McGill University
School of Computer Science
Sable Research Group

Transforming Timeline specifications into automata for runtime monitoring (extended version)

Sable Technical Report No. 2008-1

Eric Bodden and Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Québec, Canada

February 3rd, 2008

Contents

1 Introduction **3**

2 The Timeline formalism **4**

2.1 Timeline abstract syntax in AToM^3 5

2.2 Timeline concrete syntax in AToM^3 6

3 Transformation into finite automata **7**

4 Eliminating unreachable states **13**

5 Applicability to runtime monitoring **13**

5.1 Per-object specifications 14

5.2 Is Timeline optimal? 15

6 User experience with AToM^3 suggested improvements of the tool **16**

6.1 What worked well 16

6.2 Suggestions for improvements 16

7 Conclusion and future work **17**

List of Figures

1 Timeline specification 5

2 Abstract syntax of the Timeline formalism in AToM^3 6

3 Adding the artificial end event 7

4 Adding states 8

5 Marking the initial state 9

6 Creating transitions 9

7 Folding constraints 10

8 Applying constraint start and end points (left-hand sides) 11

9 Applying constraint bodies to the loops (left-hand side) 11

10 Correcting the semantics of fail events 12

11 Example - resulting automaton 12

12 Base case for calculating the transitive closure 13

13 Step case for calculating the transitive closure 13
14 Automaton-based tracematch checking for writes to closed files 14

Abstract

In runtime monitoring, a programmer specifies code to execute whenever a sequence of events occurs during program execution. Previous and related work has shown that runtime monitoring techniques can be useful in order to validate or guarantee the safety and security of running programs. Those techniques have however not been incorporated in everyday software development processes. One problem that hinders industry adoption is that the required specifications use a cumbersome, textual notation. As a consequence, only verification experts, not programmers, can understand what a given specification means and in particular, whether it is correct. In 2001, researchers at Bell Labs proposed the Timeline formalism. This formalism was designed with ease of use in mind, for the purpose of static verification (and not, as in our work, for runtime monitoring).

In this article, we describe how software safety specifications can be described visually in the Timeline formalism and subsequently transformed into finite automata suitable for runtime monitoring, using our meta-modelling and model transformation tool AToM^3. The synthesized automata are subsequently fed into an existing monitoring back-end that generates efficient runtime monitors for them. Those monitors can then automatically be applied to Java programs.

Our work shows that the transformation of Timeline models to automata is not only feasible in an efficient and sound way but also helps programmers identify correspondences between the original specification and the generated monitors. We argue that visual specification of safety criteria and subsequent automatic synthesis of runtime monitors will help users reason about the correctness of their specifications on the one hand and effectively deploy them in industrial settings on the other hand.

1 Introduction

Static program verification in the form of model checking and theorem proving has in the past been very successful, however mostly when applied to small embedded systems. The intrinsic exponential complexity of the involved algorithms makes it hard to apply them to large-scale applications. Runtime monitoring or runtime verification [1] tries to find new ways to support automated verification of such applications. This is done by combining declarative safety specifications with automated tools that allow verification of these properties, not statically but dynamically, when the program under test is executed. Research has produced a variety of such tools over the last years, many of which have helped find real errors in large-scale applications. Yet, those techniques have not yet been able to make the transition to everyday use in regular software development processes. This is due to two reasons. Firstly, many of the existing runtime monitoring tools cause a significant runtime overhead, lengthening test runs unduly. Secondly, the kind of specifications that can be verified by such tools often use a quite cumbersome notation. This leads to the fact that only verification experts, not programmers, can understand what a given specification means and in particular, whether it is correct.

The first problem of generating efficient runtime monitors has been addressed extensively in previous [2–4] and related [5, 6] work. In particular, our research group maintains an efficient implementation of *tracematches* [7], an implementation of runtime monitoring that allows specifications to match on the dynamic execution trace, using regular expressions with free variables than can bind objects. For instance, a pattern of the form `File f: open(f) dispose(f)` over the alphabet $\Sigma = \{$`open,dispose`$\}$ could denote disposing a file that is currently open. Such a specification might seem easy to read, but sometimes subtle problems can arise. For example, the aforementioned pattern would also match the event sequence `open(f1) close(f1) dispose(f1)`, where a file `f1` is properly closed before it is disposed. In order to fix the pattern, one would have to change the alphabet of the regular expression to $\Sigma = \{\texttt{open,close,dispose}\}$. We strongly believe that such subtle difficulties with existing specification formalisms are among the main reasons why formal verification techniques such as runtime monitoring have, despite their effectiveness and efficiency, not yet found widespread industry adoption.

In 2001, Smith et al. from Bell Labs proposed the *Timeline* formalism as a way to ease the specification of temporal properties [8]. They presented a visual tool to design Timeline specifications. The tool converts those specifications into Büchi automata, suitable for static verification. However, this translation is done in code, and hence it is hidden from the user. We believe that the Timeline formalism is indeed much more comprehensible than many other temporal specification formalisms. However, we also believe that a tool can and should benefit from explicit visual graph rewriting techniques. Implementing formalism (such as Timeline) semantics via visual graph transformations allows (1) to easily experiment with different semantics by altering transformation rules and (2) once the semantics is fixed, to easily reason about its correctness. Hence, in the following, we propose an explicit visual graph transformation using the AToM^3 tool [9], that rewrites specifications in the Timeline formalism to corresponding finite state machines suitable for runtime monitoring. Those state machines can then be fed into our tracematch-based back-end, which generates an equivalent and efficient runtime monitor. This monitor can be applied to arbitrary Java programs through compilation.

It is also noted that Smith et al. did not take into account per-object specifications such as the per-file specification mentioned above. In this work we show how the Timeline formalism can be used for such specifications as well. The generated Java monitors automatically take into account the necessary object bindings, exploiting our performance optimizations from previous work.

