Software and Systems Modeling (2025) 24:721–739
https://doi.org/10.1007/s10270-024-01212-x

SPECIAL SECTION PAPER

Operation-based versioning as a foundation for live executable models

Joeri Exelmans[1] · Ciprian Teodorov[2] · Hans Vangheluwe[1,3]

Received: 22 March 2024 / Revised: 4 July 2024 / Accepted: 4 September 2024 / Published online: 15 October 2024

Abstract
Live modeling is the ability to edit an executable model at run-time, and to subsequently continue the execution instead of having to restart it. Few modeling frameworks support this feature. Much of the research concerning live modeling attempts to bring "liveness" to existing modeling languages and environments, which is a complex, and often ad hoc endeavor. We instead argue to build modeling environments on an operation-based versioning foundation, to not only record edit operations, but also execution steps on an explicit run-time model. This reduces the complexity of patching the run-time state with edit operations to a simple merge-operation, while getting powerful features such as collaborative editing and debugging "for free."

Keywords Model-driven engineering · Live programming · Live modeling · Versioning

1 Introduction

The mental gap that occurs when turning a set of goals into an engineered system is called the "gulf of execution" by Donald Norman [1]. Verifying whether the system satisfies the goals, he calls the "gulf of evaluation." In the context of programming, Lieberman and Fry [2] link these gulfs to the difficulty of understanding the behavior of a program from the static source code, and introduce one of the first instances of omniscient debugging. However, debugging only addresses the gulf of evaluation, helping the user comprehend the behavior by linking it with the code and allowing their conjoint exploration. Live programming addresses the other direction, the "gulf of execution," helping the user to nudge an existing program in the right direction by modifying it during its execution. When the user can modify a program at run-time to address some newly identified gap between her goals and the running system, this frees her from traversing the entire "gulf of evaluation" once again (re-running from the beginning and evaluating if the change matches the expectations). This is supported by an empirical study [3] that found that programmers tend to make use of (simple) live programming techniques, when available.

In both software development and the engineering of (cyber-)physical systems, the use of (executable) modeling is taking on an increasingly important role. By using a modeling language that fits the problem domain at hand, rather than using some generic programming language (e.g., Java), the accidental complexity and the cognitive gap are reduced. However, modeling tools still lag behind the rich tool ecosystems available for programming languages, such as debuggers and versioning systems.

Programming languages benefit from the fact that they are usually textual, and therefore, a textual versioning system like git can be used for many languages. We similarly propose for modeling languages to use a "largest common denominator" data structure, which is graph-based. A graph-based data structure to encode not only the models themselves, but also the run-time state of an interpreter, combined with a powerful versioning system for graphs, we believe live modeling can be fairly easily supported across a wide range of modeling languages, taking the modeling experience beyond that of textual languages.

Communicated by Javier Troya and Alfonso Pierantonio.

✉ Joeri Exelmans
Joeri.Exelmans@uantwerpen.be

Ciprian Teodorov
Ciprian.Teodorov@ensta-bretagne.fr

Hans Vangheluwe
Hans.Vangheluwe@uantwerpen.be

[1] AnSyMo, Department of Computer Science, University of Antwerp, Middelheimlaan 1, 2020 Antwerp, Belgium

[2] Lab-STICC CNRS UMR 6285, ENSTA Bretagne, 2 Rue François Verny, 29806 Brest, France

[3] Flanders Make @ UAntwerp, Antwerp, Belgium

1.1 Contributions

In our previous work [4], we proposed an architecture that can support collaborative live modeling and multiverse debugging [5, 6]. The critical insight was that an operation-based optimistic versioning system (see [7] for a nice classification) can be used to record not just users' edit operations on a *design model*, but also an interpreter's execution steps on a *run-time model*. By defining the run-time model such that it is always an overlay on top of the design model, the "patching" of the execution state after a "live" edit operation can be reduced to the *merging* of changes of the design model into the run-time model. The hypothesis is that if there are no merge conflicts, then the updated run-time model is correct.

The novelty of this approach, compared to existing work, is that the only work needed to support a new language is to properly encode (changes to) the design model and run-time model. We are convinced that this is far easier than creating and maintaining ad hoc merge-functions for every modeling language (the usual approach taken).

Additionally, the language engineer gets some powerful features "for free": *branching* and merging of the design model is supported, the basis for collaborative editing. Nondeterministic executions can be represented by branching the execution state, and hence (parts of) the state space can be recorded, and used for model checking or multiverse debugging [6]. With serialization of recorded changes (part of the versioning system), execution traces can be stored or sent over a network connection without any extra work. This enables even more features such as collaborative (remote) debugging.

In this paper, we present an implementation and extension of these ideas:

- We have implemented a demo of a live modeling environment for a simple FSA language, based on our earlier running example [4]. We will give an overview of its implementation.
- We address a flaw in our previous work [4]: in order to detect merge conflicts, we record *dependencies* between changes (which we call *deltas*). Originally, we only recorded *write-after-write* dependencies between *deltas*. This was sufficient to detect conflicts between edit operations. We did not realize at the time the need to additionally record *read-after-write* dependencies to detect conflicts between execution steps and/or edit operations. Without read-after-write dependencies, certain conflicts (e.g., a concurrent read + write) remain undetected. We have thus extended our implementation with this feature.

The remainder of this paper is structured as follows: In Sect. 2, we present our running example, which consists of a number of scenarios that demonstrate the overlap between versioning and live modeling. In Sect. 3, we present our solution and implementation for supporting these scenarios. In Sect. 4, we discuss possible extensions of our work. In Sect. 5, we discuss related work. In Sect. 6, we conclude, discuss limitations and future work.

2 Running example

In this section, we introduce our running example, and the use cases we want to support.

Before we start, we state explicitly that we focus on interpretation (i.e., *operational semantics*), rather than compilation/code generation (i.e., *translational semantics*). Translational semantics can be supported, but requires extra work (explained later).

2.1 Design and run-time model

Inspired by [8], we assume two kinds of models exist:

- *Design model* is the model that is created/edited by the user
- *Run-time model* is the model that describes the current state of execution, typically between execution steps

Figure 1 introduces our running example, based on a simple finite state automaton (FSA) modeling the behavior of a burglar alarm. Four models are shown:

	Design	Run-time
Meta-model	MM_D	MM_{RT}
Model	$M_D^{(1)}$	$M_{RT}^{(A)}$

The meta-models are shown only for completeness, and are assumed not to change. The design model contains all states and transitions. The only information contained in the run-time model is the current state. The current state is stored as a link directly to a state in the design model. In general, the run-time model is able to point to elements of the design model, but not the other way around. We will define this more precisely as an *embedding* relationship when explaining our implementation in Sect. 3.

2.2 Edit operations, execution steps, god events

In non-live modeling environments, there is an explicit distinction between *design-time* and *run-time*. At design-time, the modeler performs edit operations on the design model, and at run-time, the interpreter performs execution steps on the run-time model. Possibly, there exists a debugger that

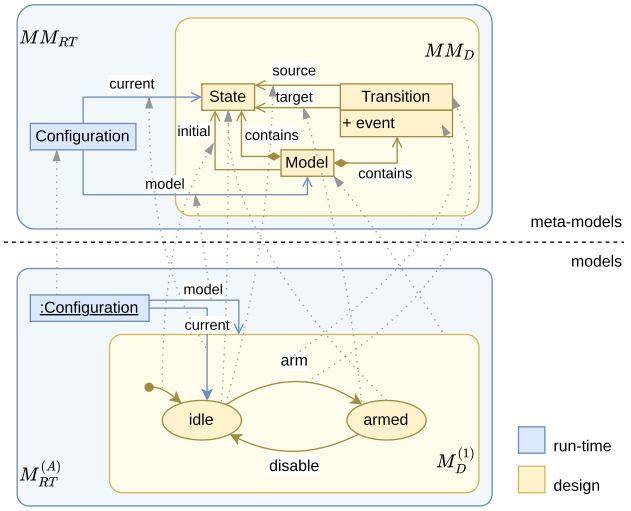

Fig. 1 Running example: design and run-time (meta-)models

Table 1 Terminology of changes

what changes?	by user	by interpreter
design model	edit operation	intercession
run-time model	god event	execution step

allows to pause the interpreter, and let the user inspect the run-time model. We also introduce a third kind of change. Some debuggers allow the user to change the run-time model directly. This kind of change is called a *god event* by [9]. For instance, the user could override the current state in an FSA, or could modify the value of a variable in an action language. To be complete, we must mention that some (reflective) languages allow the model to observe its own structure at run-time (called *introspection*), and even modify its structure (called *intercession*). In this paper, we consider reflective languages out of scope. Table 1 summarizes the terminology of changes.

In *live modeling* environments, the explicit distinction between design-time and run-time falls away: the user can continue making changes to the design model at run-time. When restricting ourselves to interpretation, the essential complexity is reduced to "patching" the run-time model to include design model changes, as explained in [8].

Running example Figure 2 shows a small scenario. First, the user performs an edit operation, adding a state "detected" and two transitions "personDetected" and "correctCode" to the FSA. Then, the user initializes execution, which results in the creation of a run-time model. We consider this initialization a kind of execution step. Another execution step occurs when an input event "arm" is received, updating the current state to "armed." If instead of performing an execution step, the user can also directly set the current state to "detected," which would be a god event.

2.3 Rollback, branching and merging

Both programmers and "modelers" have gotten used to the ability of rolling back changes at design-time, and creating different branches of their models, via "undo" and/or versioning. Some debuggers, called *omniscient debuggers* also allow rolling back execution steps [10, 11]. *Multiverse debuggers* are even more powerful, as they deal with branching run-time models, and can automatically execute multiple branches in the state space until a *multiverse breakpoint* triggers, narrowing the gap between execution and model checking. We now give examples of branching and merging at the level of design model and run-time model.

Running example Figure 3 shows a branching design model. Two different edit operations (adding one state and removing another) happen *concurrently* (i.e., there exists no ordering between them). We argue that it should be possible to merge the effects of these operations, because they are not overlapping.

Figure 4 also shows a branching design model, but this time the different edit operations overlap in such a way that they are conflicting. The state "armed" is removed, and concurrently, new transitions "personDetected"/"correctCode" are created that have this removed state as their source/target. We argue that merging these operations is not possible, because their full effects cannot both be executed: a decision needs to be made on whether to keep either the "personDected"/"correctCode" transitions, or the "armed" state.

Figure 5 shows a branching run-time model that we are already familiar with (from Sect. 2.2). Merging the execution steps is not possible here; each execution step wants to set the current state to a different target.

Figure 6 shows an extension of our running example: a Statechart model with two regions exhibiting *parallel independence*. In each region, a transition could be made independently of the other region. The transitions could be made in any order, resulting in the same run-time model. Every transition could be a concurrent execution step, and these steps could be merged on a data level, because they update non-overlapping parts of the model.

2.4 Reconciling design and run-time changes

So far we have seen how branching and merging can happen to design models and run-time models individually. We now show by means of our running example that live modeling can be supported by the merging of design and run-time model changes.

