Constraint-based Run-time State Migration for Live Modeling
Ulyana Tikhonova, Jouke Stoel, Tijs van der Storm, Thomas Degueule

To cite this version:
Ulyana Tikhonova, Jouke Stoel, Tijs van der Storm, Thomas Degueule. Constraint-based Run-time State Migration for Live Modeling. SLE 2018 - 11th ACM SIGPLAN International Conference on Software Language Engineering, Nov 2018, Boston, MA, USA, United States. 10.1145/3276604.3276611 . hal-01896207

HAL Id: hal-01896207
https://hal.archives-ouvertes.fr/hal-01896207
Submitted on 15 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract
Live modeling enables modelers to incrementally update models as they are running and get immediate feedback about the impact of their changes. Changes introduced in a model may trigger inconsistencies between the model and its run-time state (e.g., deleting the current state in a state-machine); effectively requiring to migrate the run-time state to comply with the updated model. In this paper, we introduce an approach that enables to automatically migrate such run-time state based on declarative constraints defined by the language designer. We illustrate the approach using Nextep, a meta-modeling language for defining invariants and migration constraints on run-time state models. When a model changes, Nextep employs model finding techniques, backed by a solver, to automatically infer a new run-time model that satisfies the declared constraints. We apply Nextep to define migration strategies for two DSLs, and report on its expressiveness and performance.

CCS Concepts
- Software and its engineering → Domain specific languages; Software prototyping; • Theory of computation → Programming logic;

Keywords live modeling, run-time state migration, DSL, relational model finding

1 Introduction
Live modeling [27, 28] allows users of executable modeling languages to enjoy live, immediate feedback when editing their models, without having to restart execution. Live modeling has its roots in live programming, which allows developers to change program code and use the running application simultaneously, without long edit-compile-restart cycles [18, 21]. When the programmer changes the program code, its execution state is updated accordingly on-the-fly. This effectively bridges the “gulf of evaluation” [14] between changing a program and the impact of these changes.

While the value of live modeling and live programming is widely recognized [13, 14, 26], engineering live software languages requires a lot of effort and a deep understanding of the host language and its particular domain of application [4].

A central problem for live modeling languages is how to reconcile changes to a model with the run-time state of its execution. Changes to a model or program might invalidate the current run-time state. That is, when the executing model is modified, its run-time state still corresponds to the version before the change. The problem is analogous to migrating a database after a change to the database schema. So the question we address in this paper is how to migrate run-time state after a change to an executing model.

Earlier work has explored (re)constructing the required migration steps to the run-time state from the changes applied to the model [28]. Unfortunately, this requires intimate operational knowledge of how the change of the model itself is propagated to the runtime in the first place, leading to brittle, imperative, and non-modular code.

In this work, we employ model finding techniques [16] to migrate the run-time state of an executing model. A declarative, constraint-based specification of invariants and migration policies on the run-time state is input to a solver to infer...
a model that satisfies these constraints, given a representation of the old and new models, and the old run-time state. The resulting model is taken as the new run-time state of the updated version of the DSL program, and execution can continue.

We illustrate this approach using NexTep, a prototype meta-modeling language for defining the metamodel of both modeling language and run-time state model, and migration policies as constraints. NexTep specifications are then transformed to AlleAlle, a language for relational model-finding modulo theories. The AlleAlle specification is then processed by the Z3 [8] constraint solver to obtain a new consistent run-time state instance.

The remainder of this paper is organized as follows. In Section 2, we introduce a motivating example of live modeling and detail the associated challenges. In Section 3, we present a framework for structuring the constraints related to run-time state migration. In Section 4, we introduce NexTep, our prototype language for live modeling. In Section 5, we evaluate the performance and expressiveness of NexTep. Finally, we discuss related work in Section 6 and draw some conclusions in Section 7.

2 Motivating Example

Unlike General-Purpose Languages (GPL), DSLs capture the syntax and semantics of a specific domain. As a result, the cognitive distance between DSL models and their dynamic behavior can be arbitrarily large. Live modeling can thus be considered especially beneficial for DSLs because the dynamic effect of a change in a DSL program is harder to predict.

Throughout this paper, we use a simple finite-state machine (FSM) DSL as a running example to explain the ideas and challenges of live modeling, and to illustrate our approach. A program in this DSL is a particular machine that consists of a number of states and transitions. An example scenario of live modeling for the FSM DSL is depicted in Table 1. The starting point (Step #0) is a simple machine with two states closed and opened and transitions between them. We choose to model the run-time state of machines as the combination of two elements: the current state of execution current and a map visited that keeps track of how many times each state has been visited. Note that the choice of what is part of the run-time state and how it is specified is up to the language designer. In particular, it can be specified as an extension of the static metamodel or as a separate semantic domain [6]. Finally, we assume the FSM DSL is executed by a step-based interpreter such that the user can trigger events to trigger transitions, and consequently, modify the run-time state. Changing the model pauses the interpreter for an instant so that the new version can be reloaded.

At Step #0, the current state is initialized to the initial state closed and every counter is initialized to 0. At Step #1, following the first step of execution, the user triggers an event, which changes the current state to opened and increments the visited counter for closed by one. At Step #2, the user decides to insert a new state locked in the currently running machine. As depicted in Table 1, the question that arises is: what is the new run-time state at this point? Does the newly added state locked become the current state? How should locked be captured in the visited map?