The remainder of this paper is organized as follows. In Section 2 we introduce the Timeline formalism, its visual concrete syntax, and its semantics. The visual specification of transformation into finite automata is described in Section 3. In Section 4 we further describe an (optional) optimization that eliminates unreachable states. This optimization is also implemented via graph transformations, in a way which we find very natural and elegant. In Section 5, we sketch how the resulting automata can be used in our runtime monitoring back-end. Finally, we conclude and state future work in Section 7.

2 The Timeline formalism

Each Timeline specification consists of a single time line, which is independent of all the others. This is important, as it enabled modular reasoning. A time line makes sense in its own right and its truth value does not depend on the presence of other time lines.

Each time line represents an ordered sequence of events. The first event is a distinguished start event, representing the time of start-up of the application. All events but this start event are associated with a label and one of the following three event types.

regular event Such an event may or may not occur. It imposes no requirement and is only used to build up context for a complete pattern match. Regular events are denoted with the letter **e**.

required event A required event *must* occur, whenever its left-context on the time line was matched. Required events are denoted with the letter **r**.

fail events A fail event *must not* occur after its left context has matched. Such an event is denoted with the letter **X**.

Along with those events, a time line can be augmented with constraints, restricting the matching process. A constraint holds a Boolean combination of propositions and may include or exclude the start and/or end event it is attached to.

While Smith et al. used a motivating example [8] specifying a dial-tone feature used at Bell labs, we here use a running example motivated by our own work. Fig. 1 shows an extension of the aforementioned file/dispose example. We wish to specify that a file must not be disposed as long as it is open. Furthermore, we would like to make sure that any open file is closed at some point in time, before the program exits. The Timeline specification directly states *both requirements together*: After seeing a *regular* event `open`, we *require* an event `close` (in the end of the time line) and in between we state that no `dispose` event may occur (excluded event, marked with an `X`). A constraint between the `open` and `dispose` events is used to state that those requirements

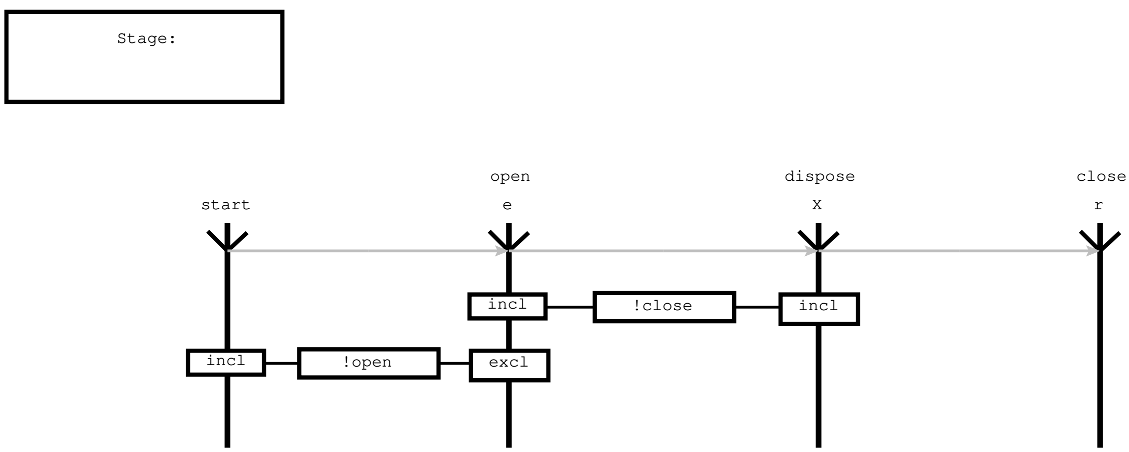


Figure 1: Timeline specification stating that any opened file should be closed and should not be disposed before closing it.

only apply if the file has not been closed already prior to disposal. A second constraint on the left states that we are only interested in the last `open` event, as our translation will assure that former events were already handled once we get to this stage of evaluation. Fig. 1 shows the Timeline specification as it is denoted in a modelling environment built using AToM^3 [9]. This environment uses the following abstract syntax in order to represent such specifications.

2.1 Timeline abstract syntax in AToM^3

We model an event as an object with a string label and one of five types: *start*, *regular*, *required*, *fail* and *end*. The “end” event is artificial. It cannot be specified by the user and is only used within the translation to finite automata.

A time line consists of a sequence of events. The sequence is established via an ordering relation. A further relation between events describes the constraints among them. Each constraint is modelled as an edge between two events. It can include or exclude the event at its start and/or end. Furthermore it is labelled with a string label, stating the actual constraint expression.

Fig. 2 shows the class diagram for the abstract syntax of Timeline in AToM^3. In addition to the aforementioned entities, it shows a *Stage* class. As we will explain in Section 3, we use a singleton object of this class for each Timeline specification to be able to implement its translation in a stateful way. This is a workaround because the version of AToM^3 used did not yet support programmed graph rewriting.

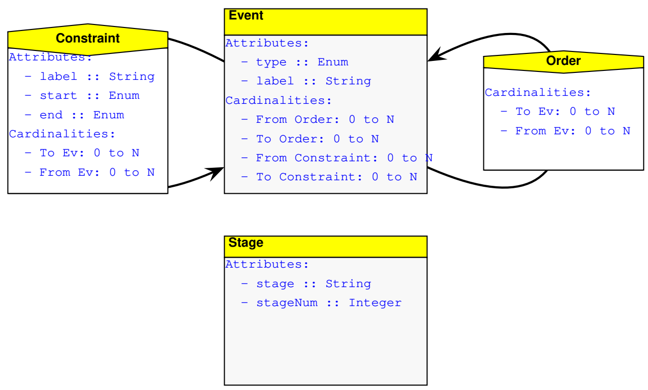


Figure 2: Abstract syntax of the Timeline formalism in AToM^3

The static semantics of the Timeline formalism imposes the following type checks on correct Timeline specifications. (see [8] for details)

1. Each time line must be fully connected by the Order relationship. In particular, this order is anti-symmetric, transitive and total.
2. In each time line, the smallest event in this relationship must be of type "start".
3. Each event must have at most one immediate predecessor and successor in this relationship.
4. When a constraint relation starts at an event e_1 and ends at e_2, then e_1 must be smaller than e_2 in the Order.
5. There must not exist any two subsequent fail events.
6. A constraint may not begin or end at a fail event, unless the fail event is the first event or last event of the time line.