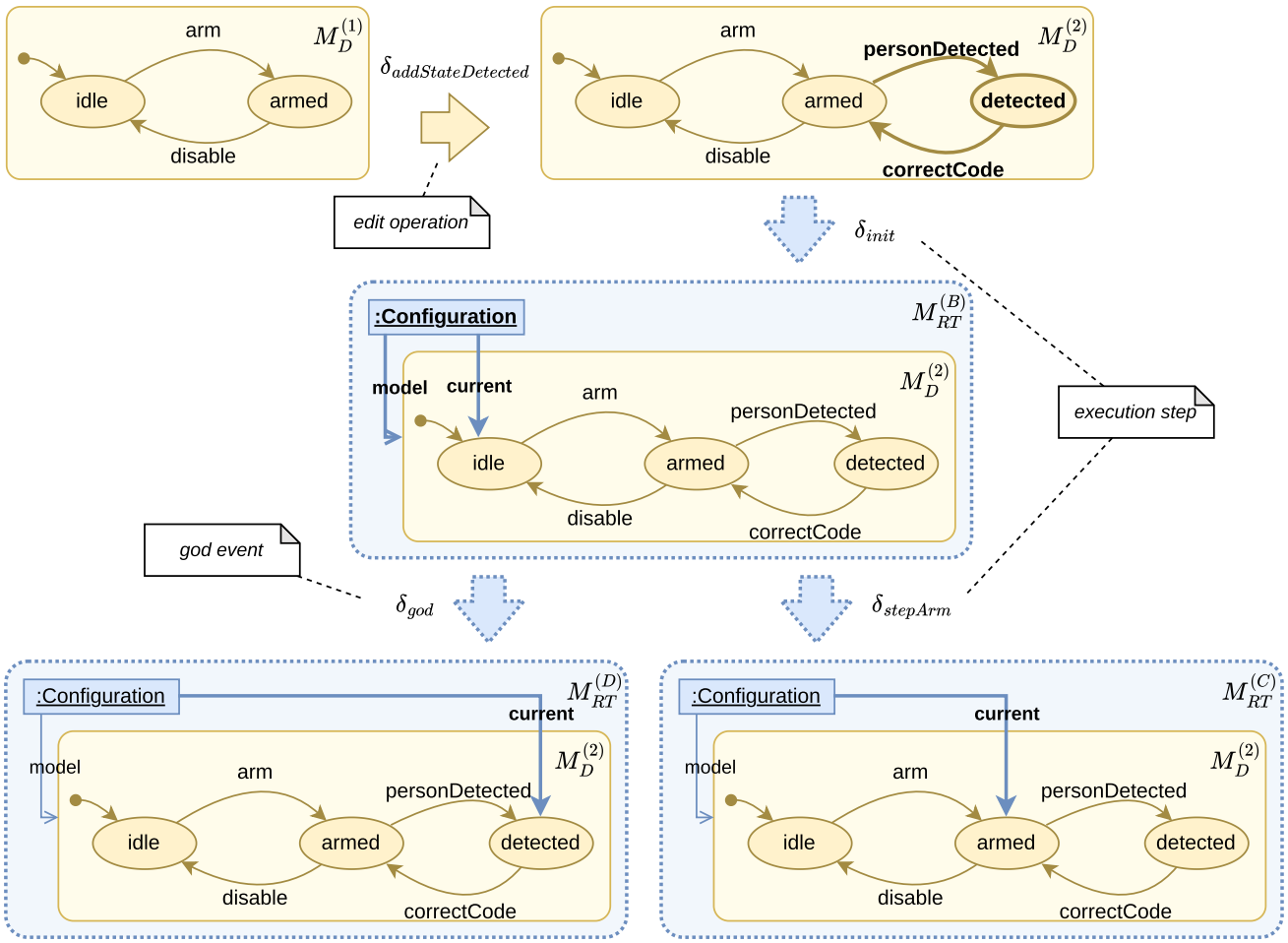

Fig. 2 Running example: edit operation, execution step, god event

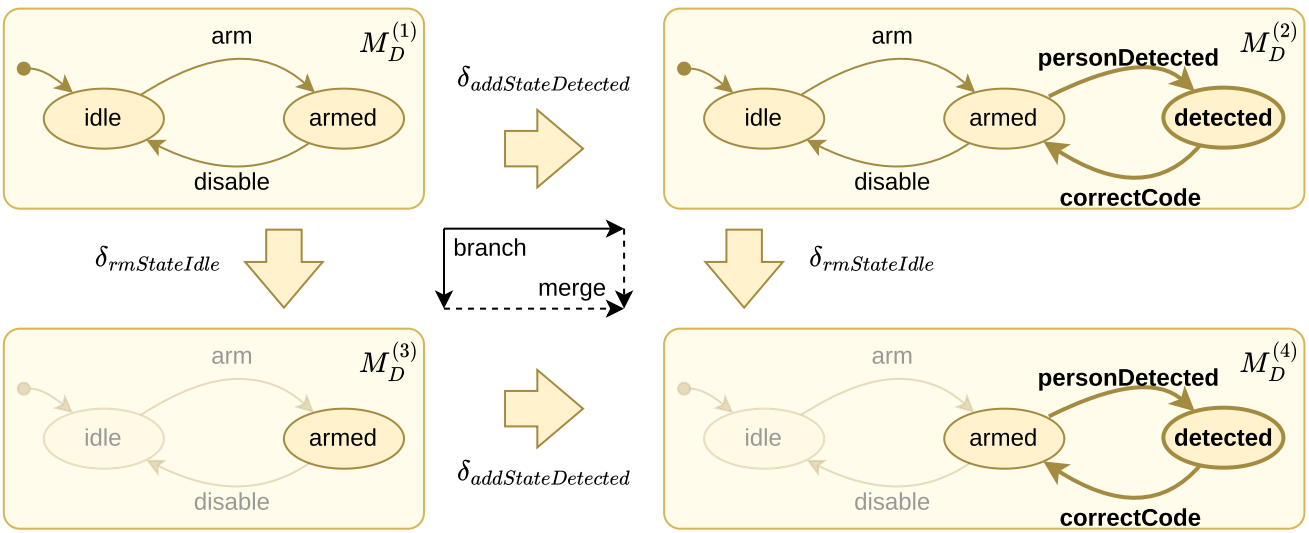

Fig. 3 Running example: design model branching and merging (no conflict)

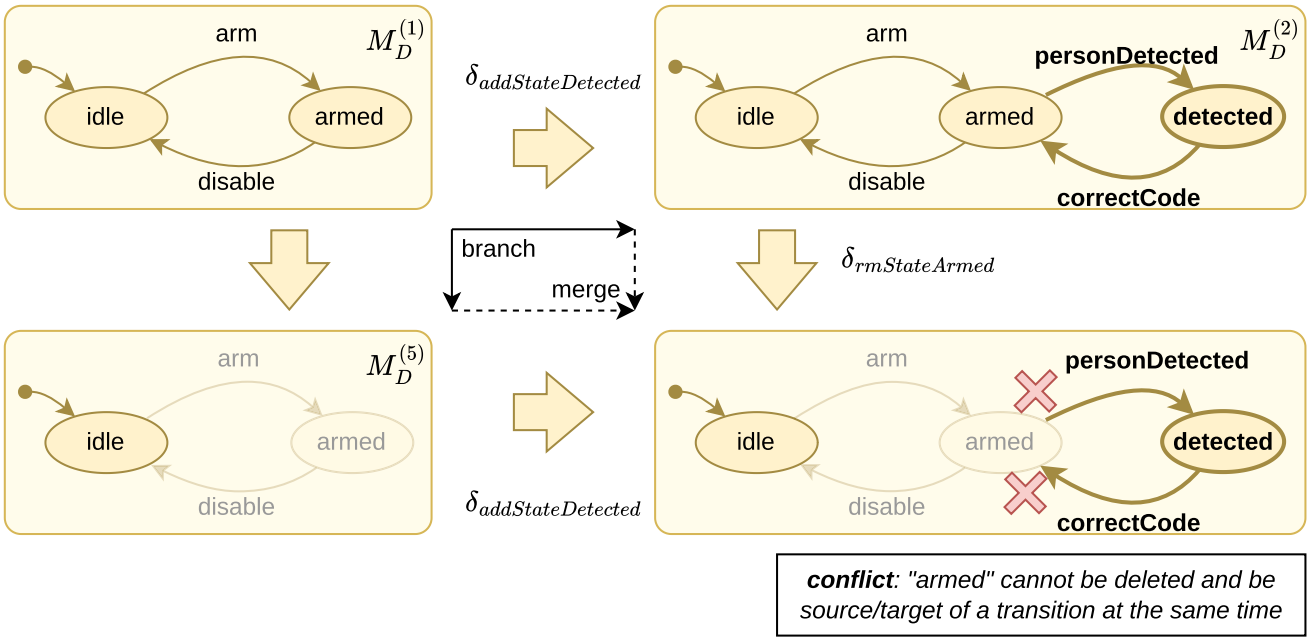

Fig. 4 Running example: design model branching and merging (conflict)

Running example Figure 7 shows two branches, one with design model changes, another with run-time model changes. In one branch, our original FSA is edited. In the other branch, the original FSA is initialized, and an execution step performed. We observe that there are no overlapping changes, and, following the same reasoning as the previous examples, we argue that these changes can be merged.

Figure 8 shows a merge conflict between design and run-time model changes: initialization happens concurrently with deletion of the initial state. Intuitively, this should be a conflict. Also at a data level, we observe that the initialization attempts to create a link ("current") to a deleted state ("idle"), which can be considered a conflict.

2.5 Recorded event versus real-time

Looking at the bottom of Fig. 8, if we delete the state "idle" *after* making the transition to "armed" (instead of concurrently), could we just continue execution with "armed" being the current state, even though the initial state has been deleted? It depends—in literature, two approaches to live modeling are described [8]:

- *Recorded event* is an approach where all execution steps are simply replayed after modifying the design model, which may fail.
- *Real-time* is an approach where the current run-time model is "patched" to incorporate changes in the design model, which may also fail. During "patching," the history of the run-time model (i.e., the execution trace) is ignored.

In our example, recorded event would fail to replay the execution steps, whereas the real-time approach would allow execution to continue. We believe that both approaches have their (dis)advantages: Recorded event is more sensitive to failure: for instance, in an FSA, removing state that ever was the current state, will make it impossible to re-execute the current trace. But if recorded event succeeds, it is assured that the reconciled run-time model is in a valid, reachable configuration. Real-time is more forgiving, and probably more intuitive to the user, although she may end up in unreachable configurations (such as $M_{RT}^{(}F$) in Fig. 8). We will show that our approach unifies both approaches, offering "the best of both worlds."

3 Solution

Now that we have seen how versioning, branching and merging can uniformly support features such as (1) collaborative editing, (2) omniscient debugging, (3) detecting parallel independence between execution steps,[1] and (4) live modeling, the essential complexity of an implementation becomes:

[1] Clearly, we can only do this at run-time, not statically.