These questions get even more complicated when considering the live modeling scenario depicted in Table 2. In Step #2 of this scenario, the user chooses to delete the state locked, which is at this point the current state of the machine. Here an obvious question is: which state should be the new current state of the updated machine? Should it be reverted to the initial state of the machine?

A generic live modeling engine cannot provide answers to these questions, because they are inherently domain-specific and should thus comply with domain-specific run-time state migration policies. Ultimately, it is the language designer’s responsibility to define these migration policies. We investigate how a language designer can express them in the next section.

3 Structuring Constraints for Run-time State Migration

When considering a situation of live modeling, as described in the previous section, we can distinguish two basic components: a static model (denoted \( p \)) and a run-time model

<table>
<thead>
<tr>
<th>Step</th>
<th>DSL program</th>
<th>Run-time state</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0 (initial)</td>
<td>closed, opened</td>
<td>current: closed, visited: closed ( \rightarrow ) 0, opened ( \rightarrow ) 0</td>
</tr>
<tr>
<td>#1 (interpreter step)</td>
<td>closed, opened</td>
<td>current: opened, visited: closed ( \rightarrow ) 1, opened ( \rightarrow ) 0</td>
</tr>
<tr>
<td>#2 (user edit)</td>
<td>closed, opened, locked</td>
<td>current: ???, visited: ???</td>
</tr>
</tbody>
</table>
Table 2. Live modeling scenario ‘Remove Current State’ for the FSM DSL

<table>
<thead>
<tr>
<th>Step</th>
<th>DSL program</th>
<th>Run-time state</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0 (initial)</td>
<td>closed opened locked</td>
<td>current: closed visited: closed (\Rightarrow) 0 opened (\Rightarrow) 0 locked (\Rightarrow) 0</td>
</tr>
<tr>
<td>#1 (interpreter step)</td>
<td>closed opened locked</td>
<td>current: locked visited: closed (\Rightarrow) 1 opened (\Rightarrow) 0 locked (\Rightarrow) 0</td>
</tr>
<tr>
<td>#2 (user edit)</td>
<td>closed opened locked</td>
<td>current: ??? visited: ???</td>
</tr>
</tbody>
</table>

(denoted \(x\)). The static model captures the structure (i.e., code) of the DSL program.\(^1\) The run-time model captures the state of the execution of the DSL program. As we focus on the situation in which a DSL program is changed during its execution, we distinguish two versions of the static and run-time models: \(p\) and \(x\) before the program has been changed, and \(p'\) and \(x'\) after the program has been changed. Thus, we can reformulate the problem of finding a new run-time state as follows: given that \(p\), \(x\), and \(p'\) are known, how do we find \(x'\)?

A potential approach would be to modify the model interpreter to repair the run-time state imperatively whenever the program is changed. Unfortunately, as we discuss in more detail in Section 5.3, this approach is error-prone and non-modular. In this paper, we apply model-finding techniques to infer the run-time state based on a set of declarative constraints. Concretely this means identifying requirements techniques to infer the run-time state based on a set of declarative constraints.

**Semantic relations** \(R(p', x')\): these constraints ensure that the new run-time state \(x'\) is semantically consistent with the current (new) version of the DSL model \(p'\). Note though that there might be multiple—possibly infinite—\(x'\) that are semantically consistent with \(p'\).

\(^1\)Although according to the main idea of live modeling the static model is not static anymore, we still use the term static for the sake of the terminology legacy.

Migration rules \(M(p, x, p', x')\): these constraints provide the possibility to take into account what has changed in the DSL model and project this change onto the run-time state. Migration rules restrict the choice of (semantically consistent) \(x'\) in preference for those \(x'\) that reflect the change that has happened to the DSL (static) model.

The definition of a solution space by semantic relations \(R(p', x')\) and its filtering by migration rules \(M(p, x, p', x')\) is illustrated by the Venn diagram depicted in Figure 1. Although migration rules restrict the choice of possible solutions, there still can be many (potentially infinite) \(x'\) that satisfy the semantic constraints. Selecting an arbitrary element of this set as the new run-time state for the updated DSL program might not result in an intuitive new run-time state. For instance, in our previously described FSM scenario ‘Add New State’ (Table 1), it is perfectly valid to reset the current to opened, and set the visited entities to 1000. Although this would be a correct run-time state (i.e., a valid solution to our set of constraints), it would only add confusion to the user of the DSL.

To find the most suitable new run-time state, we need to introduce a heuristic for selecting one of potentially many solutions. We represent such a heuristic as \(H(x')\) in Figure 1.

**Migration rules** \(M(p, x, p', x')\): these constraints provide the possibility to take into account what has changed in the DSL model and project this change onto the run-time state. Migration rules restrict the choice of (semantically consistent) \(x'\) in preference for those \(x'\) that reflect the change that has happened to the DSL (static) model.

The definition of a solution space by semantic relations \(R(p', x')\) and its filtering by migration rules \(M(p, x, p', x')\) is illustrated by the Venn diagram depicted in Figure 1. Although migration rules restrict the choice of possible solutions, there still can be many (potentially infinite) \(x'\) that satisfy the semantic constraints. Selecting an arbitrary element of this set as the new run-time state for the updated DSL program might not result in an intuitive new run-time state. For instance, in our previously described FSM scenario ‘Add New State’ (Table 1), it is perfectly valid to reset the current to opened, and set the visited entities to 1000. Although this would be a correct run-time state (i.e., a valid solution to our set of constraints), it would only add confusion to the user of the DSL.