The translation we give in Section 3 is based on the above assumptions. They can relatively easily be verified in the AToM^3 modelling tool, at design time.

2.2 Timeline concrete syntax in AToM^3

Each abstract syntax entity is given a concrete visual representation. Events are represented by vertical lines, while the temporal order relation between them is drawn as a directed edge. Constraints are undirected edges with labels. As Fig. 1 shows, AToM^3 has built-in support for displaying attribute values of entities in a text box as of its visual representation.

3 Transformation into finite automata

We assume a given time line t which fulfils the constraints mentioned in Section 2.1. Further, we formally denote t by $t = (E, O, C)$ with:

- E, a finite set of events;
- $O \subset E \times E$, a total order, the temporal order relationship;
- C, a finite set of constraints.

Each event $e \in E$ is of the form $e = (l_e, t_e)$ with l_e a string label and

$$t_e \in \{start,\ regular,\ required,\ fail,\ end\}.$$

We then transform each Timeline specification into a finite state machine, using eight transformation stages that are executed in sequential order. In our model-driven approach, each of those stages is explicitly modelled by one or more graph grammar rules. In the following, we explain each stage in detail.

Stage 1 - Add an end event. For the subsequent transformation stages it will be useful to have an additional end event, which marks the last event in the time line. Hence, our first rule adds such an event to the one and only event of the time line which has no outgoing edge in the temporal order relation. Note that there can only be one such event because the temporal order, being a total order on a finite number of elements, has a unique largest element. The graph rewriting rule stating this transformation is depicted in Fig. 3. The left-hand side of this rule is annotated with an additional matching condition, stating that there may be no outgoing edge in the Order relation:

$$matchcond(e) := \neg\exists e' \in E\ .\ (e, e') \in O$$

Note how number labels on left-hand side (LHS) and right-hand side (RHS) of rules allow one to relate nodes on both sides. Labels present on both sides denote retained nodes, labels present only on the LHS denote deleted nodes, and labels present only on the RHS denote created nodes. On the LHS, `<ANY>` matches any attribute value. On the RHS, the notation `<COPIED>` denotes attribute copying from the LHS, `<SPECIFIED>` denotes an explicitly computed attribute.

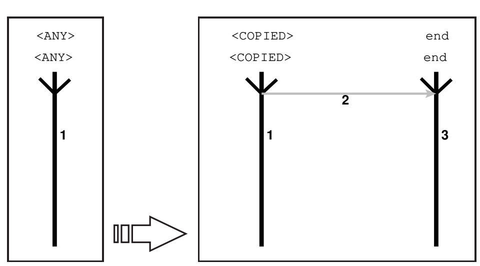


Figure 3: Adding the artificial end event

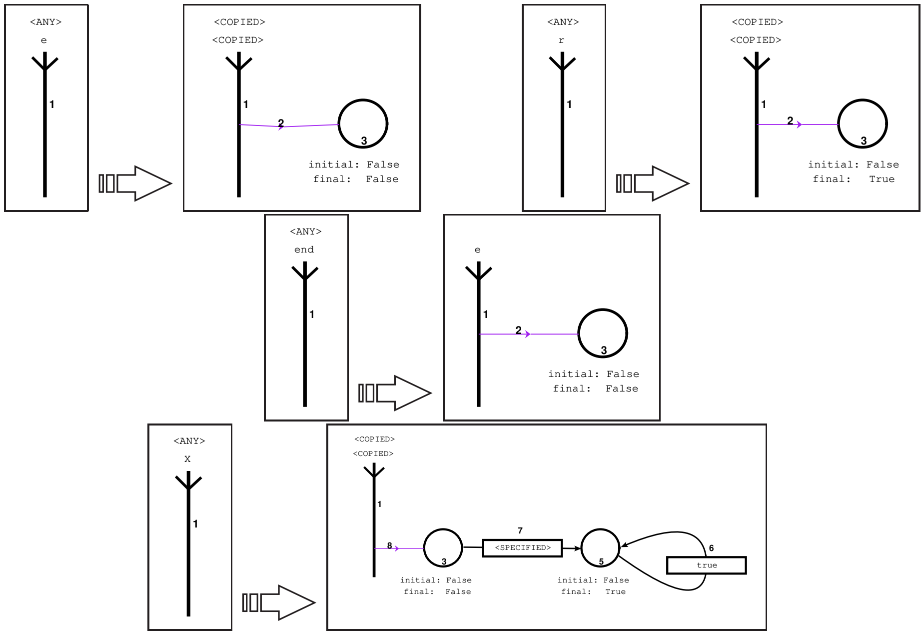


Figure 4: Adding states

Stage 2 - Add states. For each event we then generate a state which reflects the point in time immediately before the associated event occurs. We do so by using four different transformation rules, one each for regular, required and fail events plus one for the end event. We use multiple rules here, because the kind of state we generate depends on the event type.