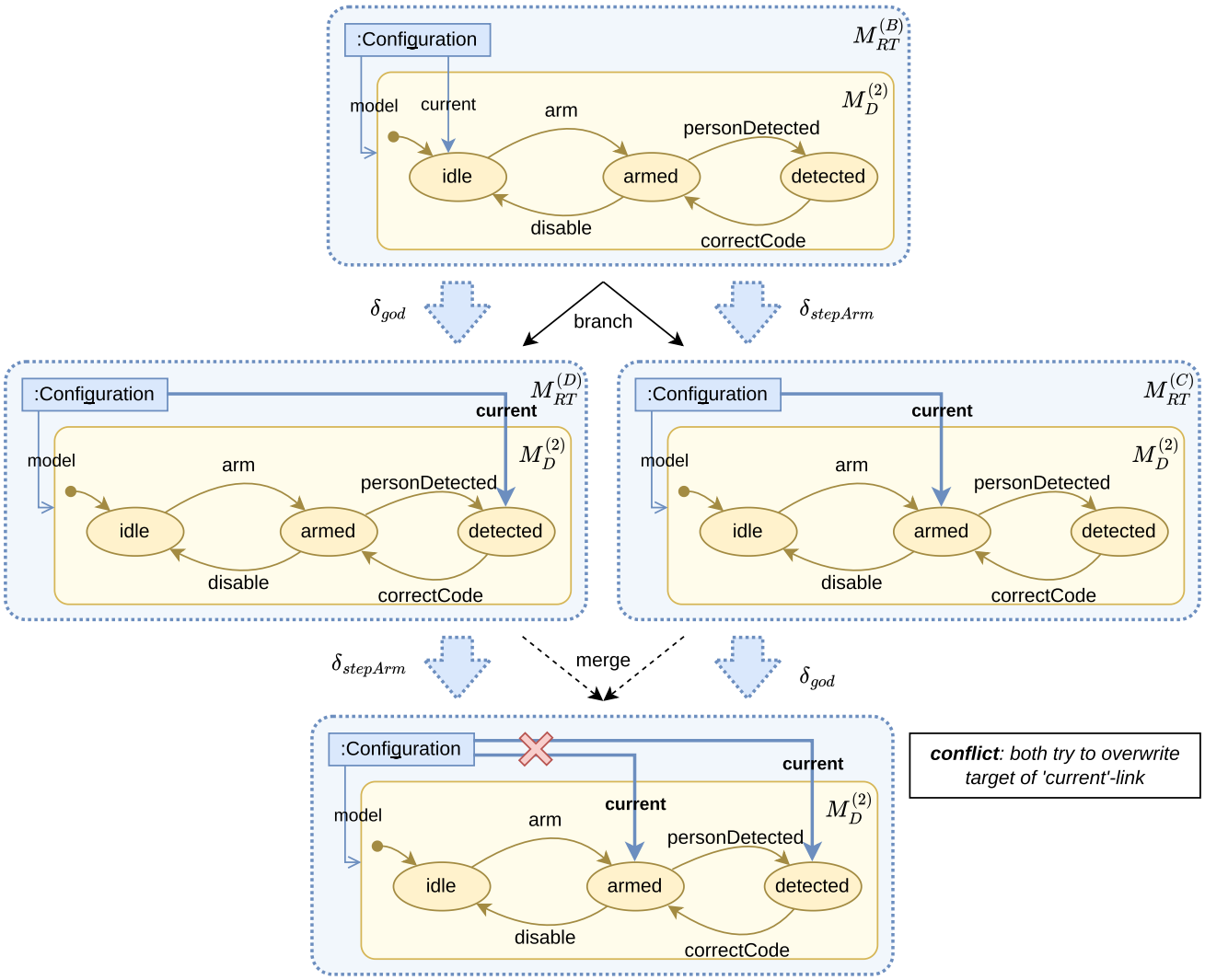

Fig. 5 Running example: run-time model branching and merging (conflict)

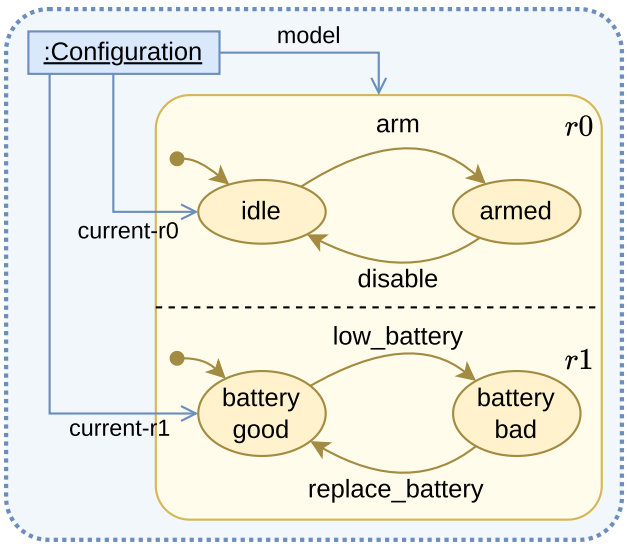

Fig. 6 Statechart model with parallel independence

1. Having a versioning system that is powerful enough to record fine-grained changes (when they happen), and to represent arbitrary models.
2. Integrating a modeling environment and interpreter with such versioning system, and encoding all changes in such a way that the versioning system will detect merge conflicts correctly.

Obviously, we will need an optimistic versioning control system (VCS) to support branching and merging. A further distinction can be made between state-based and operation-based VCS. State-based VCS, like git, only store snapshots of versions, and a partial ordering between these snapshots. They have to resort to *diffing* to figure out what the changes between two versions are. Operation-based VCS store the changes (called *deltas*) themselves as first class objects. For a detailed feature model of VCS, we refer to [7].

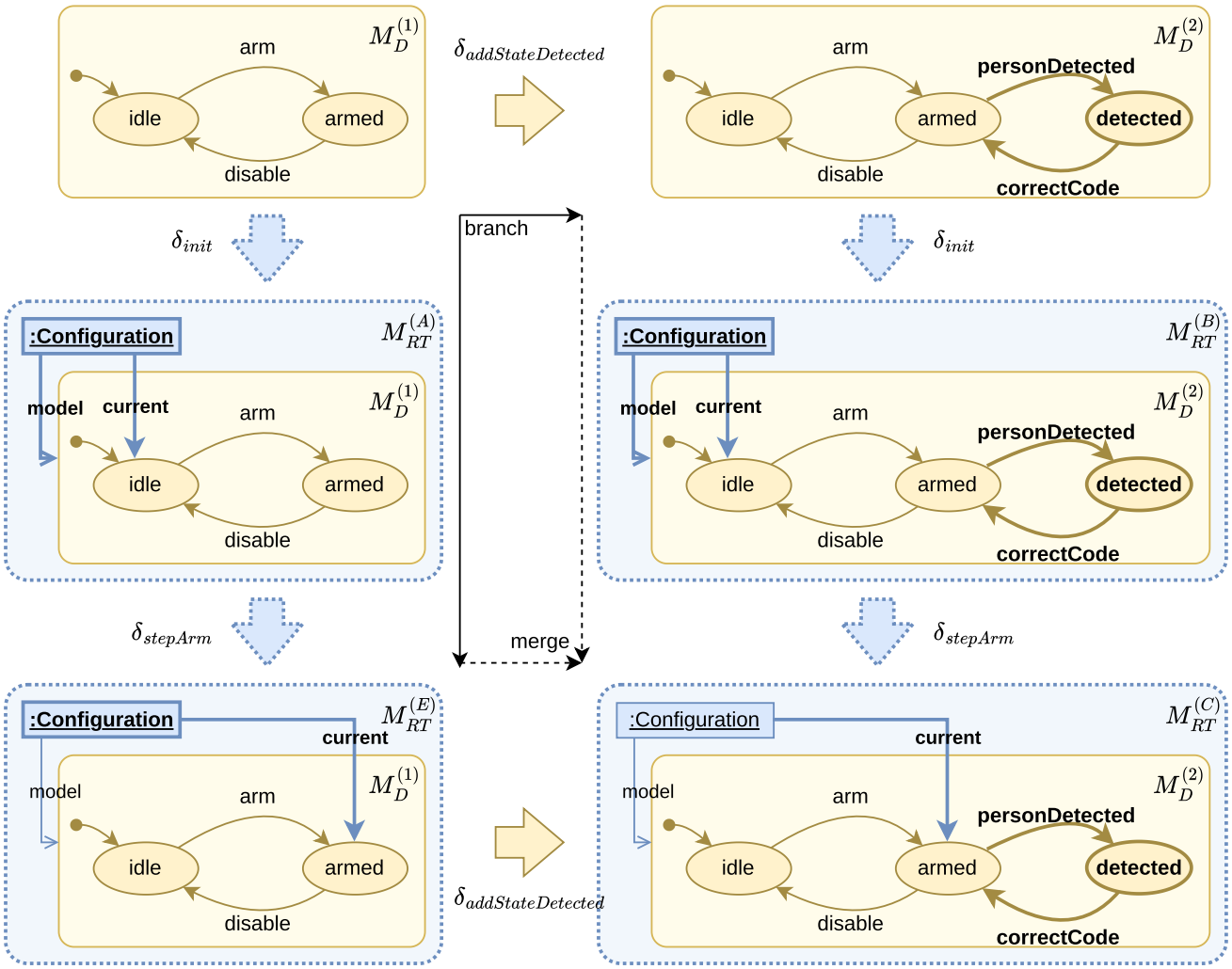

Fig. 7 Running example: merging design model changes into run-time model (no conflict)

We use an operation-based VCS from our earlier work [12], because it has several benefits; there is no need to perform diffing, and it records *dependency* relations between deltas. A delta δ_b depends on a delta δ_a if δ_b happened after δ_a and δ_b cannot be executed without δ_a having executed first. A delta will depend on another delta if it overwrites, reads or connects to an element created or written by an earlier delta. In our running example, the creation of a transition will depend on the creation of its source and target states. The creations of two different states will be independent wrt. each other, because they can be safely executed in any order. Deltas are immutable objects, and their dependency relations form a directed acyclic graph (DAG). A delta's dependencies are known when the delta is created, and are computed in constant-time ($\mathcal{O}(1)$).

We treat a *conflict* as a symmetric relation between two deltas. Two deltas can only be conflicting if they directly (non-transitively) share a dependency, as shown in Fig. 9. This can be intuitively understood as if they try to do different things with the same result; in our running example, the creation of a transition depends on the creation of its source and target states, and the deletion of a state depends on the creation of the deleted state. If a transition is created, and its target state is deleted, then there is a common dependency on the creation of the source state (and there will be a conflict, explained later).

Typically, the number of deltas with whom a delta has a shared dependency is small, so conflicts are also computed fast, and eagerly; immediately after the creation of a new delta, its conflict relations with other deltas are known (even before attempting to merge them) (Fig. 10).