To find the most suitable new run-time state, we need to introduce a heuristic for selecting one of potentially many solutions. We represent such a heuristic as \(H(x')\) in Figure 1.

**Migration rules** \(M(p, x, p', x')\): these constraints provide the possibility to take into account what has changed in the DSL model and project this change onto the run-time state. Migration rules restrict the choice of (semantically consistent) \(x'\) in preference for those \(x'\) that reflect the change that has happened to the DSL (static) model.

The definition of a solution space by semantic relations \(R(p', x')\) and its filtering by migration rules \(M(p, x, p', x')\) is illustrated by the Venn diagram depicted in Figure 1. Although migration rules restrict the choice of possible solutions, there still can be many (potentially infinite) \(x'\) that satisfy the semantic constraints. Selecting an arbitrary element of this set as the new run-time state for the updated DSL program might not result in an intuitive new run-time state. For instance, in our previously described FSM scenario ‘Add New State’ (Table 1), it is perfectly valid to reset the current to opened, and set the visited entities to 1000. Although this would be a correct run-time state (i.e., a valid solution to our set of constraints), it would only add confusion to the user of the DSL.

To find the most suitable new run-time state, we need to introduce a heuristic for selecting one of potentially many solutions. We represent such a heuristic as \(H(x')\) in Figure 1. What are good heuristics may depend on many variables related to domain-specific semantics and usability. One example of a heuristic that we will use in the remainder of this paper is the minimum distance \(D(x, x')\). The minimum distance heuristic requires that the new run-time state \(x'\) should be as close as possible to the previous run-time state \(x\). Intuitively, this heuristic captures the principle of “first, do no harm” (i.e., do not break parts of the run-time state that do not have to change), and the “principle of least surprise” (i.e., prefer small incremental changes over invasive ones). In this way, we support the incremental process of editing a DSL model, providing live feedback in relation to the previous run-time state of the DSL program.
To apply constraint-based run-time state migration, language designers must specify the semantic relations \( R(p', x') \) and migration rules \( M(p, x, p', x') \). Semantic relations can be automatically extracted from the DSL definition itself (meta-model, static and dynamic semantics). Migration constraints are explicitly specified to relate \( p, p', x \), on the one hand, and \( x' \), on the other hand.

In the case of the FSM DSL, if the user chooses to delete the state pointed to by \( current \) (Table 2), a possible strategy is to reset \( current \) to the initial state of the machine. Another strategy is to set the \( current \) to the state that was the “closest” to the deleted state. Both policies are equally valid: this is the designer’s responsibility to make a choice. In contrast, the heuristic \( H(x') \) is domain-agnostic and can be built into the live modeling tool itself.

Note that our approach infers the new run-time state \( (x') \) only based on the directly preceding step of the execution \( (x) \), i.e., migration constraints cannot be defined in terms of the full execution trace. For example, in our FSM DSL we cannot assign the new \( current \) to a state that had been \( current \) previously in the execution. Taking the execution trace into account is an essential direction for further research.

4 **Nextep: a Language for State Migration**

In this section we describe our proof-of-concept implementation of the approach described in Section 3, the **Nextep** language. In particular, we discuss all described constraints in detail and explain their implementation in **Nextep** for our running example of the FSM DSL.

4.1 Syntax of Nextep

The **Nextep** language allows for formulating preferences for (i.e., for configuring) live modeling in the form of a set of constraints. As described earlier, such constraints are expressed using the constructs of the static and run-time models of a DSL. For this, **Nextep** uses a simplified version of the Ecore metamodeling language to express classes and references between them. An example **Nextep** definition for the statemachine DSL is depicted in Listing 1.

A **Nextep** configuration consists of three parts: **static**, **runtime**, and **migration**. The **static** part (lines 1–10 in Listing 1) describes the static model of the DSL and, in essence, corresponds to (the core of) the DSL metamodel. In our example, it consists of three classes (Machine, State, and Trans) and four references (states, initial, transitions, and target).

References can be of two types: referencing exactly one object (for example, target: State, line 10 in Listing 1) or referencing a set of objects (such as states: State*, line 3 in Listing 1). However, this is not a conceptual limitation of **Nextep**, but rather a simplification for the sake of demonstration. For the sake of conciseness, we also omit static constraints in the statemachine metamodel from the **Nextep** specification, and assume that these are dealt with by other means, such as parsers or type checkers.

The **runtime** part of a **Nextep** definition (lines 12–31 in Listing 1) describes the run-time model of the DSL and the semantic relations \( R(p', x') \). The run-time model is defined in the same way as the static model, using classes and references. The semantic relations are defined in the form of invariants attached to these classes.

The class Runtime is a root class that stores references to all components that constitute both the static and the run-time models of the DSL. For the FSM DSL, these are machine for the static model, and current and visited for the run-time model (lines 14–16 in Listing 1). This way, the **Runtime** class defines the scope for the semantic relations \( R(p', x') \). We define the following semantic relations for the FSM DSL:

- the current state of the machine is one of its states (line 19 in Listing 1);
- the number of visits is defined exactly once for each state of the statemachine (lines 21–22 in Listing 1);

Listing 1. Nextep specification of the FSM DSL
• different entries in visited correspond to different states of the statemachine, visited is a function (lines 24–25 in Listing 1);
• the number of visits for each state is greater than or equal to zero (line 31 in Listing 1).

