Model-checking with the TimeLine formalism

Andrea Zaccara
University of Antwerp
Andrea.Zaccara@student.uantwerpen.be

Abstract

A logical model checker can be an effective tool for verification of software applications. However, these kind of tools are rarely used, as the requirements specification use a cumbersome textual notation, such as LTL. Previous work on the development of the TimeLine formalism shows how these tools can be made more accessible in many context. This work shows how the Timeline formalism can be used to visually define safety specifications in AToMPM, then transformed into finite automaton for static checks on the output trace for an example Chatroom application.

The final product allows for easily definable safety specification, which can produce powerful temporal checker using model-transformations and code-generation.

Keywords: TimeLine formalism, model-checking, state automaton, trace validation, model-transformation, AToMPM

1. Introduction

The verification of correctness in software application has always been an important issue in the context of software's quality. Model checking has already been proved to be an effective technique to find certain types of bugs, however it has been applied mostly to small embedded systems due to the complexity of defining requirements for the model. These are often specified using temporal logic, such as LTL, which allows for the specification of complex behavior of the tested software. However, these languages are defined using a cumbersome textual notation, which precludes its use to programmers with no expertise in software verification.

The TimeLine formalism was originally proposed [1] to remove the complexity from the definition and understanding of temporal properties. After defining a Timeline model, it was then converted to a finite state automaton, which could then be applied to a model of the system for the verification of those properties.

In later works this formalism has been applied to runtime monitoring[2]. Using a model-driven approach, a Domain Specific Modelling Language was defined in AToM^3[3]. This modeling environment allowed for a visual mapping to state automaton, using model transformations. This process leads to a monitor that is directly applicable to any java program, using the aspect-oriented programming paradigm. This allows for an immediate advantage in using the TimeLine definition for direct verification of a system, removing the manual transformation of the system to a model definition.

In this work a further application of the Timeline formalism is explored. The objective is the definition of temporal requirements in a new DSL in AToMPM[5] and the generation of automatons for the verification of the trace produced by the system under analysis.

The Timeline formalism is shown in detail in section 2. The transformation to state automaton is discussed in section 3, using an toy example. In section 4 the case study of the chat protocol is introduced and the Timeline requirements are explained. In section 5 the code generation phase is explained and a small performance comparison is executed. Section 6 discusses further development possible on this project and finally section 7 concludes this report.

2. The TimeLine formalism

Each model defined with the Timeline specification consists of a single time line, which is independent of other models. This enables modular reasoning, as each rule can check independently for different properties in the same context, reducing significantly the complexity of the rules.

A time line represents an ordered sequence of events. The first event is a unique start event, representing the time of start-up of the application. Also there is a similar End event, which is used to define the end of the time line, but in this context is not a reachable state of the application. All other events 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 a grey diamond shape.

- **required event**. A required event must occur, whenever its left-context on the time line was matched. Required events are denoted with a fuchsia pentagon shape.
- **fail events**. A fail event must not occur after its left context has matched. Such an event is denoted with a red X shape.

Along with these events, a time line can be augmented with constraints, restricting the matching process. A constraint is defined in-between two events and it holds a Boolean combination of propositions. It may include or exclude the previous and/or next event it is attached to.

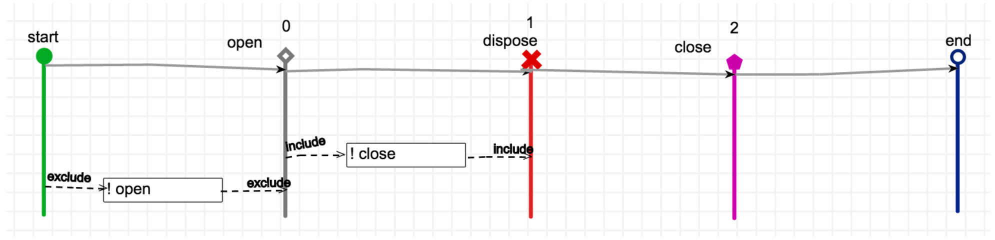


Figure 1: Timeline specification stating that any opened file should be closed and should not be disposed before closing it

In figure 1 an example of a TimeLine model is shown. This rule defines the following requirements:

1. A file must not be disposed as long as it is open.
2. Any open file is closed at some point in time, before the program exits.

The first requirement is checked as, whenever the context matches the *open* event, a *dispose* event will result in a fail if no close event happens beforehand.

The second requirement is checked as the specification sees the *close* event as required, as long as all previous regular events are matched, in this case only the *open* event.

Also, the initial constraint *!open* states that this rule is only interested in the last open event, as prior ones have already been handled at this point.

This example is derived from the one shown in figure 2, used in Bodden et al. paper, however as discussed in the other sections it has various differences both in the syntax and semantic.

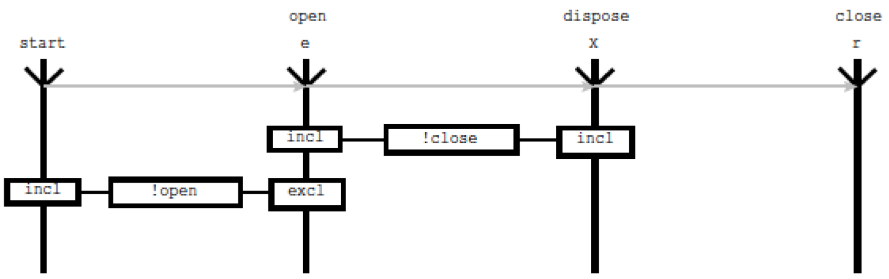


Figure 2: Timeline specification from Bodden et al. [2]

2.1. Abstract Syntax

The abstract syntax of the TimeLine formalism defined in AToMPM is shown in figure 3, using a class diagram.