3.1 Graph datatype

Rather than defining domain-specific delta-, dependency- and conflict types for every new language that we want to

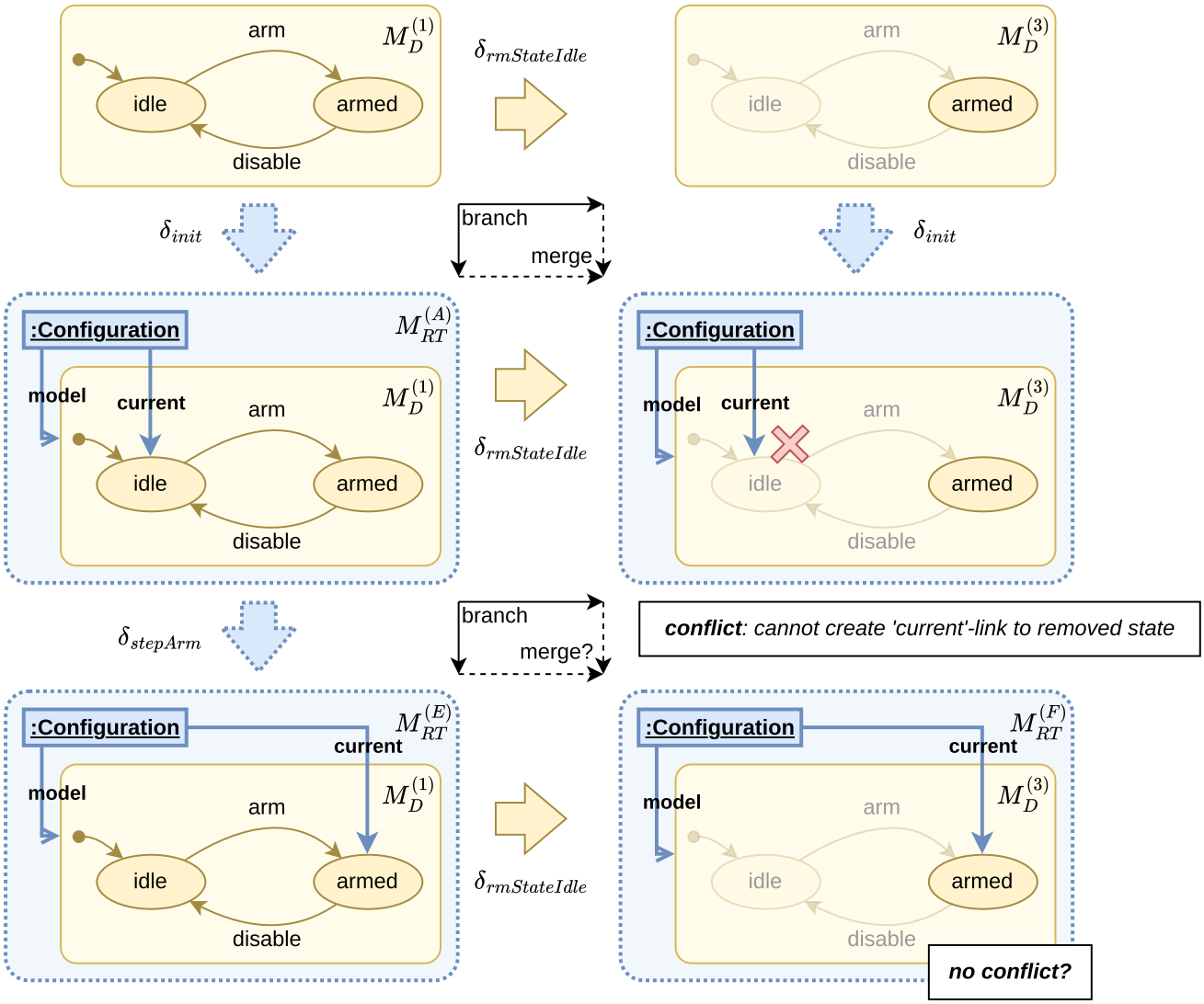

Fig. 8 Running example: merging design model changes into run-time model (conflict)

support, we instead define a generic,[2] graph datatype, and a set of *primitive deltas* that are the smallest possible operations that can be performed on this datatype. Accordingly, we have defined a small but exhaustive set of dependency and conflict types between our primitive deltas. Primitive deltas can be composed into *transactions* that represent larger changes.

For a detailed description of our graph datatype, the set of primitive deltas, their dependency and conflict relations, we refer to [12].

We summarize the graph datatype here: The graph structure can consist of (1) nodes with immutable GUIDs, (2) primitive values (numbers, strings, booleans and "null") and (3) edges, that always have a node as their source, and a node or primitive value as their target. Nodes are created and

[2] Our approach is independent of (meta-)meta-models such as MOF/UML, or serialization formats such as XMI. These standards are big, and contain redundant concepts (e.g., a property of an object could be modeled as an attribute, or as a link to an object representing the property). We prefer to keep our solution as small as possible, so we can focus on the essential complexity. It should be possible however, to map XMI-encoded models onto our own (simpler) graph datatype, a task we prefer to leave to an XMI expert. We should also note that XMI defines its own data format for encoding "differences" to XMI models, as sequences of the primitive operations "add," "replace" and "delete." However, a sequence of "differences" can only be executed on one particular XMI model, and cannot be trivially reordered or merged. Our approach is different, because our own primitive operations do not depend on a particular *state* but on *earlier operations*. As a result, they are only *partially ordered* (as opposed to *totally ordered*), making it trivial to reorder and merge operations. Finally, the main benefit of supporting a standard such as XMI would be import/export compatibility with a tool like Eclipse. But, the features we are

Footnote 2 continued
interested in (collaborative live modeling and debugging) need actual integration between the modeling/simulation tool and the versioning system. Implementing this in a tool like Eclipse would mean an unthinkable amount of re-engineering at its core.

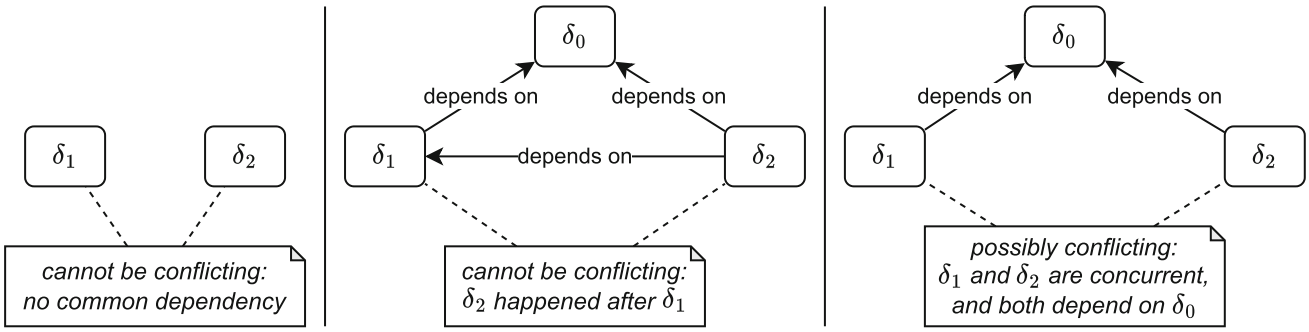

Fig. 9 A conflict can only occur when the pattern on the right is occurs

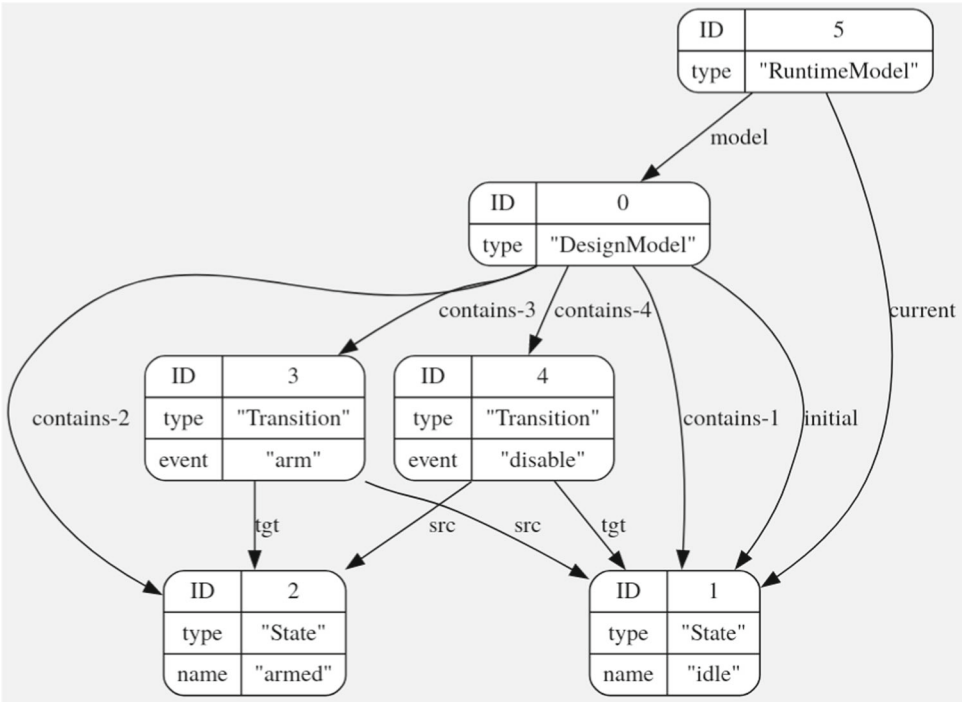

Fig. 10 Running example: graph encoding of $M_{RT}^{(A)}$ (screenshot from our demo)

destroyed. Edges and values are not created or destroyed. Edges can be updated. (Initially, all outgoing edges of a node are assumed to already exist, and have target "null.") Values are assumed to always already exist. Figure 11 shows the meta-model of the graph datatype.

3.1.1 Encoding models as graphs

Since graphs are the most generic data structure, it is fairly straightforward to use them to encode arbitrary models, or to map other data structures onto them, such as lists, sets, maps and trees [12]. In our running example, the models are essentially object diagrams, which can be mapped to graphs as follows: Every object becomes a node, and every slot/property becomes an outgoing edge with as label the name of the property, and as target the value of the property. Links between objects are encoded as edges between nodes, with the link name as label. Finally, the type of an object (i.e., the class name) can be encoded as if it were a property.

Running example Figure 10 shows how our implementation encodes of $M_{RT}^{(A)}$ from our running example. A rountangle represents a node (with unique ID), arrows represent edges between nodes, and attributes (in the rountangles) represent edges from nodes to values.