The migration part of a Nextep definition (lines 33–35 in Listing 1) describes the migration rules \( M(p, x, p', x') \). Here the keywords old and new denote the instances of the Runtime class corresponding to the old and new versions of the run-time model, respectively. Since the Runtime class refers to the static model, this also provides access to \( p \) and \( p' \). For the FSM DSL, we define the following migration rule: if the current state has been deleted (i.e., \( \text{not (old \ current in new . machine . states) } \)), then the new current state is assigned to the initial state (i.e., \( \text{new . current = new . machine . initial} \)).

4.2 Semantics of Nextep

In order to implement model finding for our live DSLs, we translate a Nextep definition of a DSL and its static and run-time models into AlleAlle. AlleAlle\(^2\) is a bounded relational model finder. That is, AlleAlle is a formalism that allows for expressing a model using a combination of first order logic and relational algebra and supports automatic construction of relational instances of such a model in a bounded scope (i.e., limited set of cases) based on the constraints of the model. In addition, AlleAlle supports declaration of optimization criteria making it suitable for solving optimization problems next to pure satisfiability problems.

As a back-end AlleAlle uses Z3, an SMT (Satisfiability Modulo Theories) solver [8].\(^3\) That is, AlleAlle translates a model expressed in its input language into a corresponding SMT formula and then invokes Z3 to find solution(s) to this formula. The solution is then translated back into an instance of the relational model.

AlleAlle is comparable to the Alloy language and analyzer [12].\(^4\) The major difference is that, unlike Alloy which is built on top of a SAT solver, AlleAlle is built on top of the Z3 SMT solver, and thus, supports unbounded integer numbers and allows for optimization criteria, which are used by the solver to find the most suitable (optimal) solution.

In this section, we describe the semantics of Nextep in the form of a translation to AlleAlle. We illustrate our description using an excerpt of the AlleAlle specification (see Listing 2) that corresponds to the Nextep specification described in the previous section (Listing 1).