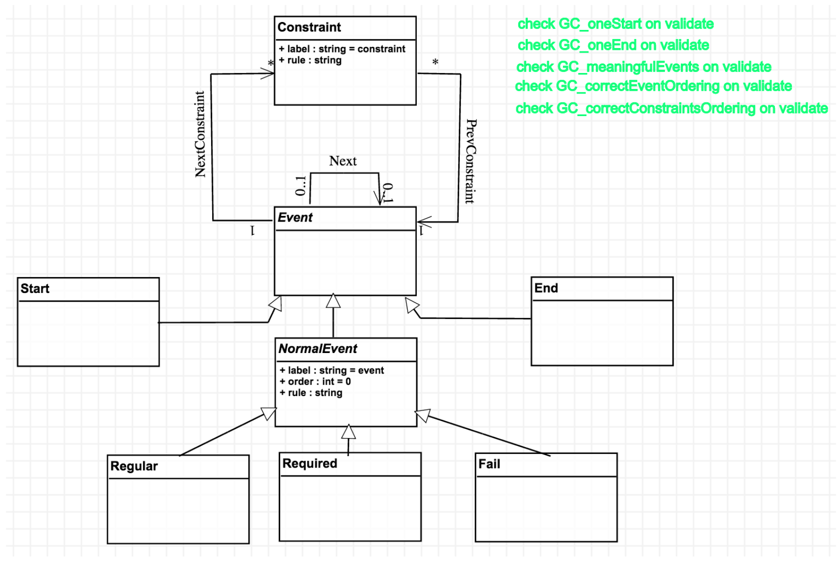

Constraint
+ label : string = constraint
+ rule : string
check GC_oneStart on validate
check GC_oneEnd on validate
check GC_meaningfulEvents on validate
check GC_correctEventOrdering on validate
check GC_correctConstraintsOrdering on validate
NextConstraint
PrevConstraint
Next
0..1
0..1
*
*
1
1
Event
Start
End
NormalEvent
+ label : string = event
+ order : int = 0
+ rule : string
Regular
Required
Fail

Figure 3: Abstract syntax of the TimeLine formalism in AToMPM

The *NormalEventEvent* is an abstract class with three possible specializations: *Start*, *NormalEvent* and *End*. The *NormalEvent* event is also an abstract class specialized by three types: *Regular*, *Required* and *Fail*. These events have a label and a rule attribute, used for defining the event to match on the trace.

The sequence of the time line is established via an ordering relation(Next). A further relation between events describes the constraints among them. Each constraint is modeled as an edge between two events. It can include or exclude its initial and/or final connected event. Furthermore it is labeled with a string label and a rule, which states the actual constraint expression.

For comparison the abstract syntax defined in AToM^3 by Bodden et al.[2] is shown in figure 4.

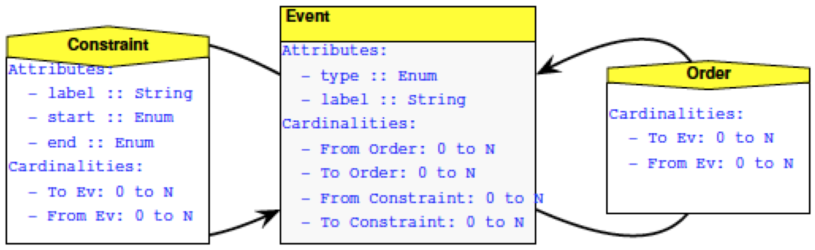


Figure 4: Abstract syntax of the TimeLine formalism in AToM^3[2]

The static semantics of the Timeline formalism imposes the following type checks on correct Timeline specifications.

1. Each time line must contain exactly one *Start* event and one *End* event.
2. Each time line must contain at least one *Required* or *Fail* event, as otherwise the model would be meaningless (matching context without checking anything).
3. Each time line must be fully connected by the *Next* relationship, forming a single Timeline. This is enforced in two ways:
   (a) using the cardinalities of events, for *NormalEvent*s there must always be an incoming and an outgoing *Next* association, for *Start* only one outgoing and for *End* only one incoming;
   (b) with a constraint on the *NormalEvent*'s order attribute, which must contain an increasing value in the Timeline sequence, to prevent looping sections.
4. When a constraint relation starts at an event *e1* and ends at *e2*, then *e1* must be smaller than *e2* in the Order.
5. The event directly connected to the *End* must not be of *Regular* type, as it creates context which cannot be used.

The first four constraints allow for a meaningful model definition, while the last one allows a cleaner definition of models and restricts the complexity of possible rules.

The model transformation are based on the assumption these constraints are verified. The most notable difference between this definition of the Timeline and the one from Bodden et al. can be seen on the option to have more than one subsequent *Fail* events on a time line. This is possible because the model transformation use a different mapping to automaton, which makes for a feasible implementation of these rules.

2.2. Concrete Syntax

The concrete syntax is shown in figure 5. It gives a distinctive visual representation to the events as described in the previous section.