3.2 Primitive deltas and dependencies

Figure 12 shows the meta-model of the three primitive delta types and their dependency relations. The classes in the figure are delta types, and the associations are dependency types. We briefly discuss the primitive deltas and their dependency types:

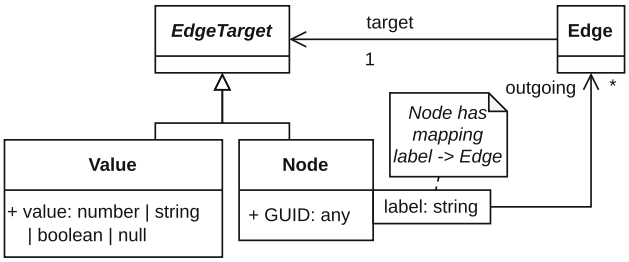

Fig. 11 Class diagram of graph datatype

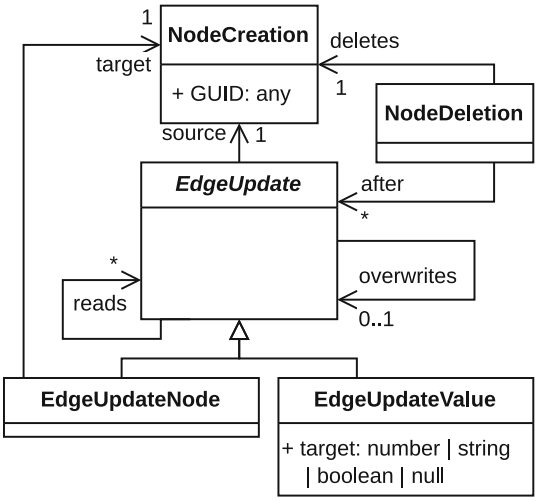

Fig. 12 Class diagram of primitive deltas and dependency relations

- *NodeCreation* A node creation. It has no dependencies.
- *NodeDeletion* A node deletion. Its dependency types are:
  - *Deletes* The NodeCreation of the node being deleted.
  - *After* A concurrent NodeDeletion and EdgeUpdate on the same node would be conflicting. Therefore, a new NodeDeletion must explicitly depend on all most recent EdgeUpdates for all edges currently existing.
- *EdgeUpdate* The creation or update of an edge. Its dependency types are:
  - *Source* The NodeCreation of the source of the edge being updated.
  - *Target* The NodeCreation of the target of the edge being updated.
  - *Overwrites* (optional) The previous EdgeUpdate is overwrites.
  - *Reads* When an EdgeUpdate is the result of some computation (e.g., an execution step), then it must have a read-dependency on all earlier EdgeUpdates whose results it depends on, but did not overwrite. This dependency type did not yet exist in [12].

3.3 Conflicts

Table 2 exhaustively shows all possible conflict types between primitive deltas. We briefly explain:

- *Source/Deletes* Creating an edge from a node that is concurrently deleted.
- *Target/Deletes* Creating an edge to a node that is concurrently deleted.
- *Deletes/Deletes* Two concurrent deletions of the same node.
- *Overwrites/Overwrites* Two concurrent updates of the target of the same edge.
- *Overwrites/Reads* Updating the target of an edge that is concurrently read.

3.4 Transactions

Deltas can be composed into *transactions*. Transactions are deltas themselves; they can have dependency and conflict relations with other transactions (derived from the deltas they consist of), and can again be composed into larger transactions.

Running example Figure 13 shows a subset of the transactions of edit operations involved in the creation of our initial design model $\left(M_D^{(1)}\right)$. Notice how the transactions are composed of primitive deltas. The dependencies between the transactions are derived from the primitive deltas they are composed of.

3.5 Versions

A *version* is simply an unordered set of non-conflicting deltas. To restore the graph state associated with a version, its deltas are replayed. The execution order of the deltas must be consistent with the dependency links between the deltas.

Merging of versions is a fairly simple operation, because much of the "work" (dependency and conflict detection) is done up-front. Versions are merged by taking the union of the sets of deltas they consist of. If conflicts exist between the union of all deltas, then not all deltas can be included in the result, and a decision needs to be made which deltas to include. One can construct automatically the *maximal non-conflicting subsets* of all deltas (i.e., to which no delta can be added without introducing a conflict). Each of these sets is a valid version that can be presented to the user to "resolve" the conflict(s).

Table 2 Conflict types between primitive deltas

	NodeCreation	EdgeUpdate	NodeDeletion
NodeDeletion	–	Source/Deletes, Target/Deletes	Deletes/Deletes
EdgeUpdate	–	Overwrites/Overwrites, Reads/Overwrites	
NodeCreation	–		

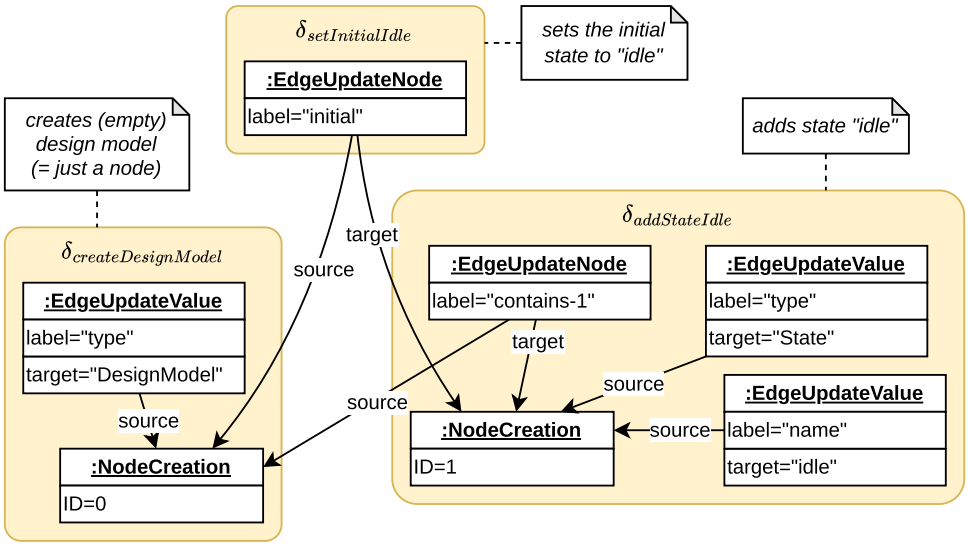

Fig. 13 Running example: transactions are composed of primitive deltas

Running example Figure 14 shows the bigger picture of all the transactions involved in the scenario of Fig. 8. Our original FSA model $M^{(1)}$ consists of a transaction ($\delta_{\text{createDesignModel}}$) that creates a node representing the model itself (with ID 0 in Fig. 10), followed by the creation of two states ($\delta_{\text{addStateIdle}}$, …) and two transitions ($\delta_{\text{addTransitionArm}}$, …), and a transaction setting the initial state to "idle" ($\delta_{\text{setInitialIdle}}$). The creation of the states and transitions depends on the transaction $\delta_{\text{createDesignModel}}$, because of the created containment links ("contains-1," "contains-2," …see Fig. 10). The creation of every transition depends on the creation of its source and target states.

The run-time model $M_{\text{RT}}^{(A)}$ additionally consists of the δ_{init} transaction. Notice its reads-dependency on $\delta_{\text{setInitialIdle}}$: the interpreter needs to know the initial state, so it reads the target of the "initial" edge created by this delta. Without this dependency, the initial state could be changed concurrently with initialization, and this conflict would be missed (a *false negative*).

Performing an execution step along the "arm"-transition (δ_{stepArm}) brings us to $M_{\text{RT}}^{(E)}$. This delta also has a read-dependency on the creation of the transition (it reads the source, target, and label). It also overwrites the current state (previously set by δ_{init}).

Going back to the design model, removing the state "idle" ($\delta_{\text{rmStateIdle}}$) obviously has delete dependencies on $\delta_{\text{addStateIdle}}$ and $\delta_{\text{addTransitionArm}}$ (its outgoing transition). It also overwrites the initial state ($\delta_{\text{setInitialIdle}}$) with "null." There is a conflict relation with δ_{init}, for which there are two reasons: (1) there is an deletes/target conflict on $\delta_{\text{addStateIdle}}$, and (2) there is a reads/overwrites conflict on $\delta_{\text{setInitialIdle}}$. We therefore cannot merge this change into our ongoing execution.

Supporting real-time and recorded event We saw in Sect. 2.5 that there are two approaches to live modeling: *recorded event* and *real-time*. Recorded event is trivially supported by attempting to merge design model changes into the run-time model, as we saw in the previous example.

With recorded event, no state that ever was the current state can be removed. With real-time live modeling, this becomes more relaxed: any state can be removed, as long as it is not the current state *right now*. Continuing the previous example, suppose we would allow the removal of the "idle" state after making the "arm"-transition. Observe that the conflict between $\delta_{\text{rmStateIdle}}$ and δ_{stepArm} would not be there if the former happened *after* the latter. This needs to be recorded with an after-dependency, but we cannot add a dependency from a design model delta to a run-time delta: the design model must remain independent of its execution(s). The solution is to create two variants of the "idle" state removal-delta. The original delta ($\delta_{\text{rmStateIdle}}$) remains used in the context of the design model. A new ($\delta_{\text{rmStateIdle}'}$) replaces the original delta in the context of the run-time model, and has the extra after-dependency on δ_{stepArm}. We persist a special "overrides" relation from the second to the first delta.