The rules are shown in Fig. 4. For a regular event (marked with an **e**), we simply generate a non-final state. We add a generic edge between the event and the state to be able to relate them to each other in later transformation stages. AToM^3 allows generic edges to connect any kind of nodes. Other connections are constrained by the formalism's meta-model. For a required event we generate a final state accordingly. This is because the generated state machine is meant to accept an input stream of events if and only if it *violates* the specification. Hence, in case the monitor has not seen a required event yet, it has to be in an accepting state. Similarly, for a fail event we actually add two states. The first one is non-final and reflects the point in time before the event occurs. The second one is final and contains a *true* loop. This "sink" state has special semantics in the sense that it allows for early error detection: once it is visited, we know that the property is violated no matter what suffix of the trace will follow. The incoming transition to this state is labelled with l_e, the label of the matched event. We copy the value from the event label. Finally, the end event is treated as a regular event.

Stage 3 - Marking the initial state. In order to construct a valid finite automaton, we have to mark its initial state as initial. We identify this initial state as the unique state that is associated with the unique successor of the start event in the temporal order relation. The corresponding rule is shown in Fig. 5.

Stage 4 - Adding transitions. This step adds the necessary transitions between the states. For any two states belonging to two events e_i, e_{i+1} where e_{i+1} follows e_i in the temporal order, we

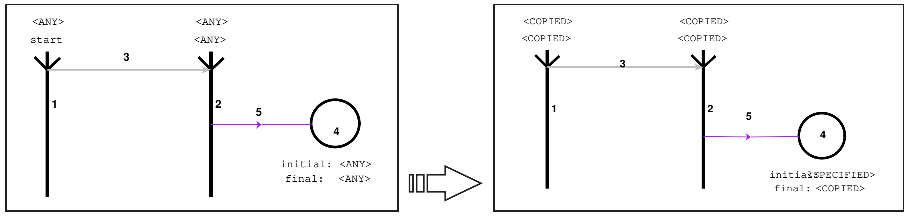


Figure 5: Marking the initial state

add a transition between those states, labelled with l_{e_i}, simply because we want to move from the state representing "before e_i" to its successor, when l_{e_i} occurs. We also add a loop to the state associated with e_i, holding the label $!l_{e_i}$ (read "*not* l_{e_i}"), so that we do not discard a partial match only because l_{e_i} has not been seen *yet*. Fig. 6 shows our rule for creating transitions.

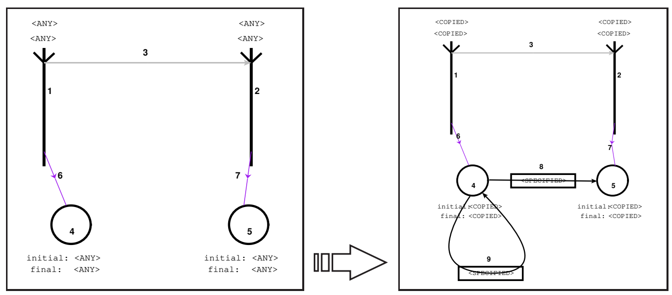


Figure 6: Creating transitions

Stage 5 - Folding constraints. The automaton we now have associated with the original time line is already a valid finite automaton, equivalent to the time line, not taking constraints into account. Hence, the constraints are handled next. The idea is to copy constraints over from the time line onto the transitions of the resulting automaton. However, one problem still exists: a constraint may be linked to two states which are *not* immediate successors in the temporal order, i.e., between events e_i, e_j with $j - i > 1$. In such a case, the constraint also takes effect at all events $e_{i+1}, \ldots, e_{j-1}$, *even though* those are not directly connected to the constraint. In [8], Smith et al. propose a tableau based approach in order to calculate the constraints which apply to each single transition. We rather opted for a visual approach, which we find easier to understand and implement.

The rule we describe here resolves the transitive notion of a constraint by connecting all the intermediate events *explicitly* to an equivalent constraint. This is depicted in Fig. 7 and makes the above observation explicit: whenever we see two events e_i, e_j with a constraint between them and there exists an event e_{j-1} preceding e_j in the temporal order, then we split the constraint into two, one covering the region between e_i and e_{j-1} and one covering the step from e_{j-1} to e_j. Note that the first of those two constraints might still reach over multiple events. In the general case, where $\delta := j - i$, we hence have to apply this rule $\delta - 1$ times until the fixed point is reached. This is automatically performed by virtue of AToM^3's graph transformation semantics. When folding the constraints in this way, we also have to make sure that the first constraint includes its starting event only when the original constraint did so. Similarly, the second constraint must include its end event only if the original constraint did so. We hence copy over those properties. Fig. 7 reflects this by showing `<SPECIFIED>` at the appropriate labels. For the intermediate events it is clear that

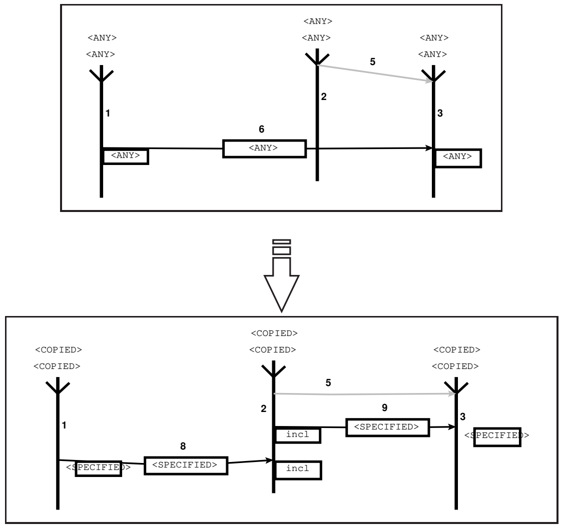


Figure 7: Folding constraints

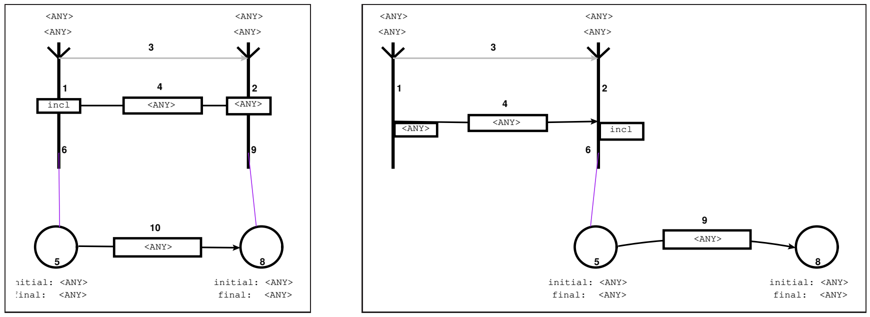
those have to be included. Hence, we set this property explicitly to that value.