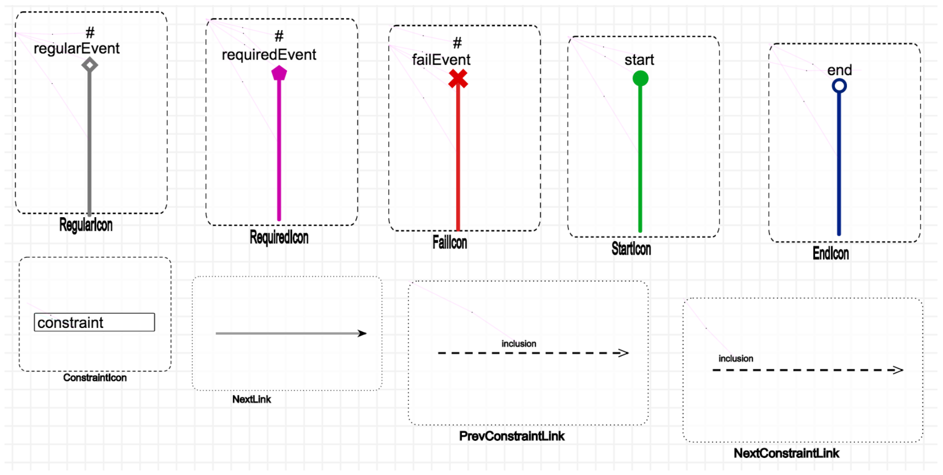
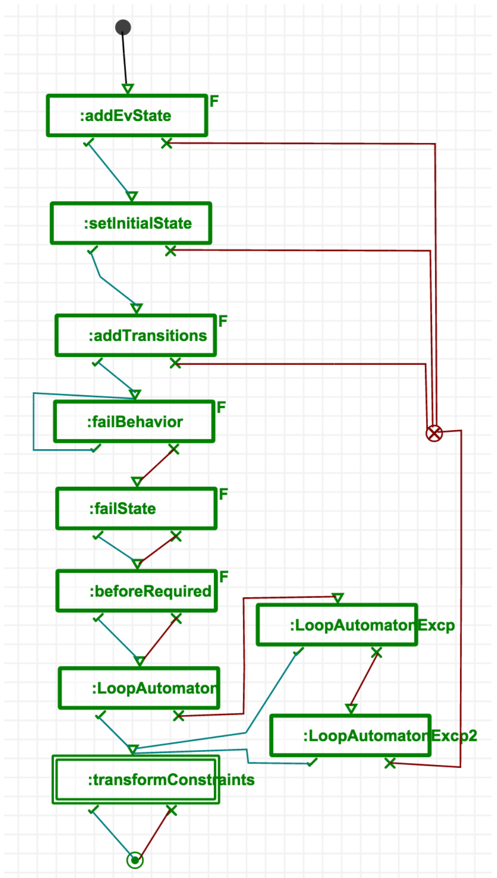


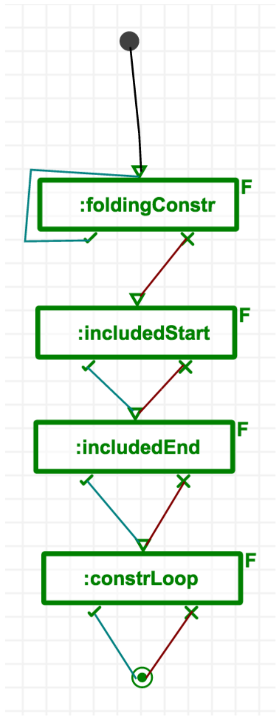
Figure 5: Concrete syntax of the TimeLine formalism in AToMPM

3. Model transformation to state automaton

The transformation from the TimeLine model to state automaton is defined using AToMPM model transformations, given a model which fulfills the constraint from section 2.1. The transformation is defined as a set of rules executed following a schedule, which is shown in figure 6.



(a) The main schedule



(b) The constraint management sub-schedule

Figure 6: The transformation schedule

The rules can be specified by defining a left-hand side (LHS) for matching objects in the model and a right-hand side (RHS) for defining the result of the transformation on the found objects. Additionally in some rules a Negative Application Condition (NAC) is used to define what to exclude from the possible matches.

The resulting model is composed of three Meta-models: the TimeLine, an augmented finite state automaton, where each node event transition contains a regular expression, and the Generic Graph used for tracing links between the two models. The expected meaning of the generated finite state automaton is that it will accept a sequence of events if the it violates the original specification.

The transformation rules are defined as follow:

1. **Add event state**(Figure 7). For each event of type *NormalEvent* a FSA state is added, with a link to the original event.

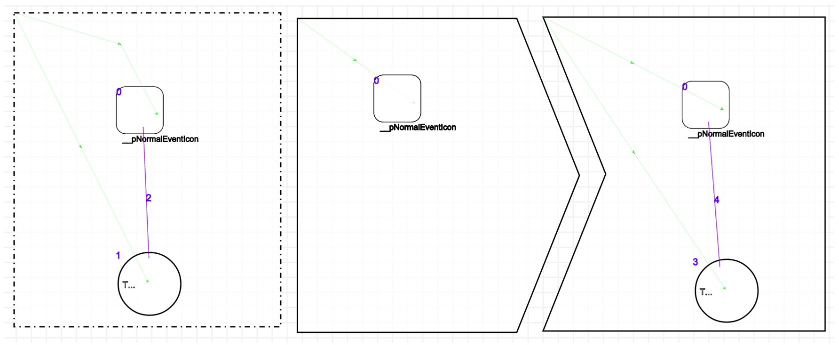


Figure 7: Add normal event states

2. **Set initial state**(Figure 8). A state defined as the initial state of the automaton is added, with a link to the *Start* event.