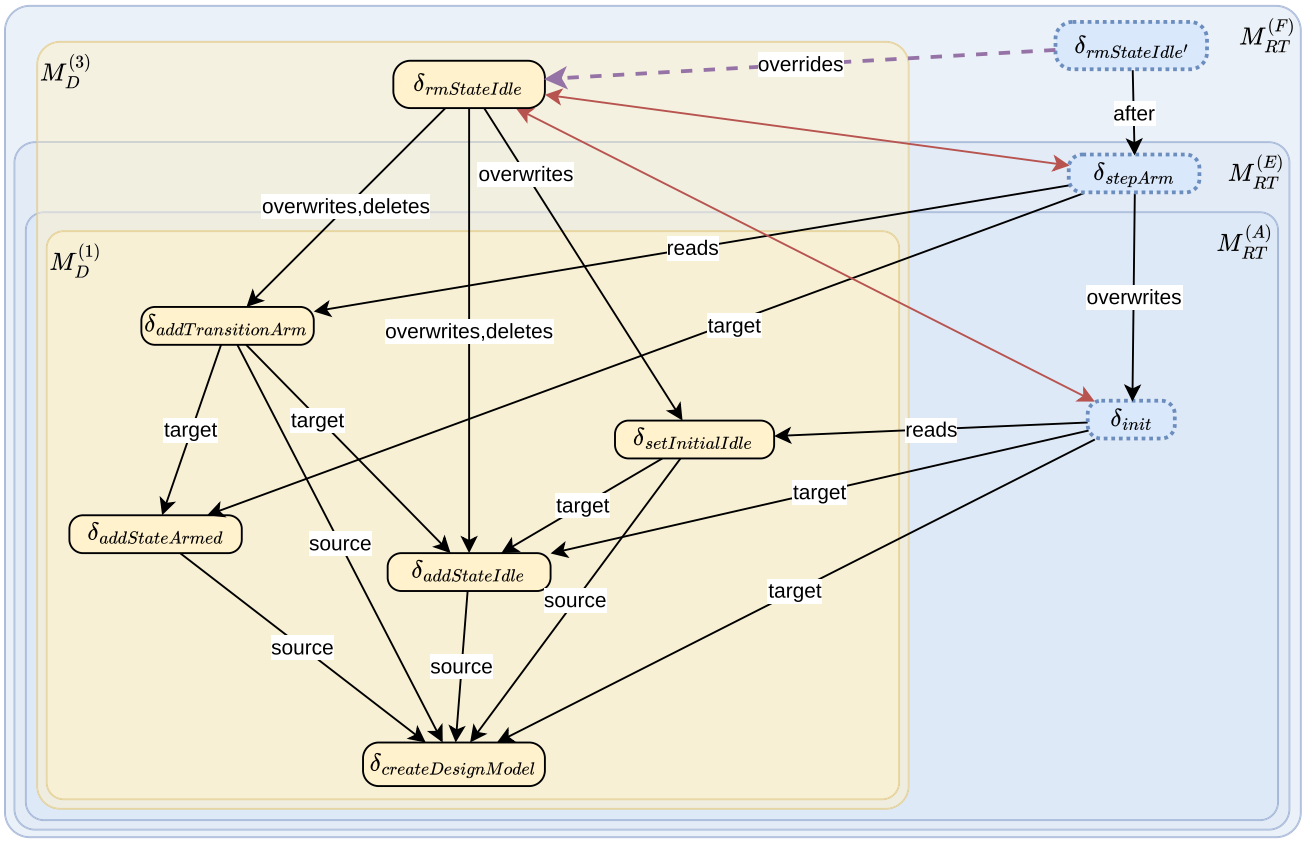

Fig. 14 Running example: deltas and dependencies

3.6 Graph embedding

We have seen that every run-time model version always contains a design model version. This hierarchical containment relationship comes naturally in many kinds of models, but also in projects, repositories, etc. For instance, an FSA can be part of a Statechart model (Fig. 6), which can in turn be part of a larger model that couples the Statechart with a Causal Block Diagram, which can be part of a project, which can be part of an organization's (mono-)repository.

We thus define an acyclic *embedding* relation between model *versions* that our versioning system can enforce: A guest model version M_G can be embedded in any number of host model versions M_{H_i}. The host depends on the guest, and the guest remains independent / "unaware" that it is embedded in a host. All model elements of the guest become part of the host. More precisely, all of the guest's deltas become part of the host. Deltas of the host may depend on deltas of the guest, but not the other way around. The only exception to this rule is the *overrides*-relation, discussed in the previous subsection. It allows the host to "inject" additional after-dependencies, to allow nodes in the guest to be deleted, without introducing a conflict if the host had an edge pointing to such a deleted node. Another example of overriding can be found in [12].

When host versions branch, their guests also branch. When merging host versions, their guests are also merged.

Running example Figure 15, a screenshot from our demo, shows a number of design—and run-time model versions. A black arrow (labeled with a delta) points to a version's *predecessor* or "parent" (git terminology). In other words, the black arrows go *against* the "arrow of time." A blue arrow represents an embedding relation: for instance, the version at the top, and the one below it, are run-time model versions, and they embed the same design model version. (Ignore that every version also has a blue arrow to itself with a label "runtime" or "design." This is just a hack, to visually distinguish between those two kinds of versions.)

Figure 16 shows the versions involved in our familiar scenario from Fig. 8. The fact that a delta from the design model was overridden (the removal of the "idle" state) is shown by the exclamation point (`design!`) on the rightmost embedding relation's label. This indicates to the user that the current execution trace cannot be replayed on the updated model (as in "recorded event"). Nevertheless, the user can simply continue her live modeling session (as in "real-time live modeling").

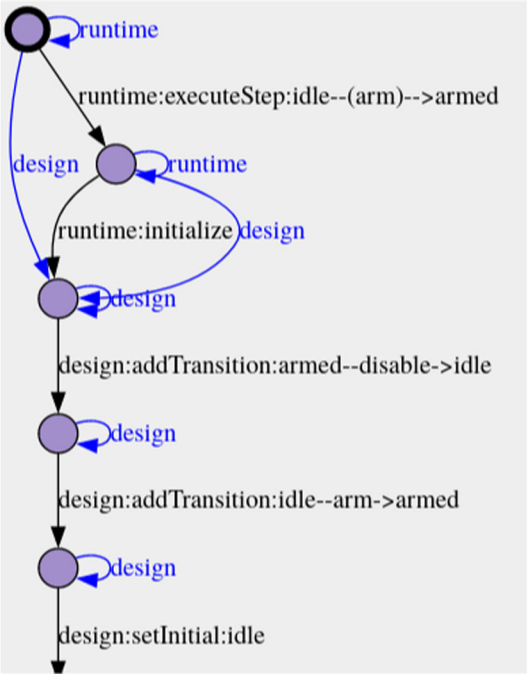

Fig. 15 Running example: embedding relation (screenshot)

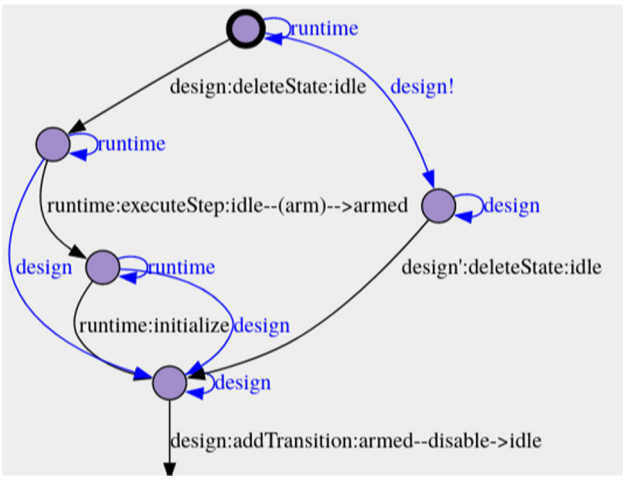

Fig. 16 Running example: runtime model version overrides a delta (screenshot)

3.7 Implementation

Our web-based demo implementation,[3] can run all of the scenarios from our running example (Figs. 1, 2, 3, 4, 5, 6, 7, 8). Figure 17 shows a screenshot of the demo.

To give the reader some idea of the implementation complexity of the live modeling actions of our FSA language, we include two (JavaScript) code fragments: Listing 1 is the handler that is called every time the user changes the initial state. Listing 2 is the handler called when the user initializes execution. Handlers not listed in this paper are: add/remove state/transition, execute step, change current state (god event), and abort execution. Each of these handlers uses the API of our graph versioning system to create and execute (a transaction of) deltas, resulting in a new design and/or runtime model version. For details on the implementation of the versioning system itself, we refer the reader to [12].

```
function onInitialStateChange(value) {
 // 'editDesignModel' creates a new version of the design model,
 // and attempts to merge it into the current run-time model version (if there is one)
 editDesignModel((graphState, modelNode) => {

   // create/overwrite the 'initial' edge:
   graphState.exec(modelNode.getDeltaForSetEdge(deltaRegistry, // <- source of the edge
     "initial", // <- label of the edge
     findState(graphState, value)?.creation || null)); // <- target of the edge

   // done!
   return {compositeLabel: "setInitial:"+value}; // <- human-readable description for this transaction
 });
}
```

Listing 1 Handler for setting/updating the initial state

```
function onInitialize() {
 // only initialize if a design model with an initial state exists:
 if (runtimeStuff.modelNode !== null && runtimeStuff.initial !== null) {
   // 'editRuntimeModel' simply creates a new version of the run-time model
   editRuntimeModel((graphState) => {
     // create a node, representing the run-time model:
     const runtimeId = generateUUID();
     const nodeCreation = deltaRegistry.newNodeCreation(runtimeId);
     graphState.exec(nodeCreation);

     // create a property, indicating that the newly created node represents a 'RuntimeModel':
     graphState.exec(deltaRegistry.newEdgeUpdate(
       nodeCreation.createOutgoingEdge("type"), // <- source and label of the edge
       "RuntimeModel")); // <- target of the edge

     // create edge pointing to the design model:
     graphState.exec(deltaRegistry.newEdgeUpdate(
       nodeCreation.createOutgoingEdge("model"), // <- source and label of the edge
       runtimeStuff.modelNode!.creation)); // <- target of the edge

     // create the 'current state'-pointer - which has a read dependency on 'initial':
     graphState.exec(deltaRegistry.newEdgeUpdate(
       nodeCreation.createOutgoingEdge("current"), // <- source and label of the edge
       runtimeStuff.initial!.creation, // <- target of the edge
       [(runtimeStuff.modelNode!.outgoingDeltas.get("initial")!.read())])); // <- read dependency

     // done!
     return {compositeLabel: "initialize"}; // <- human-readable description for this transaction
   });
 }
}
```

Listing 2 Handler for initializing execution

4 Extensions

4.1 Translational semantics

Our approach only deals with operational semantics, i.e., interpretation. With translational semantics, the design model is translated to a design model in another language, for which

[3] To run the demo, go to https://mstro.duckdns.org/public/onion/. The source code is also available: https://msdl.uantwerpen.be/git/jexelmans/onioncollab/.