Stage 6 - Applying the constraints. After having folded the constraints, we can safely assume that constraints only exist between immediate successor events e_i, e_{i+1}. This assumption provides us with a direct and local mapping between any two events, their associated constraints and states. In the following, we explain three different rules which are used to propagate the constraints onto the related transitions of the finite automaton.

Applying a constraint at its start point. The first rule is shown in Fig. 8(a) (we only show the left-hand side here, as the right-hand side has the same structure). Its purpose is to propagate a constraint from an included start event of a constraint to the corresponding transition. If a starting event e is included in a constraint c this means that we only accept this event (i.e., make progress in the automaton) if c holds when e occurs. Consequently, we propagate c from the left event onto the transition connecting the two associated states — the label of that transition changes from l to (l *and* c). We remind the reader that the left state of the two reflects the point in time before e was read and the right one the point in time after e was read. Also, we should mention that we made the rule match only if the constraint does not already exist at the target transition. This prevents AToM^3 from applying the same rule repeatedly.

Applying a constraint at its end point. Similarly, we have to handle cases where the end point of a constraint is included. The rule in Fig. 8(b) shows how we propagate the constraint label onto any transition moving out of the end state of the constraint, in case the right event is included in the constraint.

Applying a constraint to an interval. The "body" of the constraint, i.e., the part between its start point and end point finally has to be applied to the corresponding loop, since the loop — as is the case with the constraint — describes what behaviour is allowed *before* the next event occurs. The left-hand side of the equivalent transformation rule is shown in Fig. 9. For each such match



(a) Applying at start points (b) Applying at end points

Figure 8: Applying constraint start and end points (left-hand sides)

we add the negation of the label of the constraint onto the label of the loop, which means that whenever the constraint is violated, we may *not* return to this state, i.e., in the absence of other matching transitions, the partial match is discarded.

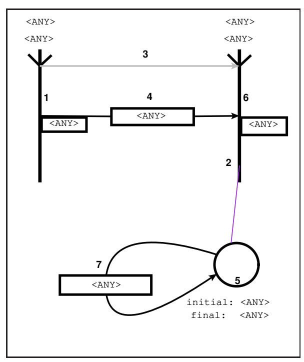


Figure 9: Applying constraint bodies to the loops (left-hand side)

Stage 7 - Implement semantics of fail events. The way we generated states for fail events does not yet exactly reflect the semantics given in [8]. In the current state machine, the scope of a fail event would extend until the end of the input instead of only until the event following the fail event. This means that we would falsely detect a violation if the fail event occurs *anywhere* on the remaining path. However the semantics state that it only must not occur until the next regular (or required) event occurs. The rule shown in Fig. 10 depicts the appropriate change to implement the correct fail event semantics.

Assume that e is a fail event. We eliminate the state q_e, changing its incoming transition to have q_{e+1} as target state. The transition from q_e to the failure state q'_e is changed to start at q_{e+1}.

We wish to remind the reader that each state q_e in the automaton models the point in time right *before* event e was seen. Taking this into account, we can now see that after the transformation, the semantics are implemented correctly: when reading the event preceding e, we move to the state associated with the event following e directly, because this is the next event on our "progress path". Should in the meantime however, the fail event occur, then we move to the failure state.

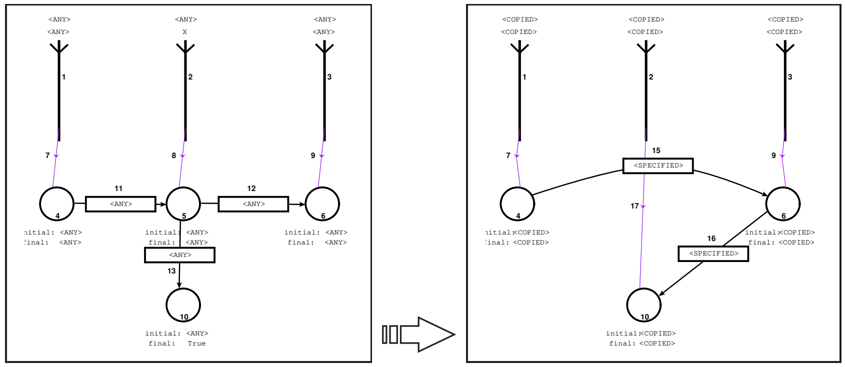


Figure 10: Correcting the semantics of fail events

Stage 8 - Removing the events. After all the previous steps we now have a finite automaton model which encodes the semantics of the original Timeline model. Hence, we can remove all event information. Here, it suffices to remove the events alone, because AToM^3 automatically removes all (dangling) associated relations automatically. Consequently, we can simply implement this step by means of a rule with an unspecified event on the left-hand side and an empty right-hand side. Fig. 11 shows the result of the complete translation (steps 1 through 8) of our example from Fig. 1.