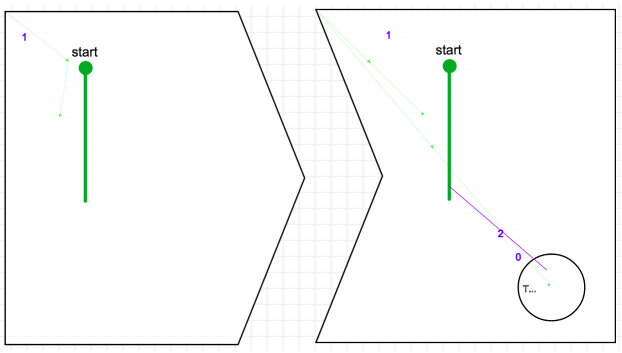


Figure 8: Add state for start event

3. **Add transitions**(Figure 9). For each pair of states connected to adjacent events, two complementary transitions are added from the state associated with the previous event. The looping transition is the negation of the next event rule, while the outgoing transition is the actual rule for the next event.
   This means that a state is reached when the associated event has occurred.

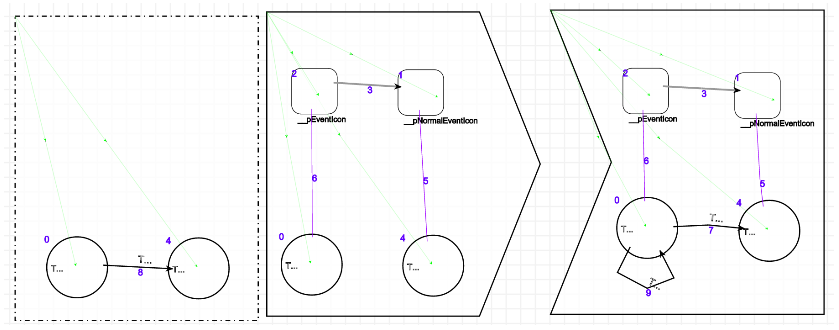


Figure 9: Add transitions between adjacent event

4. **Set fail behavior**(Figure 10). This rule is executed multiple times for each *Fail* event, with associated a state with an outgoing transition, and no incoming transitions from other *Fail* states, until all matches have been exhausted. The effect is to remove all outgoing and looping transition of every *Fail* event and create another path to reach the next state from the previous one.

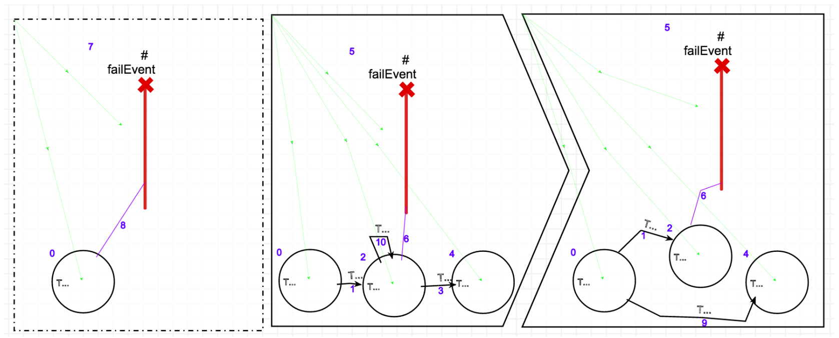


Figure 10: Define behavior of fail state

5. **Set fail state**(Figure 11). This simple rule just sets the states associated with a *Fail* event to being an accepting state. This is done here as the previous rule wouldn't be able to match a *Fail* event located at the end of the time line, with no successive events.

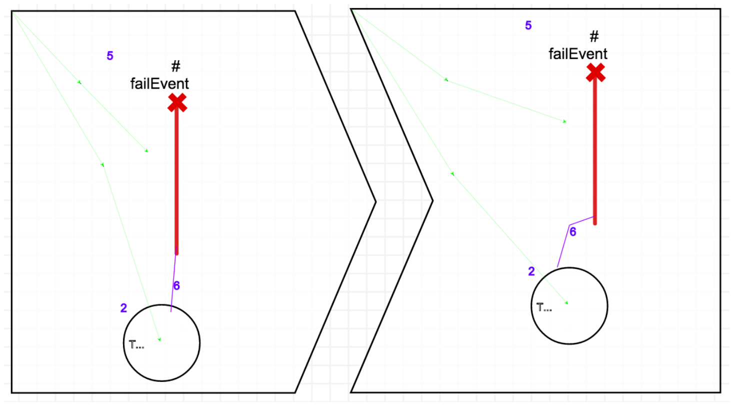


Figure 11: Set accepting state for fail event

6. **Set behavior for event before required**(Figure 12). Similar to the previous rule, this one sets the state preceding a required event as an accepting state. If the automaton is in this state after reading the full trace, the automaton will accept it, meaning the requirement is not satisfied.

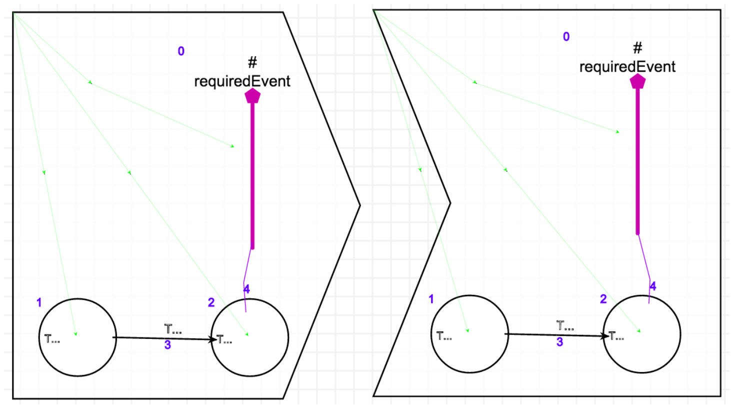


Figure 12: Set accepting state before requirement event

7. **Loop the automaton**. This rule can be applied in three different ways, depending on the structure of the time line:

   (a) **Normal case**(Figure 13). If the last event before the *End* is not a *Fail*, the same outgoing transitions of the initial state are added to its associated state. This allows for a looping automaton, which after validating a portion of the trace can start again as if from the initial state.