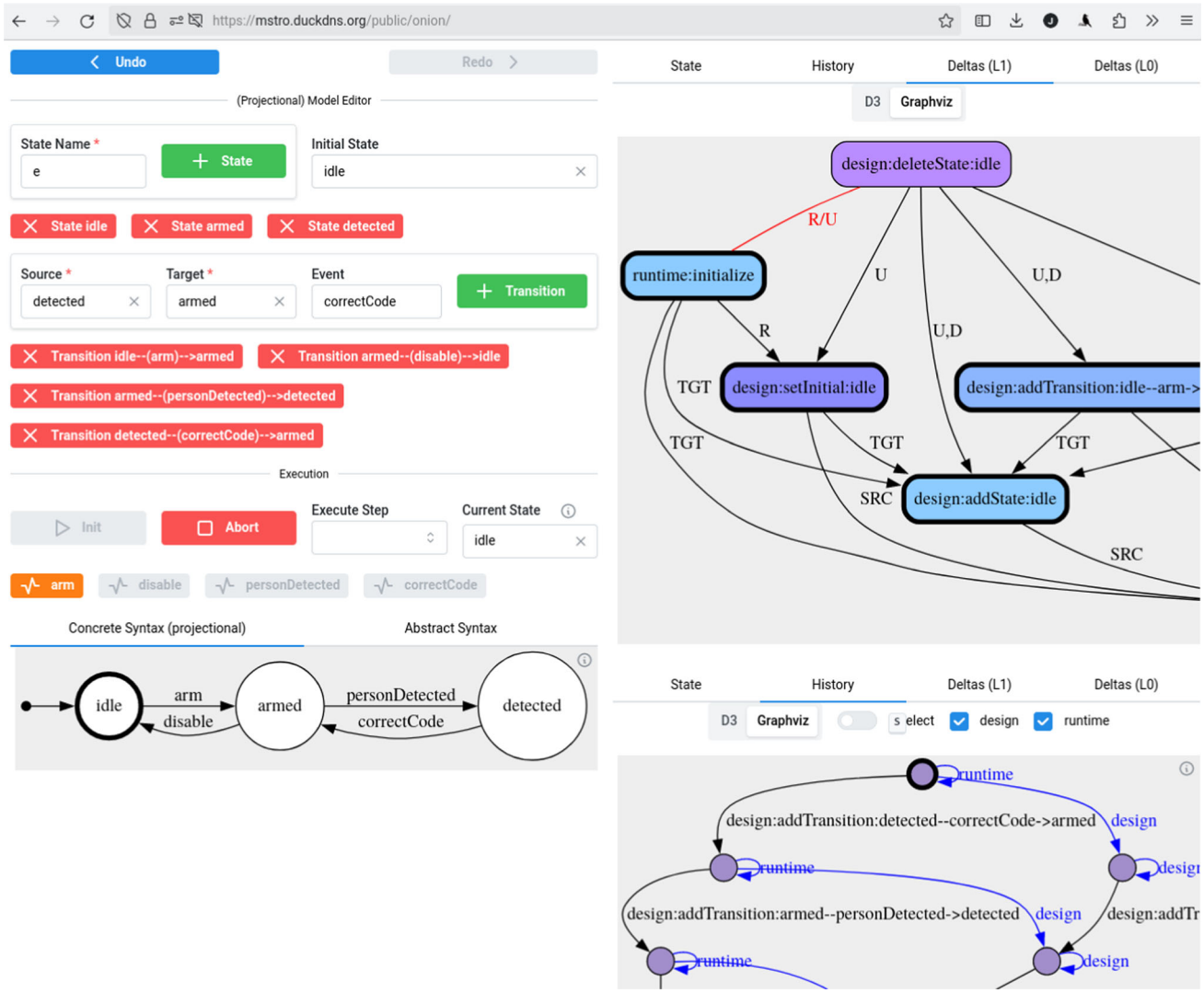

Fig. 17 Screenshot of our demo

an operational semantics is defined. For instance, when compiling an FSA to an action language, the action language may have an interpreter. If the action language has no interpreter, but is instead compiled to machine code, the machine itself can be seen as a (hardware) interpreter of machine code. The bottom line is that, in the end, there is always some kind of operational semantics at work.

Figure 18 shows a source design model (FSA) being transformed to a target design model (action language) for which an operational semantics is defined. If this transformation is incremental, a change to the source model results in a change in the target design model. This change can be merged into the target run-time model, following the approach explained in this paper. The incremental transformation can be implemented via an evolving (and versioned) correspondence model, as explained in our earlier work [12].

We think that support for translational semantics would be mainly useful in the debugging of model transformations (e.g., code generators), rather than the debugging of the (source) model itself. In the latter case, the mental gap between the design model in the source language, and the run-time model of the target language, would be difficult to overcome.

4.2 Detecting parallel independence

We have already seen an example of parallel independence in Fig. 6. By allowing the run-time model to branch when executing non-deterministic models, and by merging any non-conflicting deltas (on a data level), parallel independence can be detected (at run-time), and the size of the state space to explore (e.g., by a multiverse debugger) can be greatly reduced. Whether conflicts are correctly detected depends

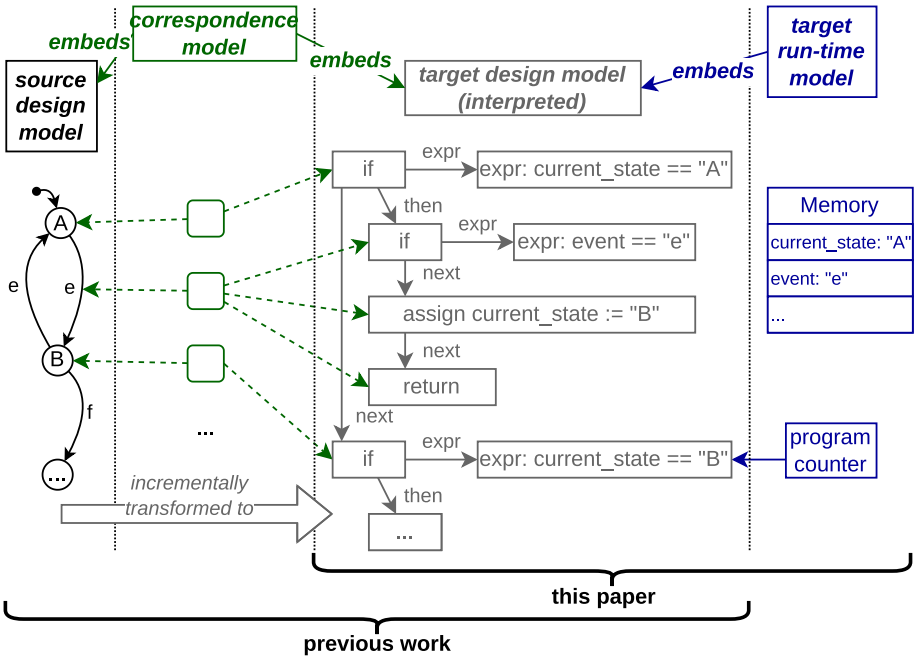

Fig. 18 Translational semantics (in this figure: code generation) could be an extension of our approach

on whether all read- and write dependencies of the execution steps are correctly recorded.

Incorrect detection of conflicts can be classified into two categories:

- *False positive* is when a conflict is detected, but there shouldn't be one.
- *False negative* is when a conflict should exist, but is "missed."

False positives blow up the state space, but do not result in incorrect merge results. False negatives result in incorrect merge results.

4.3 Serializing execution steps

Our versioning system is designed to make it possible for all changes to be serialized. As a consequence, not only edit operations, but also execution steps and even entire live modeling scenarios can be serialized, and communicated over a network connection (enabling collaborative debugging), or stored on disk (to suspend, and resume or analyze later).

5 Related work

Dynamic/reflective programming Dynamic/reflective programming languages have been around for some time. Lisp [13], Smalltalk [14], etc., allow both the program to introspect and modify itself. Smalltalk was the first language to be extended with an omniscient debugger [10, 11].

Erlang [15], a programming language that implements the Actor Model [16], can spawn new objects/*actors* that run updated code (i.e., code that did not exist when the program was started). This is similar to *dynamic linking* (Unix: loading of shared objects), but Erlang allows actors that run old code to keep running, in parallel with actors that run updated code. This solution fits well for actors (or objects) that have finite lifetimes: such actors eventually die (either in a controlled fashion, or by crashing), while being replaced by a new generation.

Despite the power of reflective languages, reflectivity cannot be "ported" to modeling languages in the general case, because some modeling languages lack the expressiveness to make use of a reflective API. The same holds for Erlang's flavor of live programming: many modeling languages have no notion of dynamically spawning new instances.

State migration (real-time live modeling) Our solution is strongly inspired by [8], which explicitly defines models for design and run-time. However, the "patching" of the run-time model is considered a language-specific ad hoc function, and changes to the models are not recorded, and therefore branching, merging and "recorded event"-live modeling are not supported.

In [17], the authors leverage a textual diffing algorithm to obtain model deltas which are used to update the running system. Traceability links are maintained between the textual concrete syntax and the run-time state. The authors propose a language-specific patch function to "fix" the run-time state

(e.g., in FSA, when the current state is deleted, go back to the initial state).

Cascade [18] is a meta-language that allows the language developer to explicitly define, as part of the language, edit operations that have pre- and post-conditions on the execution state. These conditions are specified in an operational manner. The focus is to automatically "fix" the run-time state when it "breaks" due to an edit operation. As in [17], deltas are sequential and do not support branching.

Another live modeling application prototype is presented in [19]. However, the delta types of the prototype are not generic, but specific to the modeling language of the prototype. The collaborative aspect is speculated upon, and the overlap with *versioning* (see also Fig. 19, discussed later) is mentioned, but no attempt is made to sketch a possible implementation.

Bagherzadeh et al. [20] presents a live modeling approach where UML-RT state machine models are transformed to C++ code. First, the source models are instrumented with additional transitions to support god events. Next, these models are transformed to C++ code that is implements certain constructs (from a C++ library) that support modification of the model at run-time. The authors support both "real-time" (state migration) and "recorded event"-style live modeling, but these features are implemented separately (ad hoc), rather than being supported via a unified mechanism, as in our approach. Non-determinism is not supported.

Ohshima et al. [21] presents a web-based live collaborative programming environment built on Croquet, a real-time shared experience platform. Croquet assumes good real-time network connectivity, and relies on a central server to provide a global ordering on events. Croquet's model does not support branching and merging. Rather, it is conceptually a single shared virtual machine among all users. Our approach, on the other hand, is fully decentralized, and supports branching and merging.

Restart and replay (recorded event live modeling) YinYang [22] is an interactive programming environment that re-runs the entire program after every edit operation. Program output (caused by print statements) is shown next to the program and can be traced back to the statement that caused the output, as well as the complete execution context (e.g., call stack, variable values) when the statement was executed.[4]

Similar to this, Hazel [23] is a functional programming language and editor that re-evaluates the entire program after every user edit. Hazel's unique feature is the ability of parse and evaluate incomplete expressions. For instance, when parsing "let $x = 5$ in $x*$," Hazel can figure out that at the end, a second value for the multiply-operator is missing, and will output "$5*?$."

Spreadsheet editors such as Microsoft Excel can be considered live modeling environments as well. Behind the scenes, a dependency graph between cells is maintained, and only cells whose dependency(ies) changed, are re-calculated in a cascading fashion. Circular dependencies are not allowed, unless "iterative calculation" is turned on, which will keep re-calculating values until they stabilize, or until some maximum number of iterations is exceeded. Just like our approach, Excel supports synchronous collaboration, but instead all computation is done on a central server [24].

The "restart and replay" approach is easy to implement, and gives an impression of liveness for deterministic programs that never receive input, but cannot scale to large (sequential) programs.

6 Conclusion

We presented a unifying foundation for collaborative editing, debugging and live model execution. We make design and run-time models explicit, encode them as graph structures, and record edit operations, execution steps and god events, all possibly concurrent, as inter-dependent deltas on those structures. Conflicts are detected efficiently and eagerly, and the essential complexity of reconciling design model changes with an ongoing run-time trace is reduced to a simple "merge"-operation.

We argue informally that the essence of our approach is correct for *causal* interpreted languages (meaning that every execution step can be traced back to a set of causes, which may be design model edit operations, or earlier execution steps). Our approach intends to capture all the causes of every execution step (as dependencies). A design model change can only impact an ongoing execution if it intersects with any of these causes. Such intersections are systematically identified as conflicts.

Figure 19 shows the "big picture" of the overlaps between versioning, live modeling and debugging. In this figure, it can be seen that versioning is orthogonal to (design-time) modeling and (run-time) execution. It is also clear that versioning overlaps with advanced debugger features: omniscient debuggers also maintain history (of execution), and multiverse debuggers support branching (of execution). With our proposed solution, we advocate to use a (sufficiently powerful) versioning system to track changes to both the design- and run-time-model, to support collaborative live modeling, omniscient debugging, and non-determinism.

[4] https://www.youtube.com/watch?v=01Xyoh-G6DE.

