences of events. Their algorithm is inspired from sequence alignment algorithms used in bioinformatics, similarly to our *Compare* operator. Overall, while the motivation for their work is the same as ours, our approach relies on a set of more low-level operators that manipulate sequences of both states and events. In other words, our operators could be used as basic building blocks for providing higher-level operators.

Process mining is a process-centric management technique bridging the gap between data mining and traditional Business Process Management (BPM) [25,26]. The main objective of process mining is to extract process-related information from event logs for providing information about actual processes [26]. Events are defined as process steps and event logs as sequential events recorded by an information system [27]. In [25], discovery is mentioned as one of the main goals of process mining, *i.e.,* taking an event log as input and to produce a process model as output. Event log comparisons techniques are also discussed in the realm of process mining [28]. Compared to our presented approach, process mining starts with logs produced by information systems and not directly by the interpretation of the process models. Furthermore, current process mining techniques are only applicable on business process modeling languages such as BPMN and their formal representation such as Petri nets. Our techniques are general enough to be applicable for executable modeling languages in general.

[8] http://gemoc.org/ecmfa18

Furthermore, we consider events and data while the latter is mostly neglected by process mining approaches.

7 Conclusion and Perspectives

Traces obtained from the execution of behavioral models are essential both as sources of feedback and to perform trade-off analyses. Yet, it remains difficult for a modeler to understand how a design change impacts the obtained execution traces. To address this problem, we proposed in this paper a set of formally defined *trace comprehension operators* which can be used for dynamic information filtering, trace comparison with *diff* computation and visualization, and graph-based views extraction to analyze cycles. We implemented our approach as part of the GEMOC Studio, an Eclipse-based language and modeling workbench, and we validated the approach using model variants conforming to a State Machine DSL. We showed that our operators can be used to better understand the impact of small but significant changes made to the considered model.

The direct perspectives of this work include extending the trace comparison operator to consider events along model states (*e.g.,* to compare different operational semantics with different observable events), improving the execution trace metamodel to support execution traces of concurrent behaviors (*i.e.,* a partial ordering of events), extending the state graph operator to consider multiple traces as input, providing an additional operator to compare state graphs, and specifying rigorous guidelines explaining how and when to use which operator.

Acknowledgments

This work was supported by: the Austrian Science Fund (FWF) P 28519-N31, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and the Safran/Inria/CNRS collaboration GLOSE.

References

1. Object Management Group: Semantics of a Foundational Subset for Executable UML Models, V 1.1 (August 2013)
2. Bendraou, R., Combemale, B., Crégut, X., Gervais, M.P.: Definition of an executable SPEM 2.0. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07), IEEE (2007) 390–397
3. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Proceedings of the 6th International Workshop Theory and Application of Graph Transformations (TAGT'98). Volume 1764. (2000) 157–167
4. Harel, D., Lachover, H., Naamad, A., Pnuelli, A., Politi, M., Sherman, R., Shtull-trauring, A., Trakhtenbrot, M.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering **16**(4) (1990) 403–414
5. OASIS: Web Services Business Process Execution Language Version 2.0 (2007)
6. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC'12). (2012) 282–287
7. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: Executable DSMLs based on fUML. In: Proceedings of the 6th International Conference on Software Language Engineering (SLE)'13. Volume 8225 of LNCS., Springer (2013)
8. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Proceedings of the 3rd International Conference on the Unified Modeling Language (UML'00). Volume 1939 of LNCS., Springer (2000) 323–337
9. Bandener, N., Soltenborn, C., Engels, G.: Extending DMM Behavior Specifications for Visual Execution and Debugging. In: Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10). Volume 6563 LNCS., Springer (2010) 357–376
10. Hegedüs, Á., Bergmann, G., Ráth, I., Varró, D.: Back-annotation of Simulation Traces with Change-Driven Model Transformations. In: Proceedings of the 8th International Conference on Software Engineering and Formal Methods (SEFM'10), IEEE (2010) 145–155
11. Soden, M., Eichler, H.: Towards a model execution framework for Eclipse. In: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture (BD-MDA'09), ACM (2009)
12. Tatibouët, J., Cuccuru, A., Gérard, S., Terrier, F.: Formalizing Execution Semantics of UML Profiles with fUML Models. In: Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14). Volume 8767 of LNCS., Springer (2014)
13. Bousse, E., Mayerhofer, T., Combemale, B., Baudry, B.: Advanced and efficient execution trace management for executable domain-specific modeling languages. Software & Systems Modeling (2017) 1–37
14. Object Management Group: Meta Object Facility (MOF) Core Specification, V 2.5 (June 2016) http://www.omg.org/spec/MOF/2.5.
15. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd Edition. Eclipse Series. Addison-Wesley Professional (2008)
16. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In: Proceedings of the International Conference on Software Language Engineering (SLE'15), ACM (2015)

17. Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: International Colloquium on Automata, Languages, and Programming, Springer (1990) 626–638
18. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. Volume 10. (1966) 707–710
19. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11). (2011) 163–172
20. Maoz, S., Ringert, J.O., Rumpe, B.: Cddiff: Semantic differencing for class diagrams. In: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11). (2011) 230–254
21. Maoz, S., Ringert, J.O., Rumpe, B.: Addiff: semantic differencing for activity diagrams. In: Proceedings of the SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13). (2011) 179–189
22. Langer, P., Mayerhofer, T., Kappel, G.: Semantic model differencing utilizing behavioral semantics specifications. In: Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS'14). (2014) 116–132
23. Addazi, L., Cicchetti, A., Rocco, J.D., Ruscio, D.D., Iovino, L., Pierantonio, A.: Semantic-based model matching with emfcompare. In: Proceedings of the 10th Workshop on Models and Evolution (ME'16). (2016) 40–49
24. Alimadadi, S., Mesbah, A., Pattabiraman, K.: Inferring hierarchical motifs from execution traces. (2018)
25. van der Aalst, W.M.P.: Process mining: making knowledge discovery process centric. SIGKDD Explorations **13**(2) (2011) 45–49
26. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. 1st edn. Springer Publishing Company, Incorporated (2011)
27. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software Through Process Technology. Wiley (2005)
28. Bose, R.P.J.C., van der Aalst, W.M.P.: Process diagnostics using trace alignment: Opportunities, issues, and challenges. Inf. Syst. **37**(2) (2012) 117–141