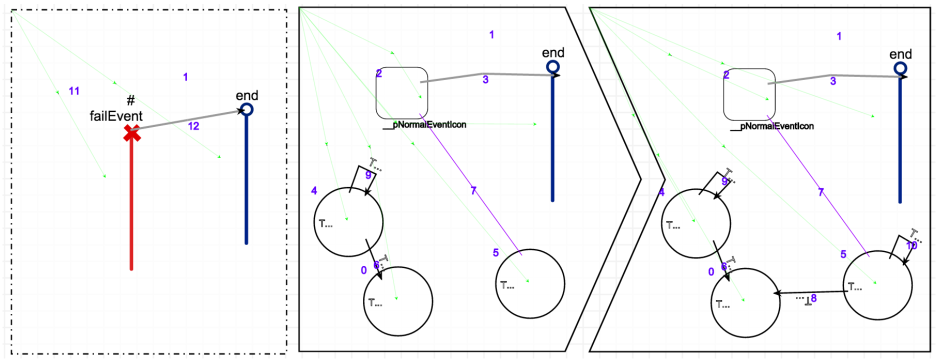


Figure 13: Loop rule for common cases

   (b) **Final event is fail**(Figure 14). The same procedure is applied to the event preceding the last one, in case it is a *Fail* event.

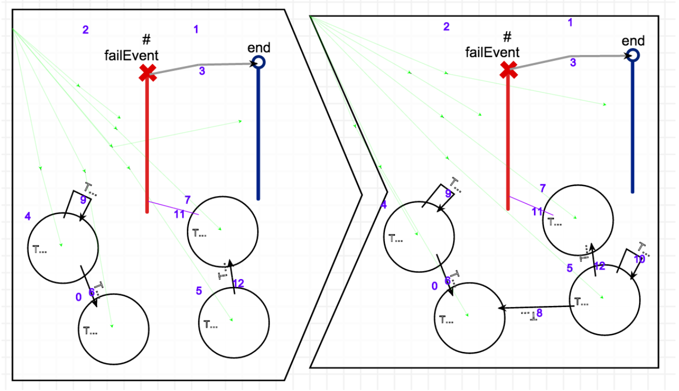


Figure 14: Loop automaton if last event is fail

   (c) **Final event is fail and there is only one previous event**(Figure 15). This final rules occurs in case the final event is a *Fail* and has the second state of the automaton as its previous one. (State 0 and 5 from figure 14 are the same state in figure 15)

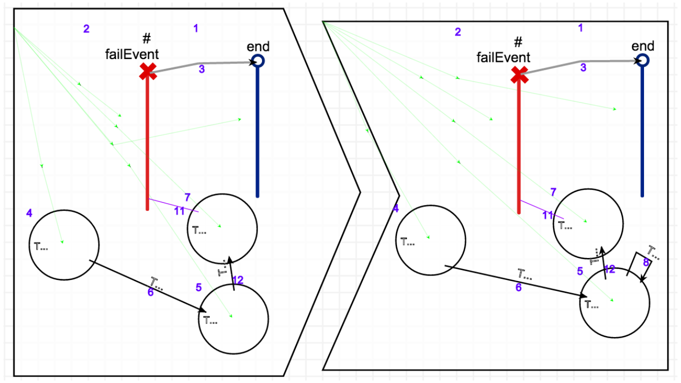


Figure 15: Loop small automaton if last event is fail

8. **Constraint transformation schedule**(Figure 6b). 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:
   (a) **Constraint folding**(Figure 16). Whenever we see two events e_i, e_j with a constraint between them and there exists an event e_j 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. This rule is executed until every constraint is connected to two directly connected states.

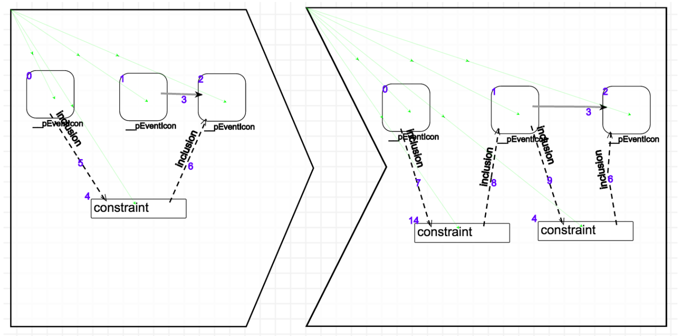
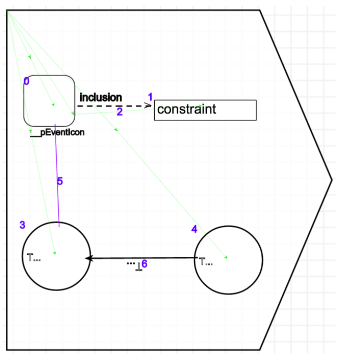
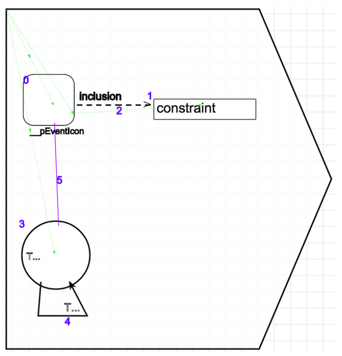