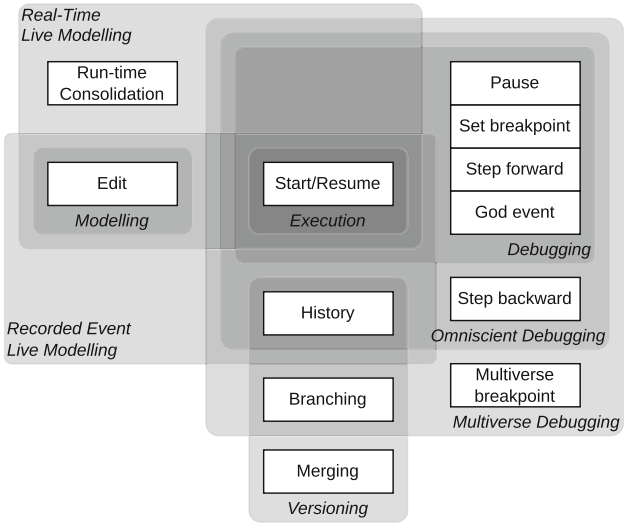

Fig. 19 Big picture: overlaps between versioning, debugging and live modeling

6.1 Limitations and future work

6.1.1 Language development

The steps that need to be taken to support a new language are: (1) On paper, work out a graph encoding of design—and run-time models. (2) List the design-time edit operations, and the run-time actions (e.g., firing a transition, receiving an input event, …) and possibly god events, that need to be supported. (3) Implement handlers for each of these actions, making calls to the graph API, similar to Listing 1. (4) Create a user interface for displaying the current version of the design and run-time models, as well as the actions that are currently enabled. React components for displaying deltas and their dependencies, and edit/execution history are already available in our repository, and can be reused.

When implementing the handlers (3), our graph API for creating deltas is fairly simple, and will figure out write-after-write dependencies automatically (because it knows which deltas most recently "touched" a node/edge). However, read-after-write dependencies currently need to be explicitly created in the handler code, and can be a source of errors if forgotten or done incorrectly. In principle, it must be possible to capture these dependencies automatically. The best approach would be to write the handlers in a domain-specific language (DSL) for graph transformation, rather than general-purpose code. With such a DSL, transformation rules execute in a "sandbox," and all the "inputs" of a transformation can be recorded as read-dependencies, whereas general-purpose code can have implicit dependencies (e.g., global variables), and even behave non-deterministically.

For the time being, porting existing languages and interpreters to use our approach will be of similar complexity as completely rewriting them. We have attempted to create a transparent data layer that seemingly allows manipulating JavaScript objects, while behind the scenes, recording all changes as deltas. We were unsuccessful, largely because of JavaScript's lack of power when it comes to transparently wrapping arbitrary objects/values.

6.1.2 Versioning core

Our VCS has a number of fundamental limitations, that we want to address. We currently have three separate, hard-coded graph structures: (1) graph state, (2) deltas & dependencies, (3) versions & parent links. We have concrete plans for their unification into a single, append-only graph structure in the near future. We believe this will result in a more compact implementation, while gaining power/expressiveness. Further, we want to scale up to the level of multi-user and multi-organization model management, including access control, following the principles of capability-based security.

Acknowledgements Author J. Exelmans is an SB PhD fellow at Fonds Wetenschappelijk Onderzoek–Vlaanderen (FWO) (1S70622N).

References

1. Norman, D.A.: Cognitive engineering. In: Norman, D.A., Draper, S.W. (eds.) User Centered System Design; New Perspectives on Human-Computer Interaction, pp. 31–62. L. Erlbaum Associates Inc., Mahwah (1986). Chap. 3
2. Lieberman, H., Fry, C.: Bridging the gulf between code and behavior in programming. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '95, pp. 480–486. ACM Press/Addison-Wesley Publishing Co., New York (1995). https://doi.org/10.1145/223904.223969
3. Kubelka, J., Robbes, R., Bergel, A.: The road to live programming: insights from the practice. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 1090–1101 (2018). https://doi.org/10.1145/3180155.3180200
4. Exelmans, J., Teodorov, C., Heinrich, R., Egyed, A., Vangheluwe, H.: Collaborative live modelling by language-agnostic versioning. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion,

Västerås, Sweden, October 1–6, 2023, pp. 364–374. IEEE, USA (2023). https://doi.org/10.1109/MODELS-C59198.2023.00067
5. Singh, R.G., Lopez, C.T., Marr, S., Boix, E.G., Scholliers, C.: Multiverse debugging: non-deterministic debugging for non-deterministic programs (artifact). Dagstuhl Artifacts Ser. **5**(2), 04–1043 (2019). https://doi.org/10.4230/DARTS.5.2.4
6. Teodorov, C.: G∀min∃: exploring the boundary between executable specification languages and behavior analysis tools. (G∀min∃ : Exploration de la Frontière Entre les Langages de Spécification exécutables et les outils d'analyse du comportement), (2023). https://tel.archives-ouvertes.fr/tel-04066483
7. Pietron, J., Raschke, A., Exelmans, J., Tichy, M.: Collaboration and versioning framework-a systematic top-down approach. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1–6, 2023, pp. 767–777. IEEE, USA (2023). https://doi.org/10.1109/MODELS-C59198.2023.00124
8. Van Tendeloo, Y., Van Mierlo, S., Vangheluwe, H.: A multi-paradigm modelling approach to live modelling. Softw. Syst. Model. **18**(5), 2821–2842 (2019). https://doi.org/10.1007/S10270-018-0700-7
9. Van Mierlo, S., Vangheluwe, H., Breslav, S., Goldstein, R., Khan, A.: Extending explicitly modelled simulation debugging environments with dynamic structure. ACM Trans. Model. Comput. Simul. **30**(1), 1–25 (2020). https://doi.org/10.1145/3338530
10. Pothier, G., Tanter, É.: Back to the future: omniscient debugging. IEEE Softw. **26**(6), 78–85 (2009). https://doi.org/10.1109/MS.2009.169
11. Lewis, B.: Debugging backwards in time. CoRR arXiv preprint arXiv:cs/0310016 (2003)
12. Exelmans, J., Pietron, J., Raschke, A., Vangheluwe, H., Tichy, M.: A new versioning approach for collaboration in blended modeling. J. Comput. Lang. **76**, 101221 (2023). https://doi.org/10.1016/J.COLA.2023.101221
13. Sandewall, E.: Programming in an interactive environment: the LISP experience. ACM Comput. Surv. **10**(1), 35–71 (1978). https://doi.org/10.1145/356715.356719
14. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading (1983)
15. Armstrong, J.: A history of erlang. In: Ryder, B.G., Hailpern, B. (eds.) Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9–10 June 2007, pp. 1–26. ACM, USA (2007). https://doi.org/10.1145/1238844.1238850
16. Agha, G.A.: ACTORS-a model of concurrent computation in distributed systems. In: MIT Press Series in Artificial Intelligence. MIT Press, Cambridge (1990)
17. Rozen, R., Storm, T.: Towards live domain-specific languages. Softw. Syst. Model. **18**(1), 195–212 (2019). https://doi.org/10.1007/s10270-017-0608-7
18. Rozen, R.: Cascade: a meta-language for change, cause and effect. In: Saraiva, J., Degueule, T., Scott, E. (eds.) Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, Cascais, Portugal, October 23–24, 2023, pp. 149–162. ACM, USA (2023). https://doi.org/10.1145/3623476.3623515
19. Storm, T.: Semantic deltas for live DSL environments. In: 2013 1st International Workshop on Live Programming (LIVE), pp. 35–38 (2013). https://doi.org/10.1109/LIVE.2013.6617347
20. Bagherzadeh, M., Jahed, K., Combemale, B., Dingel, J.: Live modeling in the context of state machine models and code generation. Softw. Syst. Model. **20**(3), 795–819 (2021). https://doi.org/10.1007/s10270-020-00829-y
21. Ohshima, Y., Lunzer, A., Evans, J., Freudenberg, V., Upton, B., Smith, D.A.: An experiment in live collaborative programming on the croquet shared experience platform. In: Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming. Programming '22, pp. 46–53. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3532512.3535224
22. McDirmid, S.: Usable live programming. In: Hosking, A.L., Eugster, P.T., Hirschfeld, R. (eds.) ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, Part of SPLASH '13, Indianapolis, IN, USA, October 26–31, 2013, pp. 53–62. ACM, USA (2013). https://doi.org/10.1145/2509578.2509585
23. Yuan, Y., Guest, S., Griffis, E., Potter, H., Moon, D., Omar, C.: Live pattern matching with typed holes. Proc. ACM Program. Lang. **7**(OOPSLA1), 609–635 (2023). https://doi.org/10.1145/3586048
24. Excel web services documentation. https://learn.microsoft.com/en-us/sharepoint/dev/general-development/excel-web-services. Accessed: 2024-06-28

Joeri Exelmans is a PhD student at the Antwerp Systems and Software Modelling (AnSyMo) group within the Department of Computer Science at the University of Antwerp in Belgium. The topic of his PhD is the modular composition of modeling languages and their (modeling, simulation, and debugging) tools. He is interested in model versioning and model management, decentralized software architectures and (pure) functional programming languages.

Ciprian Teodorov is a computer science researcher in Lab-STICC Laboratory/P4S team at ENSTA Bretagne. His main research interests are the industrialization of automata-based model checking techniques for embedded system verification and model-driven design tools for reconfigurable system-on-chip. He leads the OBP2 Semantic Diagnosis & Formal Verification research team. In the past he worked as EDA/CAD Software Engineer at Dolphin Integration, Meylan, France. His main responsibility was the implementation of a new, modern VHDL language infrastructure to increase standard compliance, to reduce memory consumption and to improve the simulation speed. Ciprian TEODOROV received a PhD degree in Computer Science from the University of Western Brittany, France. He was a member of the "Methods, tools for circuits and systems" (MOCS) team of Lab-STICC. His research work focused on physical-design tools for nanoscale computing architectures. During his PhD he created R2D NASIC, a nanoscale architectural template, based on the NASIC fabric, which enables arbitrary placement and routing at nanoscale. He also designed MoNaDe, a model-driven physical-design framework. MoNade enables agile and incremental exploration of the architecture/design tools adequacy.

Hans Vangheluwe is a Professor in the Antwerp Systems and Software Modelling (AnSyMo) group within the Department of Mathematics and Computer Science at the University of Antwerp in Belgium, where he is a founding member of the NEXOR Consortium on Cyber-Physical Systems (CPS). AnSyMo is a Core Research Lab of Flanders Make, the strategic research center for the Flemish manufacturing industry. He heads the Modelling, Simulation and Design Lab (MSDL), at the University of Antwerp. His fundamental work covers the foundations of modeling and simulation, of model management, model transformation, and domain-specific (visual) modeling environments. This work is always accompanied by prototype tools such as Python-PDEVS, the Modelverse, T-Core, AToM3 and AToMPM. He was the co-founder and coordinator of the EU ESPRIT Basic Research Working Group 8467 "Simulation in Europe," a founding member of the Modelica Design Team, and an advisor to national and international granting agencies in Europe and North America. He was the chair of the EU COST Action IC1404 Multi-Paradigm Modeling for Cyber-Physical Systems (MPM4CPS).