Structure translation. All structural elements of both static and dynamic parts of a Nextep definition are translated to relations in AlleAlle:

```plaintext
Listing 2. An excerpt of the generated AlleAlle code for the statemachine DSL.

- Nextep classes are translated to AlleAlle unary relations;
- Nextep references are translated to AlleAlle binary relations.

As the resulting AlleAlle specification is used for solving our initial problem “given \( p, x, \) and \( p' \), what is \( x'? \)” all the listed relations are generated for the concrete instances of \( p, x, p' \), and \( x' \).

For example, in Listing 2 relations with \( pn_ \) as their prefix (lines 1–5) correspond to the new version of the program \( p' \); relations with \( xo_ \) as their prefix (lines 7–8) correspond to the previous run-time state \( x \); and relations with \( xn_ \) as their prefix (lines 10–13) correspond to the new run-time state \( x' \).

Bounds translation. As mentioned earlier, bounds are used to restrict the scope of search for the back-end solver. In particular, bounds introduce a set of atoms available for the instantiation of relations in an AlleAlle specification. In the context of Nextep, \( p, x, \) and \( p' \) are known and, thus, determine the exact bounds for the corresponding AlleAlle relations. In other words, each of the relations that represent the original \( p, x, \) and \( p' \) are assigned specific values (i.e., tuples) corresponding to the concrete instances of \( p, x, \) and \( p' \) (lines 1–9 in Listing 2, on the right side of the “\( = \)” symbol).

While \( p, x, \) and \( p' \) are known, the new run-time model \( x' \) is what we are searching for. Therefore, bounds for the AlleAlle relations representing \( x' \) define an extended search scope, not limited to the concrete instances of \( p, x, \) and \( p' \).

---
\(^2\)http://github.com/cwi-swat/allealle
\(^3\)http://github.com/Z3Prover/z3
\(^4\)http://alloytools.org
particular, the $x'$ relations are constrained by an upper bound (lines 10–14 in Listing 2, on the right side of the ‘$\leq$’ symbol). The upper bounds include the atoms from the instances of $p$, $x$, and $p'$, and a finite number of auxiliary atoms of the same type. For instance, in the example, this means that there is one extra machine $m1$ and five extra states $s1..s5$.

**Constraints translation.** Semantic relations and migration rules of a Nextep definition are translated to the corresponding AlleAlle constraints (formulas) instantiated for $p$, $x$, $p'$, and $x'$ (lines 18–24 in Listing 2). Moreover, additional AlleAlle constraints are generated to capture the structural properties of (static and runtime) classes: types and multiplicities of references (line 16 in Listing 2). The expressive power of Nextep constraints is determined by the expressive power of AlleAlle formulas, which includes relational algebra extended with transitive closure. Thus, Nextep supports set theory notation, first-order predicates, (reflexive) transitive closure, and unbounded integer arithmetic. For object navigation we use the standard dot notation.

**Minimum distance.** As described earlier in Section 3, the heuristic for the minimum distance $D(x, x')$ is not specific to each particular DSL. In particular, in AlleAlle, the minimum distance heuristic is generically represented as optimization criteria. For this, $D(x, x')$ is represented as a set of metrics in AlleAlle, over which the minimization criteria are applied (lines 26–28 in Listing 2).

Minimization is performed over the following metrics:

- the number of elements in the difference of two sets representing $x$ and $x'$ correspondingly (such as current in the statemachine DSL);
- the sum of absolute differences between integer values of two vectors representing $x$ and $x'$ correspondingly (such as visited in the statemachine DSL).

As a result, for instance, not modifying the current state is preferred over modifying it, because in the former case the set difference will be the empty set, which is minimal. Similarly, minimization of integer values ensures that no arbitrary values will be put in the map visited.

### 4.3 Output of Nextep

When applied to the two live modeling scenarios described in Section 2, Nextep produces the following results. Tables 3 and 4 show the instances of $p$, $x$, and $p'$ and the corresponding $x'$ for the scenarios of Table 1 and Table 2, respectively.

Here, the instances of $x'$ (right bottom corner in both tables) represent the solutions found by the solver. They are constructed by mapping an instance of the relational model (calculated by Z3 and translated to AlleAlle) on the corresponding instance of the Nextep run-time model.

As can be seen in Table 3, the addition of the locked state does not change the runtime state, except for adding a new entry to the visited map, initialized to zero. However, if the current state is removed (i.e., locked in Table 4), the current field is reset to the initial state, and the locked entry is removed from the visited map.

The obtained results are determined by the domain-specific migration policies in the Nextep definition (in Listing 1). For example, if we remove the migration rule from our Nextep definition, the calculated $x'$ for the scenario ‘Remove Current State’ would be some arbitrary state (e.g., opened), instead of the initial state of the statemachine.

### 5 Evaluation

In this section, we use the Nextep implementation for an early evaluation of our approach. In Section 5.1, we perform a quantitative evaluation to estimate the performance of Nextep. In Section 5.2, we perform a qualitative evaluation to challenge the expressive power of Nextep by applying it to a DSL for robotic arm control. Finally, in Section 5.3, we conceptually compare Nextep with the manual encoding of migration strategies as presented in previous work [28].
5.1 Statemachine DSL Case Study

To evaluate the performance of our method, we ran two different scenarios for the FSM example.\footnote{The source code of our benchmark experiments can be found at https://github.com/cwi-swat/live-modeling/tree/master/nextstep/src/benchmark/statemachine}

In the first scenario, we tested the live modeling situation where one extra state and two new transitions are added to the new program. The corresponding new run-time state has its visited map extended with an entry for the newly added state assigned to ‘0’ visits. In the second scenario, we tested the case where the current state is removed from the statemachine. This forces the new current to be reset to the initial keeping all the values for the visited map intact, minus the removed state.

In both scenarios, we gradually increased the number of states in the statemachine to assess the impact of model size on performance. The results are shown in Figure 2. The execution times reported in the figure include the following three phases of our implementation: (1) translation of the Nextep specification to the relational specification, (2) translation of the relational specification to SMT constraints (from AlleAlle to SMTLIB), and (3) solving (model finding) by Z3.

We ran the benchmarks on an early 2015 model MacBook Pro with a 2.7 GHz quad core Intel processor with 8 GB of DDR3 RAM. We ran each configuration of the benchmark 10 times, ensuring that each execution is independent of the other (i.e., no shared memory, cache, etc.). Figure 2 reports the mean time per configuration of each of the three different phases.

The translation times reported in both scenarios show a similar pattern. The translation of Nextep specifications to AlleAlle specifications in the different configurations is fairly stable time-wise with a minimum of 943 ms and a maximum of 1529 ms.