Figure 16: Constraint folding

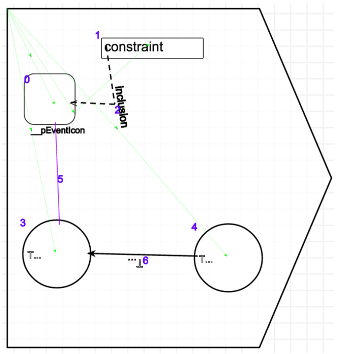
Figure 17: Applying constraints to automaton



(a) Inclusion of the constraint for initial event



(b) Inclusion of the constraint for looping event



(c) Inclusion of the constraint for final event

(b) **Constraint initial included**(Figure 17a). We find every state associated with an event included in the initial part of a constraint, and augment the incoming transition with the rule specified in the constraint.

(c) **Constraint loop**(Figure 17b). We find every state associated with an event in the initial part of a constraint, and augment the looping transition with the rule specified in the constraint.

(d) **Constraint final included**(Figure 17c). We find every state associated with an event included in the final part of a constraint, and augment the incoming transition with the rule specified in the constraint.

This model transformation is partially derived by the definition from Bodden et al.[2], which also uses model transformation in the AToM^3 environment.

Some differences are present in the semantic mapping to state automaton. In this project the states associated to the original events in the time line represent a sequence of event where it has occurred, while in Bodden et al. it had not occurred yet. This change allows for the definition of time lines containing consecutive fail events. An example is discussed in section 3.2.

3.1. Resulting Automaton

The application of this sequence of transformation to the model in figure 1 gives the automaton in figure 18 as a result. Instead, the same model using

Bodden et al. model transformation gives he automaton in figure 18 as a result.

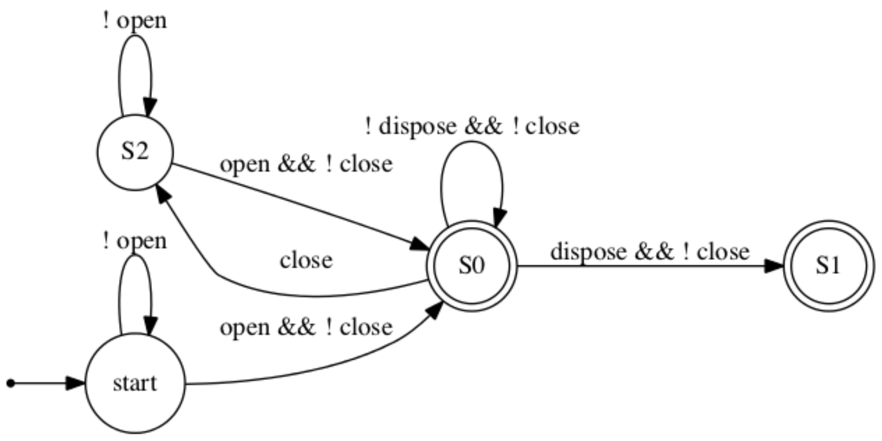


Figure 18: Resulting automaton for dispose and close requirements - AToMPM

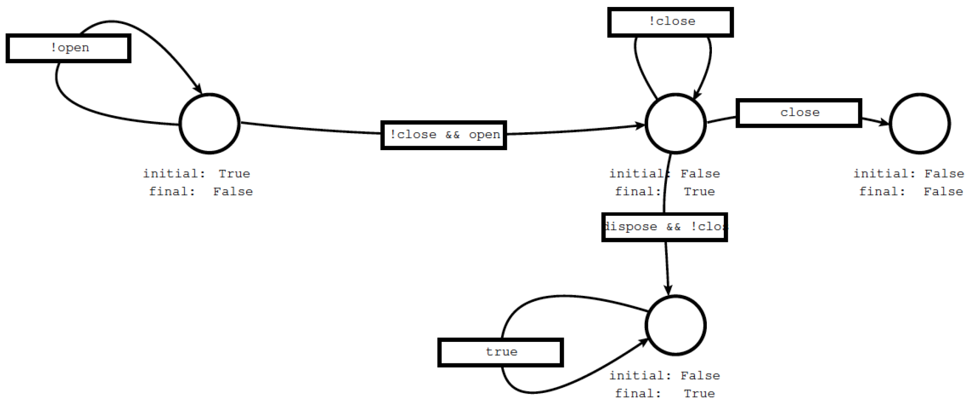


Figure 19: Resulting automaton for dispose and close requirements - AToM^3[2]

The resulting automaton are fairly similar, the most important difference is in the transition from **S2** back to **S0** in figure 18. This allows for a continuous evaluation of the trace for any number of iteration of the correct sequence of events.

3.2. Consecutive fail automaton

In figure 20 is represented a Timeline model with an additional fail event for the **delete** command. The meaning of this model is given by the mapping to state automaton shown in figure 21.