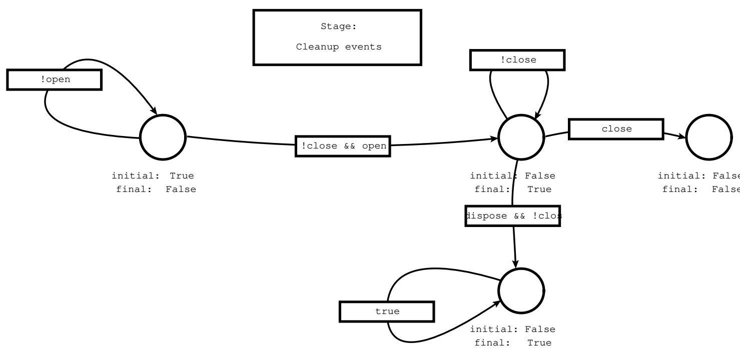


Figure 11: Example - resulting automaton

Stateful transformations, termination and correctness. In order to prevent unwanted recursive application of the different transformations, we had to make parts of the graph transformation model stateful, which means that we carry around an explicit state, giving information about what rule was last applied. This prevents for instance the rule for "adding transitions" being applied again after transitions have been removed by the correction step for the fail event semantics. We store the state in a visual label called "stage" as shown in Fig. 11. Future versions of AToM^3 will support programmed graph rewriting, allowing for the elegant and explicit description of transformation stages. Each stage terminates due to implicit or explicit termination conditions. The folding of constraints, for instance, automatically reaches a fixed point when there is no constraint any more that spans more than two events. The propagation of constraint values, however, uses a hand-coded check as described above. With respect to correctness it is noted that a formal proof of transformation properties such as termination is out of the scope of this paper. Nevertheless, such a proof by structural induction over the different Timeline language constructs is quite straightforward.

4 Eliminating unreachable states

Despite the fact that the resulting finite automaton accurately reflects the semantics of the original Timeline model, it may still contain states which are unproductive. That is, no final state can be reached from such states, as can be seen on the right side of Fig. 11.

Denoting a finite automaton $\mathcal{A}$ by the usual quintuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$, a state $q \in Q$ is said to be *unproductive* if there exists no path from q to some $q_F \in F \subseteq Q$. The set $P \subseteq Q$ of productive states can hence be defined as the smallest set for which holds:

$$F \subseteq P \ ; \ q_j \in P \wedge \exists l \in \Sigma \ . \ (q_i, l, q_j) \in \Delta \rightarrow q_i \in P$$

As one can see, this definition is recursive. It implies that a state is productive if a final state can be reached via the transitive closure of the transition relation. In our approach, we explicitly model the transitive closure of the transition relation in AToM^3 by two rules. The base case is shown in Fig. 12 where we add a generic edge wherever a transition exists.

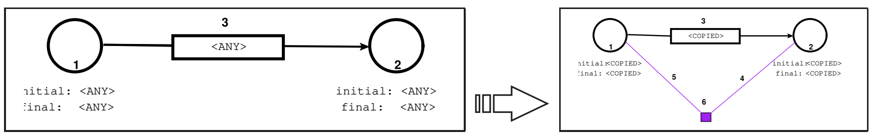


Figure 12: Base case for calculating the transitive closure

The step case is shown in Fig. 13: For each two consecutive generic edges we add a third one bridging the two. The AToM^3 tool automatically applies this rule until a fixed point is reached. In the fixed point there exists a generic edge (q_i, q_j) if and only if q_j is reachable from q_i.

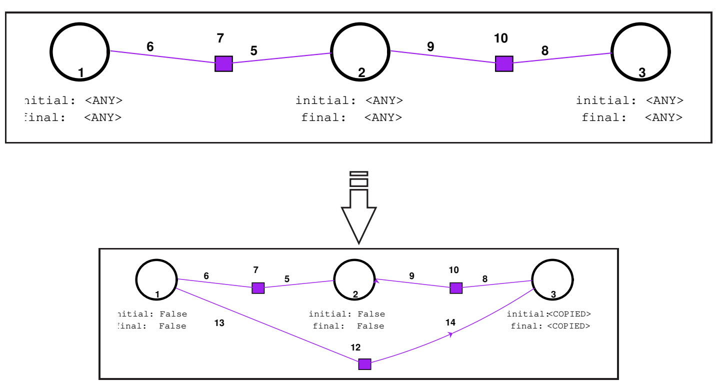


Figure 13: Step case for calculating the transitive closure

Given the transitive closure, we can now simply remove any state which is not (directly) connected to a final state via the transitive closure relation. In Fig. 11 this would eliminate the right-most state.

5 Applicability to runtime monitoring

As mentioned earlier, the generated finite state machines can be used for the purpose of runtime verification. While Büchi automata, which are used for static verification, read an input of infinite length, the automata we use here accept a finite input. This is because in runtime verification a program is indeed executed and hence, every event sequence is terminated as the program shuts down.

As described in [7], our current implementation of *tracematches* generates finite-state monitors from regular expressions with free variables, where each variable is bound to matching objects at runtime. Hence, it is relatively easy to modify the back-end in such a way that it does not generate the finite state monitor from a given regular expression but instead reads it in directly. In tracematches, abstract events are mapped onto concrete events in the code via *pointcuts* in the aspect-oriented programming language AspectJ [10]. A pointcut in this setting is a predicate over runtime events.

```
1 tracematch(File f) {
2     sym close after returning: call(* File.close()) && target(f);
3     sym write before: call(* File.write(..)) && target(f);
4     sym dispose before: call(* File.delete()) && target(f);
5
6     initial state 0; final state 1; final state 2;  //define states
7     (0,open,1); (1,dispose,2);                      //define transitions
8     { System.err.println("State violation on file "+f+"."); }
9 }
```

Figure 14: Automaton-based tracematch checking for writes to closed files

Fig. 14 shows what such a state-based tracematch syntax could look like for our *file* example (automatic generation of this textual representation is future work). In its header in line 1, the tracematch declares to reason about a single file `f`. Lines 2-4 hold two user-defined symbols based on AspectJ pointcuts. The transition table for the tracematch automaton follows in lines 6-7. This part of the specification can be directly generated from the visual state machine model. Note that unreachable states do not show up, cf. Section 4. Also, certain negated labels on transitions do not need to be copied due to the event-based semantics of tracematch automata. Line 8 finally holds the body of code that is to be executed on each single match. Note that this body has access to the bound variable of `f`, an important feature of tracematches.