The translation times from AlleAlle to SMTLIB also follow a similar trend in both scenarios: the translation time goes up as the Nextep specification grows. An explanation is that increasing the size of Nextep specifications causes the number of tuples in the AlleAlle relational models to increase as well. Then, the more tuples in an AlleAlle specification, the longer it takes to translate from AlleAlle to SMTLIB. This is especially the case when quantifiers are used in the AlleAlle specifications—and Nextep relies heavily on these quantifiers.

The solving times reported for Z3 tell a different story. For many configurations, Z3 was not able to find an optimal solution within a time limit of 3 seconds. We empirically set this limit to 3 seconds after observing that Z3 either manages to find a solution within a second or quickly explodes to multiple minutes. Since the time frame of multiple minutes is not in line with the spirit of live modeling, we decided to consider all solving times exceeding 3 seconds as timeouts.

For the first scenario, Z3 timed out for 9 configurations out of 19. For the second scenario, it timed out for 13 configurations out of 19.

Figure 2. The Statemachine Case Study benchmark results
**Discussion**  Our benchmark results suggest that there is much room for improvement in our prototype. These improvements can be classified according to the different phases of our method: improving the translation from **Nextep** to **AlleAlle**, improving the translation from **AlleAlle** to **SMTLIB**, and improving the solving times by **Z3**.

A first improvement would be to optimize the translation from **Nextep** to **AlleAlle**. As mentioned earlier, the current implementation of **Nextep** is an early prototype that has not been aggressively optimized yet. Both the performance of the translation and the generated **AlleAlle** specifications can be improved. In particular, improving the latter would help in speeding up the total execution time of our method. To verify this intuition, we optimized some **AlleAlle** specifications manually, bypassing the automatic translation phase of our approach while keeping the exact semantics of the original **Nextep** specification, and observed substantial improvements on performance. Generating an optimized **AlleAlle** specification would help in minimizing the translation time from **AlleAlle** to **SMTLIB** as well, as we have observed that translation times decrease when the **AlleAlle** specification shrinks. Finally, generating optimized **AlleAlle** specifications would also help **Z3** to find optimal solutions quicker: the smaller the **AlleAlle** specification the less variables are declared in the resulting SMT formula, often making it easier for the solver to find optimal solutions.

Next to optimizing **Nextep**, we could look at optimizing **AlleAlle**. The current implementation of **AlleAlle** is also in a prototype phase. Early experiments with an optimized native Java version of the **AlleAlle** model finder suggest that translation times can potentially be improved by a factor 10 to 15.

Improving the stability of the solving times of **Z3** is the hardest challenge which for now is considered an open question. As mentioned earlier, generating optimized **AlleAlle** specifications will help to improve the solving times for **Z3**. However, whether solving time can be predicted upfront from a given **Nextep** specification remains an open question.

### 5.2 Robotic Arm DSL Case Study

The **Robotic Arm DSL** is used for controlling industrial robots that consist of various subsystems (i.e., mechanical and/or electrical parts). For example, a robot can consist of a **hand**, responsible for manipulating objects, an **arm**, responsible for moving the hand to a particular position, and smaller subsystems like **triggers** and **actuators**. A program in the Robotic Arm DSL describes the coordination of subsystems of a robot during its execution. An example of such a program is shown in the middle column of Table 5.

To perform a certain task, a robot executes specific actions on its subsystems. Each subsystem can execute its actions independently from other subsystems of the robot. However, some of these actions should be executed in certain order, or in other words, some actions have to wait for other actions to be executed first; such dependencies are shown as arrows in Robotic Arm programs. For example, the subsystem **hand** should not execute the action **grab** before the subsystem **arm** has executed the action **down** (see the DSL program in the second row of Table 5).

The semantics of the Robotic Arm DSL maps a Robotic Arm program to the following execution behavior. In order to execute actions independently, each subsystem maintains a queue of actions that should be executed. First, all actions are collected in one **Request** pool (Step 0 in Table 5). Then, the

<table>
<thead>
<tr>
<th>Step</th>
<th>DSL program</th>
<th>Run-time state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>arm hand</td>
<td>Request: {grab, down, up} Queues: arm → [ ] hand ↔ [ ]</td>
</tr>
<tr>
<td></td>
<td>down grab up</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>arm hand</td>
<td>Request: {grab, up} Queues: arm → [down] hand ↔ [ ]</td>
</tr>
<tr>
<td></td>
<td>down grab up</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>arm hand</td>
<td>Request: {grab} Queues: arm → [down, up] hand ↔ [ ]</td>
</tr>
<tr>
<td></td>
<td>down grab up</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>arm hand</td>
<td>Request: {grab, switch, up} Queues: arm → [down] hand ↔ [ ] trigger ↔ [ ]</td>
</tr>
<tr>
<td></td>
<td>down grab up</td>
<td></td>
</tr>
</tbody>
</table>

in other words, some actions have to wait for other actions to be executed first; such dependencies are shown as arrows in Robotic Arm programs. For example, the subsystem **hand** should not execute the action **grab** before the subsystem **arm** has executed the action **down** (see the DSL program in the second row of Table 5).

The semantics of the Robotic Arm DSL maps a Robotic Arm program to the following execution behavior. In order to execute actions independently, each subsystem maintains a queue of actions that should be executed. First, all actions are collected in one **Request** pool (Step 0 in Table 5). Then, the
actions are taken from the Request one by one and pushed to the corresponding subsystem Queue (Steps 1–2 in Table 5). When an action is executed, it is removed from the queue. To enforce the ordering of actions, an action cannot be queued until all of the actions that this action depends on are in the Request pool or in the Queues. An exception to this constraint is the situation in which both ordered actions belong to the same subsystem. Then the actions can be queued in the corresponding order. For example, in Step 2 of Table 5, both actions down and up are queued, as they belong to the same subsystem arm.

The bottom row of Table 5 introduces a live modeling step for the Robotic Arm DSL. In particular, interrupting the normal execution of the Robotic Arm program (Steps 0–2), in Step 3 the user updates the program by adding a new subsystem trigger with a new action switch and a new dependency between the actions up and switch. As a result of this update, an obvious migration strategy for the run-time state would be to add a new queue for the subsystem trigger. It is not clear though what to do with the action switch. According to the semantics of the Robotic Arm DSL, the action switch can be executed independently by trigger, and thus, does not need to appear in the Request or Queues. However, intuition suggests that a newly added action should (explicitly) show up in the execution. Therefore, we introduce a migration rule that puts newly added actions to the Request.