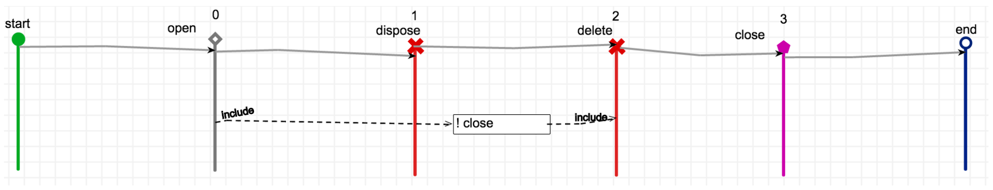


Figure 20: Timeline model with consecutive fail event

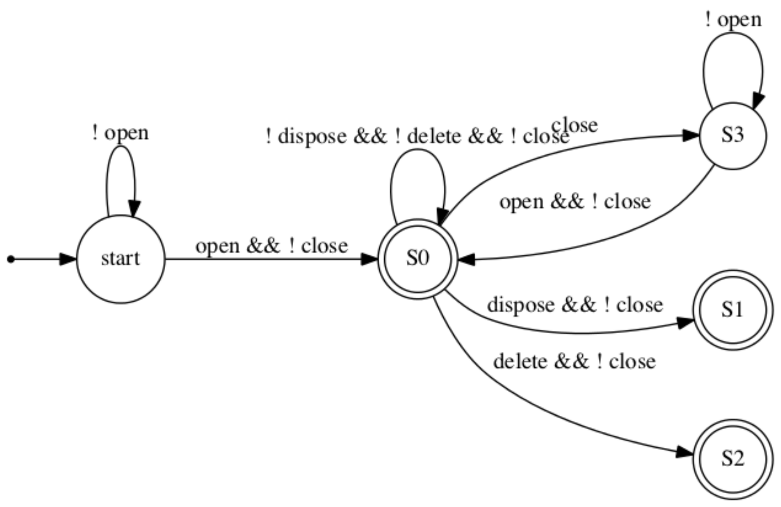


Figure 21: Mapping to FSA for model in figure 20

4. Chat protocol instance

The context used for the next sections is the chat protocol program used in the first MoSIS assignment of 2014-15, for the definition of a finite state automaton in Python. We consider the two requirements implemented and look at their specification with Timeline:

- Figure 22. On receiving a connection request, the chat room immediately makes a decision whether to accept the client or reject it. (originally req. 2)
- Figure 23 The sender cannot receive its own message after it sends it. (originally req. 7)

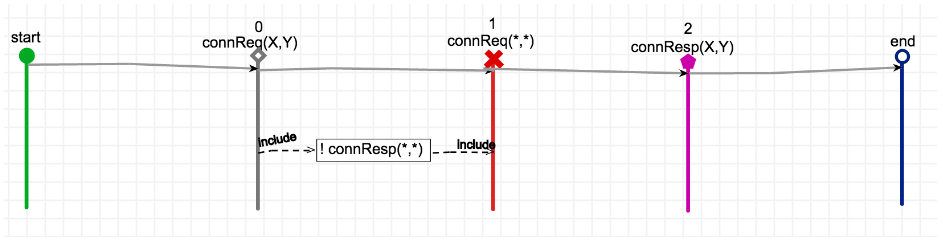


Figure 22: Timeline model of requirement 2

These models are fairly simple to understand, as they follow a similar definition as the previous examples. 2) A connection request from client **Y** to chat room **X** must always receive a connection response, and no other connection request should arrive in between. 7) A message **T** sent to the chat room **X** from client **Y**, must not be received by client **Y**.

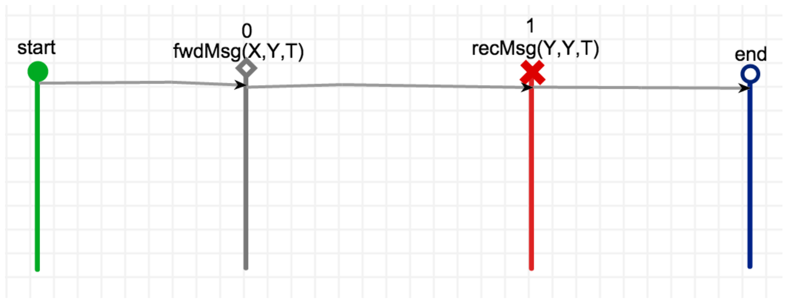
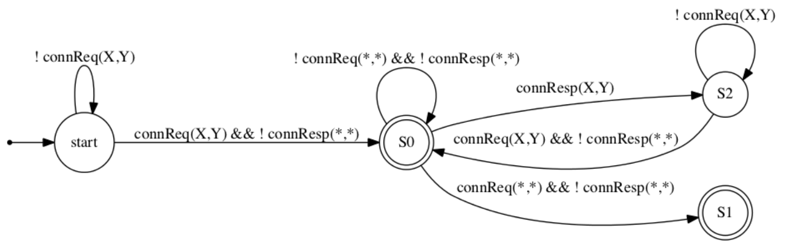


Figure 23: Timeline model of requirement 7

The actual rule is defined as a custom regular expression that can contain one or more of these syntactic components:

- **'! '** can be used to negate the subsequent matching,

- **’%’** is used to identify variables to be stored,
- **’?’** is used to check the text with a variable already, stored
- **’*’** is used when the text to check can be anything.
- **’[str1|str2]’** is used when the text to check can have two possible values.