5.1 Per-object specifications

As we saw in the introduction, many interesting safety properties need to be evaluated "per object". For instance, if we want to say that a file must be closed after it was opened, this is a property *per file*. While tracematches support such per-object specifications by default, Timeline was not designed with such specifications in mind. Nevertheless, we claim that Timeline specifications could be modified to incorporate such a feature. One possibility would be to replace an event `e` by an `e(x)` where `x` is a free variable binding an object. For our file example, we could change the time line to include events `open(f)` and `close(f)`. An alternative would be to combine Timeline with object life lines. We leave further investigation such extensions to future work.

5.2 Is Timeline optimal?

In this paper we stated a graphical translation from Timeline to runtime monitors. The intend was to see whether such a translation is feasible and whether it would in fact enhance our understanding. Looking at our running example, we can find the following advantages of Timeline over the existing tracematches syntax.

Conciseness of representation In tracematches, a regular pattern matches a *violating sequence*. This is roughly equivalent to stating a Timeline specification consisting of a sequence of regular events and ending in a fail event. Timeline is more direct when it comes to specifying that an event has to occur. We simply state a required event. In tracematches, such a pattern would have to be simulated. For example, for the requirement that a file must be closed after it was opened, we would have to match the regular pattern `open shutdown` over an alphabet {`open, close, shutdown`}, where `shutdown` would be the event of shutting down the program. A violation would occur whenever we see `open` followed by `shutdown` with no `close` in between. Although such simulation is possible, we claim that it is indirect and hence make specifications harder to read.

Compositionality and fail events In our example, we further saw that two requirements could easily be combined in a single time line: (1) Always close a file after it was opened and (2) do not dispose the file before this required close-event. In tracematches, such a pattern would have to be written using a logical "or" operator: `open shutdown | open dispose` or alternatively `open (shutdown | dispose)`, over an alphabet {`open, close, dispose, shutdown`}. While the Timeline specification exactly states the requirement that there *must* be a close-event following the open-event, with *no* dispose-event in between, the tracematch specification states this requirement only implicitly, by defining that shutdown and dispose are both forbidden after open, (if there has no close event happened in between). Again, we feel that the direct way of composing required and fail events directly in one specification makes this specification easier to read.

Constraints in an event-based setting Timeline was initially developed to generate Büchi automata which were then fed into the model checker Spin [11]. Traditionally, such model checkers were used to check propositional logics. Such logics are based on the notion of a proposition, a statement that *at each program state* can either hold nor not. Using static verification, one can then for example show that a given proposition holds in *all states* (or possibly a subset of states, defined by some other proposition). While in a static view such specifications make sense, in the field of runtime verification such a property would be hard to validate, since it would have to be checked after the execution of each single instruction. Hence, in runtime verification one is much more interested in *events*: The constraint that a variable `i` should never have a value smaller than 0 would be expressed by stating that no event should occur, setting `i` to such a value. Consequently, constraints, as present in Timeline, are probably not really suitable to include in a specification formalism meant for runtime verification. Again, we leave the investigation of concrete alternatives to future work.

Applicability to real-world specifications Generally, we believe that Timeline or Timeline-based formalisms should scale very well. This is because with any event added to the time line, this time line only needs to be extended with that event. This is in contrast to temporal logics like

LTL [12] where each additional event increases the nesting depth of the formula, rendering nontrivial specifications almost unreadable. Interestingly though, a study conducted by Dwyer et al. [13] suggests that even real-world specifications used in industry are usually of a fairly simple structure. The researchers even go so far to propose a pattern language to cover the most common specification patterns.

6 User experience with AToM^3 suggested improvements of the tool

In this section we briefly summarize our experience with using AToM^3 as a tool for visual specification of modelling languages and model transformations. We highlight what worked for us but also needs for further improvements.

6.1 What worked well

The following worked very well.

Modelling with concrete syntax The ability to describe both models and transformations, in concrete syntax is useful for domain experts. Indeed, we identified this as the number one reason for using visual graph transformations opposed to hand written code. With concrete syntax, the transformation becomes visually explicit to the modeller. It is straightforward to picture the effects of a transformation in one's mind, because this transformation can directly be *seen* already in the transformation rules themselves.

Large productivity increase In [8] the original creators of the Timeline formalism reported that they spent about one month on implementing a modelling environment for Timeline. Using AToM^3 we were able to achieve the same task in less than three days. A more experienced user of AToM^3 would probably have been able to finish the implementation in an even smaller amount of time. Furthermore, because in AToM^3 the semantics are implemented via visual graph transformation rules, this implementation will easily allow us to experiment with different semantics, by just modifying the rewrite rules accordingly.

6.2 Suggestions for improvements

We believe that although our overall user experience with AToM^3 was highly satisfying, the following issues remain.

Negative application conditions. In many instances *negative application conditions* (NACs) would have been very useful to prevent a rule from applying in certain situations. The Montréal version of AToM^3 we used allowed such conditions only in hand coded form, via inserting Python code. Note that the Madrid version of AToM^3 does have support for NACs.