The Robotic Arm NEXTEP definition that captures the described semantic relations and specifies the stated migration rule is depicted in Listing 3. The new run-time state calculated according to this NEXTEP definition is shown in the bottom right cell of Table 5. Note that in this new run-time state, the newly added action switch is added to the Request and, as a result, the action up is moved (back) to the Request. This allows for observing the execution order according to the newly added dependency.

The NEXTEP definition of Listing 3 defines classes for all constructs introduced above: Task for a Robotic Arm program, Subsystem for a robot subsystem, Action for a subsystem action, OrderedPair to represent order dependencies, Queue and QueuedAction to model a subsystem queue, and Runtime for the run-time state.

The semantics of the Robotic Arm DSL is captured by the following semantic relations:

1. a Queue contains only actions of its corresponding subsystem (line 21);
2. a request contains only actions of the current task (i.e., program being executed) (line 33);
3. if an action of a subsystem appears in the task, then there should be a corresponding queue for this subsystem (lines 35–36);

Listing 3. NEXTEP definition for the Robotic Arm DSL
4. queues and request do not intersect (line 38), i.e., when an action is queued, it is removed from the request;
5. an action, that succeeds another action which is stored in the request, should be also in the request, i.e., should not be queued yet (lines 40–41);
6. an action, that succeeds an action stored in a queue, should not be queued yet, unless these actions belong to the same subsystem (lines 43–46);
7. if two ordered actions belong to the same subsystem queue, then their indexes correspond to the order of the actions (lines 48–55).

Finally, the result that we get for the scenario of Table 5 is to a large extent determined by the migration rule specified in lines 57–59 of Listing 3 according to our earlier design decision. In particular, the constraint states that all actions from the new task (new.task.actions) that are not in the old task (old.task.actions) should be added to the request (in new.request.actions).

To conclude, in this section we demonstrated how NEXTep can be applied for configuring live modeling for a DSL with a more complicated semantics than our running example of the FSM DSL. The obtained result (i.e., the calculated new run-time state) corresponds to our expectations about the semantic consistency of the DSL and to our intuition about migration strategy of the run-time state.

5.3 Comparison to Manual Migration
Earlier work in live modeling explored an operational way of encoding migration strategies, called RMPatch [28]. RMPatch consists of the following steps:
1. after a user changes a DSL program (static model), a delta is computed using a model-based differencing algorithm;
2. a patch corresponding to the delta is then applied to the run-time model, which is a copy of the static model extended with the necessary run-time data;
3. the code that applies the patch to the run-time model is specialized using inheritance and updates the run-time data.

One of the take-aways of the experiment with RMPatch is that even for simple examples such as the statemachine language, it is quite hard to correctly implement migrations, and the code quickly becomes unwieldy. Below we consider the main drawbacks of manually coded migrations in contrast with NEXTep.

**Coupling between model and run-time state.** RMPatch requires that run-time state is an instance of the runtime meta-model which is a proper extension of the static meta-model of the language. For instance, the State class defines the number of visits as its own field. Similarly, the class Machine contains the reference to the current state. Using NEXTep the representation of a run-time state is decoupled from the meta-model, thus allowing for more flexibility for languages where the relation between model and run-time state is less clear cut.

**Operational instead of declarative.** Second, using program code to encode migration policies requires careful scheduling of operations, since there might be dependencies between modification incurred by the user’s model change and migration side effects. In contrast, NEXTep’s migration rules are defined as declarative constraints, thus allowing for abstracting away from any ordering constraints.

**Hard to reason about correctness.** The third problem of encoding migration policies manually using code is that migration is interleaved with the update of (the copy of) the static model within the run-time model. As a result, the run-time model is in an inconsistent state itself when the migrations are being applied. Separating migration logic until after the update has fully finished is non-trivial, since migrations depend on the knowledge of what has changed (i.e., the delta itself).

**Scattering of migration logic.** Another problem is that migration logic often requires expressing and maintaining global invariants on the run-time state. Modularizing the code according to the type of model elements (e.g., State, Transition, etc.), or delta operations (e.g., Add, Remove, etc.), causes scattering of migration logic over multiple classes and methods.

**Lack of extensibility.** Finally, as a result of the previous point, migration logic is not modularly extensible. Extending the DSL with new constructs and corresponding additional invariants and rules requires invasive modification of the existing migration code. With NEXTep, migration rules and invariants can simply be added to the NEXTep definition, i.e., in conjunction with existing constraints. The potential interactions between constraints are handled by the solver back-end.

To summarize, NEXTep improves upon the earlier migration work in that it provides a declarative, decoupled, and modular way of expressing migration policies. The heavy lifting of finding the new run-time state that is consistent with the new version of the DSL program is delegated to the solver.

6 Related Work & Discussion
In this section, we discuss the related work from three different points of view: the problem of run-time state migration (Section 6.1), the solutions based on constraint solving (Section 6.2), and the area of DSL debugging (Section 6.3). Next to highlighting the existing approaches, we discuss which of their findings we can adopt in our work in order to improve efficiency, expressiveness, and usability of our approach.
6.1 Model Synchronization

In this paper, we address the problem of finding a new run-time state that is consistent with an updated DSL program. In a broader perspective, this is a particular case of the problem of keeping different models in sync, which includes well-known sub-problems: consistency checking between different models, change propagation, co-evolution, model repair, etc. As such, this problem has been studied in different research fields and from multiple angles. For instance, the field of views and viewpoints engineering studies the problem of consistency checking between source models and views. The corresponding approaches that address this problem include incremental backward change propagation [23], lenses [11], and triple-graph grammars [22]. Our approach distinguishes itself in letting the language designer specify her own migration policies in a declarative way.