! connReq(X,Y)
start
connReq(X,Y) && ! connResp(*,*)
! connReq(*,*) && ! connResp(*,*)
S0
connResp(X,Y)
! connReq(X,Y)
S2
connReq(X,Y) && ! connResp(*,*)
connReq(*,*) && ! connResp(*,*)
S1

Figure 24: Finite state automaton obtained from requirement 2

We look here at the rules specification for the requirement 2:

1. regular event, connection request from Y to X:

$$connReq(X, Y) <=> (CR\ \%X\%)\ RR\ \%Y\%. \tag{1}$$

2. constraint, any connection response:

$$connResp(*, *) <=>!\ (CR\ ?*?)\ [A|R]C\ ?*?. \tag{2}$$

3. fail event, any connection request:

$$connReq(*, *) <=> (CR\ ?*?)\ RR\ ?*?. \tag{3}$$

4. required event, connection response from X to Y:

$$connResp(X, Y) <=> (CR\ ?X?)\ [A|R]C\ ?Y?. \tag{4}$$

This set of rules is ported to the state automaton generated from the model transformation shown in figure 24. This automaton will accept this trace, which contains the the regular, followed by the required event.

```
## (Chat room 0) Received connection request from client 3.
(CR 0) RR 3.
## (Chat room 0) Accepted client 3.
(CR 0) AC 3.
```

5. Code generation

The next phase is the code generation from the automaton, to obtain an executable program to run the validation for the temporal specification of the model.

Using the AToMPM metadepth plugin, the FSA model is exported into a corresponding metadepth model. Using the EGL language the *network.py* script is generated from the model, containing the information on the automaton in a particular data structure.

When this script is executed, it launches the *ParserGenerator.py* with the information relative to the automaton, and another script is produced, *fsm.py*, containing the automaton implementation in the class *AutomatonScanner*.

This can then be used by calling the *tester.py* script, which executes the parsing for the *fsm.py* automaton. Given four traces, one that is completely correct, one with errors going against requirement 2, one with errors for requirement 7 and one with both kinds, it was tested with the two automatons, which give a positive result as errors are found and otherwise the requirements are validated.

5.1. Optimization and performance analysis

To obtain a high performance in the generation of the temporal logic checker, the most important phase is the pre-processing of the model, so that it generates an efficient finite state machine. In this project performance was not the focus, so the generated checker is a simple python script composed of if statements. Furthermore, each line of the trace is read completely and checked as a whole until a match is found, instead of reading progressively character by character.

To have a frame of reference, I compared the average time required for each automaton with the results obtained by the equivalent python program on the four traces. The python program was written as part of the MoSIS assignment #1 and contains the implementation of the regular expression which validates the requirements discussed. This implementation parses the trace character by character.

The results from table 1 show that this implementation of the temporal checker is up to 60 times faster then the manual automaton implementation, even considering that each requirement is checked separately. However these results vary depending on the size of the trace, the complexity of the expression definition and the presence of one or both errors, and in part it is probably influenced by the inefficient string matching provided by Python, used for single characters comparison. All files have approximately the same size (∼400 lines and 17-19,000 characters).

	FSA	**TC**	**TC2**	**TC7**
correct	55.66 ms	**0.84 ms**	0.41 ms	0.43 ms
error 2	3.95 ms	**0.15 ms**	0.07 ms	0.08 ms
error 7	3.80 ms	**0.97 ms**	0.49 ms	0.48 ms
error both	3.62 ms	**0.14 ms**	0.07 ms	0.07 ms

Table 1: Performance comparison - Finite State Automaton and Temporal Checker

Also we need to consider the scope of the two analysis. The FSA uses regular expression with strict requirements, which considers the whole trace and set of requirements for the validation. The temporal checker only considers a sequence of events using the commands specified, while others are ignored.

So it can be seen as less powerful, however with the current implementation of the temporal checker model transformation and code generation a useful tool can be produced for fast model checking, with only 5-10 minutes of modeling on the Timeline formalism.

6. Future works

Using this work as a starting point for the expansion of the Timeline formalism and its applications, time should be dedicated to improve the transformation from automaton to an efficient regular expression recognition algorithm, also by combining different rules into a single one.

More focus should be given to the rule in the Timeline formalism, as the current grammar available is limited. Also more modularity could be given to the rules by using an additional element containing the complete alphabet, which is then referenced from the events.

Finally more experiments could be performed to address the scalability of the model, both on the Timeline for easy understanding of the models and on the generated model checker execution time on big traces.

7. Conclusion

In this work I have ported the Timeline temporal specification formalism to AToMPM and defined models transformations to finite automaton suitable for trace checking. The resulting automaton can directly be used to generate programs for temporal checking of the defined specification, given a well defined trace.

This definition of temporal requirements facilitates reasoning about and debugging of specifications. In particular the visual representation of the model transformation provides a one-to-one mapping between entities in the Timeline specification and the resulting finite automaton.

This tool can be considered as a useful complement to add in the process of software correctness validation, to easily obtain additional information on the status of the system.

References

[1] M. H. Smith, G. J. Holzmann, K. Etessami, Events and constraints: A graphical editor for capturing logic requirements of programs, in: aaa (Ed.), Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on, IEEE, 2001, pp. 14–22.

[2] E. Bodden, H. Vangheluwe, Transforming timeline specifications into automata for runtime monitoring, in: Applications of Graph Transformations with Industrial Relevance, Springer, 2008, pp. 249–264.

[3] J. Lara, H. Vangheluwe, Atom3: A tool for multi-formalism and meta-modelling, in: R.-D. Kutsche, H. Weber (Eds.), Fundamental Approaches to Software Engineering, Vol. 2306 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 174–188.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of aspectj, in: ECOOP 2001Object-Oriented Programming, Springer, 2001, pp. 327–354.

[5] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, Atompm: A web-based modeling environment.