Programmed graph rewriting was lacking In addition, we had to insert the aforementioned "Stage" label into each of our visual specifications. This label was then used to keep track of the current rewriting phase in order to schedule the rewriting correctly. The actual scheduling was again written in Python code. *Programmed graph rewriting* is a solution to this problem as put forward by the PROGRES [14] model transformation tool. Recent AToM^3 developments [15] presented at AGTIVE do support programmed graph rewriting.

Copying/computation of labels not visually explicit We further found that the way in which labels are copied from one model object to another should be more visually explicit. As our figures show, AToM^3 currently only shows `<SPECIFIED>` at labels where values are explicitly specified. In our opinion it would help if the labels that are specified to be copied there were displayed. A color-coding scheme could enhance user experience further.

Static semantics were hard to specify Often the programmer of a graph transformation might wish to specify rules that check the static semantics of a given visual model. For instance in our case we wanted to make sure that the "Order" relationship is a total ordering, without cycles. In AToM^3 we had to program this check manually in Python code. However for future versions we envision a more explicit mechanism in the form of negative application conditions that are evaluated not at transformation time but rather when the model is saved. In our particular case, the user could draw a circular dependency with the "Order" relation. The semantics would then demand that this pattern may not match when the validity of a given model is evaluated. Note that PROGRES [14] has some limited support for static checks of that kind.

Layouting not yet optimal We found the layout algorithms in AToM^3 to be suboptimal. Although in general best effort is made by the AToM^3 modelling environment, it still happens that nodes or edges overlap. Even in cases where no overlapping occurs, objects might be arranged in a way that to the tool user hardly makes sense. For instance in the case of Timeline, the time line should really be a line, with arrows starting on the left and ending to the right. There should be layout algorithms available which take such constraints into account. Maier and Minas have devised a generic layout algorithm for meta-model based editors [16] which promises to mitigate some of those problems.

7 Conclusion and future work

In this work we have shown that it is feasible to visually specify the transformation from the Timeline temporal specification formalism to finite automata suitable for runtime monitoring. The resulting automata can directly be used to generate efficient finite-state monitors for Java programs using an existing back-end for tracematches [7].

We believe that this explicit way of transforming specifications to monitors facilitates reasoning about and debugging of specifications. In particular, our translation is completely visual and provides a one-to-one mapping between entities in the Timeline specification and the resulting finite automaton. We plan to express this bi-directional relationship (i.e., backward trace-ability) between Timeline and finite automata in the form of Triple Graph Grammars [17]. These allow for the declarative specification of consistency relationships between graphs. This will enable us to easily relate errors at execution level to constraints in the original Timeline specification. We believe that our approach is yet another stepping stone on our path to bringing temporal specifications and runtime monitoring closer to widespread industry adoption.

In future work, we also plan to give a formal description of the actual tracematch code and how it is generated from the obtained finite state machines. We also wish to study the scalability of temporal specification formalisms with respect to the size of the pattern that needs to be specified. Last but not least, we want to apply our approach to real-world applications, for instance parts of the DaCapo benchmark suite [18].

Acknowledgements We wish to thank the anonymous reviewers for their pertinent comments. Further we thank the organizers of AGTIVE for making this symposium an unforgettable event. Last but not least, the first author wished to express his gratitude towards the Deutsche Forschungsgemeinschaft (DFG) and the AGTIVE steering committee for the awarded travel grant. The second author acknowledges partial support of this work by the Canadian National Sciences and Engineering Research Council.

References

[1] 1st to 7th Workshop on Runtime Verification (RV'01 - RV'07). (2001-2007) http://www.runtime-verification.org/.

[2] Avgustinov, P., Tibble, J., Bodden, E., Lhoták, O., Hendren, L., de Moor, O., Ongkingco, N., Sittampalam, G.: Efficient trace monitoring. Technical Report abc-2006-1, http://www.aspectbench.org/ (March 2006)

[3] Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. SIGPLAN Not. **42**(10) (2007) 589–608

[4] Bodden, E., Hendren, L.J., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In Ernst, E., ed.: ECOOP. Volume 4609 of Lecture Notes in Computer Science., Springer (2007) 525–549

[5] Martin, M., Livshits, B., Lam, M.S.: Finding application errors using PQL: a program query language. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. (2005) 365–383

[6] Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis, New York, NY, USA, ACM Press (2006) 133–144

[7] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: Object-Oriented Programming, Systems, Languages and Applications, ACM Press (2005) 345–364

[8] Smith, M.H., Holzmann, G.J., Etessami, K.: Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs. Proceedings of the 5th IEEE International Symposium on Requirements Engineering (2001) 14–22

[9] de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE) (2002) 174–188

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: European Conference on Object-oriented Programming. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 327–353

[11] Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2003)

[12] Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press (1977) 46–57

[13] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP '98: Proceedings of the second workshop on Formal methods in software practice, New York, NY, USA, ACM Press (1998) 7–15

[14] Schürr, A.: Developing Graphical (Software Engineering) Tools with PROGRES. In: International Conference of Software Engineering. (1997) 618–619

[15] Syriani, E., Vangheluwe, H.: Programmed Graph Rewriting with DEvS. [19] To appear.

[16] Maier, S., Minas, M.: A Generic Layout Algorithm for Meta-model based Editors. [19] To appear.

[17] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Tinhofer, G., ed.: WG'94 20th International Workshop on Graph-Theoretic Concepts in Computer Science. Volume 903 of Lecture Notes in Computer Science (LNCS)., Heidelberg, Springer Verlag (1994) 151–163

[18] Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, New York, NY, USA, ACM Press (October 2006) 169–190

[19] Nagl, M., Schürr, A., eds.: Applications of Graph Transformation with Industrial Relevance (AGTIVE), Kassel, Germany. In Nagl, M., Schürr, A., eds.: Applications of Graph Transformation with Industrial Relevance (AGTIVE), Kassel, Germany, Springer (October 2007) To appear.