In Model-Driven Engineering (MDE), consistency relations between different models are typically defined through model-to-model transformations. Such model transformations can be used to (re)construct a new (run-time) model for an updated (static) model. Concretely, our work was inspired by the work of Macedo et al. [15, 16], which implements QVT-R bidirectional model transformations using Alloy and its SAT solver to construct consistent models. In comparison, in our work we use Alloy and the SMT solver as a back-end, which allows for more expressiveness when configuring live modeling for a DSL.

Bill et al. use UML and OCL to formalize a synchronization model that includes both a change model and a consistency model [1]. The synchronization model defines inter-model relations and constraints (comparable to our Nextep definition); the change model introduces all possible changes that can be used to construct a new model and assigns a cost to each of such changes. The USE model finder uses these cost values to find the most suitable (optimal) model. On the one hand, defining all possible changes (including language-agnostic, language-specific, and composite changes) requires operational thinking and can result in a tangled and complex model. In our approach, we strive towards a declarative definition. On the other hand, using cost values the authors define five different heuristics for finding the most suitable solution (including the least change strategy that we employ in Nextep). Thus, we plan to leverage their experience in future work to extend Nextep with additional heuristics.

6.2 Constraint-based Solving

In our approach, we use the SMT solver to find a new run-time model that satisfies the specified constraints. Recently, constraint solvers are being adopted in MDE for the automatic analysis of modeling languages. For instance, Erata et al. present AlloyInEcore, a meta-modeling language that allows for adding Alloy-like invariants into Ecore metamodels [10]. Built on top of the Kodkod (SAT) solver, their tool automatically detects inconsistent models and completes partial models. As both AlloyInEcore and Nextep translate to the relational logic (of Kodkod and AlleAlle correspondingly), the syntaxes of these two languages are similar. As AlloyInEcore is a more developed and mature language comparing to Nextep, we consider learning from the design decisions of AlloyInEcore in our future work.

Straeten et al. assess the performance of using constraint solvers (Kodkod in particular) for resolving inconsistencies in models [25]. They experiment with models that were reverse-engineered from a set of open-source projects and further translated to Kodkod, to which they add consistency rules formulated directly in Kodkod. The model sizes range from 2064 to 8572 elements. The results obtained in this work demonstrate that “the approach does not provide instantaneous resolution on medium scale models”. These results suggest a potential threshold for our approach. However, we believe that we can still significantly improve the performance of Nextep.

An alternative approach to employing SAT/SMT solvers is to use constraint programming systems. For instance, Cabot et al. translate UML class diagrams extended with OCL constraints into Constraint Satisfaction Problems (CSPs) and the ECLiPSe constraint programming system to construct model instances [5]. Steinmann et al. use constraints embedded into an attribute grammar to check the well-formedness of programs and to compute repair alternatives for malformed programs [24]. Both these works propose various optimizations and search algorithms in order to improve performance of constraint-based model finding.

6.3 DSL Debugging

Although the objectives are different, live modeling is also closely related to debugging. Debuggers enable programmers to monitor the execution of a program, set breakpoints, inspect and modify runtime values, and, under certain constraints, hot swap some parts of the code itself. Over time, a number of debuggers have been developed for modeling languages, for instance for DEVS [19], fUML [17], or individual diagrams of the Unified Modeling Language (e.g., [7, 9]).

Beyond classical forward-in-time debuggers, Bousse et al. proposed a methodology for the development of generic omniscient debuggers for DSLs [2] backed by efficient and domain-specific execution trace management facilities [3]. Ráth et al. use the VIATRA [29] framework to simulate Petri nets. Users can edit models on-the-fly at simulation time, for instance to resolve cases of non-determinism [20].

We expect many potential cross-fertilization between model debugging and live modeling, and we would like to investigate in the future how our constraint-based approach can solve some of the current problems of debuggers, regarding, for instance, migration of run-time state after code swapping.
7 Conclusion & Future Work

Conclusion Live modeling has the potential to improve the experience of using executable DSLs. Immediate feedback allows DSL users to better assess the impact of the changes they make to their models. A central problem for live modeling languages is how to reconcile changes to a model with the run-time state of its execution. In this paper, we proposed to formulate the problem of run-time state migration in terms of constraints on the run-time state.

We have identified two categories of such constraints: semantic relations which ensure that the new run-time state is consistent with the new version of the DSL model, and domain-specific migration rules which are specified explicitly by the language designer. To choose the most suitable run-time state out of potentially many solutions, we used the heuristic of minimum distance between the new and old run-time states.

We have illustrated this approach using Nextep, a metamodeling language that allows to define such invariants and migration policies. Nextep employs model finding technique, backed by a solver, to automatically infer a new run-time model that satisfies the declared constraints.

We have evaluated the performance of Nextep on a simple FSM DSL. Initial results show that performance is satisfactory, albeit unpredictable even for similar problem specifications. Furthermore, we have applied Nextep to an existing DSL for robotic arm control, which is semantically richer than the FSM DSL. Overall, our results show that constraint-based state migration as realized by Nextep is a promising technique for engineering live modeling languages.

Future Work A number of research questions stem from our early prototype and evaluation which we hope to address in future work.

First, we did not address all the steps of the live modeling experience. Once a new run-time state is inferred, how should it be provided back in a seamless way to the user? How should it be provided to the DSL interpreter itself to let it resume the execution transparently?

Second, we believe user experience plays a crucial aspect in live modeling. How should a user interact with the live modeling framework, through which interface? To be adopted, live modeling must be as close to real time as possible to provide a seamless experience. How to optimize the time it takes to find a new run-time state? We believe that live modeling tools should be evaluated empirically with real users to assess their benefits, and we hope to address these questions in the future.

Finally, we consider the use of a generic minimum distance heuristic to guide the constraint solving process as both a strength and a weakness of our approach. Investigating whether other generic heuristics or domain-specific heuristics could be employed requires more experience with various kinds of DSLs and remains future work.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work is partially supported by the Agile Language Engineering (ALE) associate team (http://gemoc.org/ale/), and the EU’s Horizon 2020 Project No. 732223 CROSSMINER.

References


Constraint-based Run-time State Migration for Live Modeling


