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 paper describes how this formalism can be used to visually define safety specifications, and which is the meaning by mapping it to state automaton.

Keywords: TimeLine formalism, model-checking, state automaton, run-time monitoring

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. These techniques have not been incorporated in everyday software development processes.

This is in part due to the complexity of the algorithms needed to verify the requirements, but another important factor is 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.

To cope with this limitation, the TimeLine formalism has been proposed [1]. This formalism was designed to remove the complexity from the definition and understanding of temporal properties.

To give meaning to models defined with the TimeLine formalism, it was then mapped to Buchi automatons. The generated automaton can then be used to check that the original property is true for the model of the system. However, the model checking is not directly executed on the program, but on a model obtained by abstracting the existing code. This process, shown in figure 1, still requires that the system is translated into the appropriate model, before the automaton can be used to verify the requirements. The resulting model can then contain errors not present in the source code, or it can remove errors actually present in it.

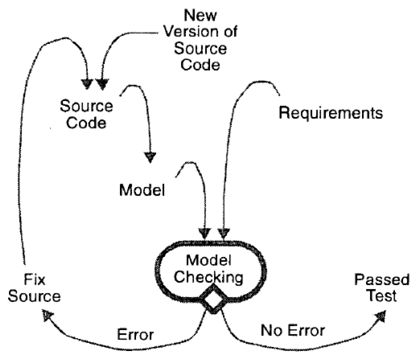

Figure 1: The typical process of model-checking

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.

For both the TimeLine formalism and the transfromation this paper refers to Bodden et al. definitions[2]. The Timeline formalism is shown in detail in section 2. The transformation to state automaton is discussed in section 3. Section 4 presents the starting point for the project implementation.

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.

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. 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 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 start and/or end event it is attached to.

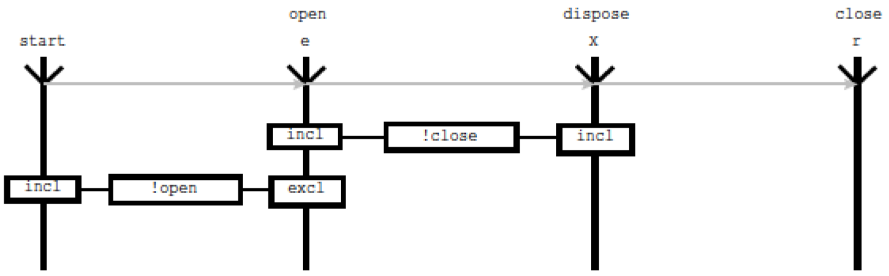

Figure 2: Timeline specification stating that any opened file should be closed and should not be disposed before closing it

In figure 2 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.

2.1. Abstract Syntax

The abstract syntax of the TimeLine formalism defined by Bodden et al.[2] is shown in figure 3, using a class diagram.

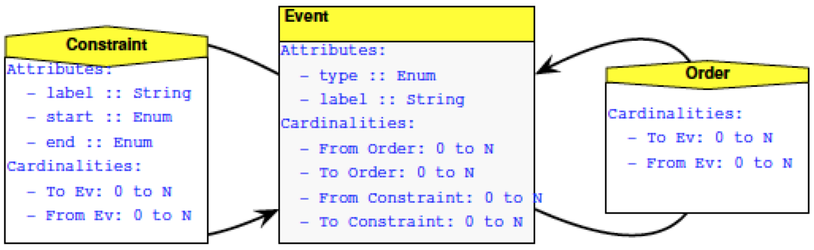

Figure 3: Abstract syntax of the TimeLine formalism in AToM^3

The Event is 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 automaton.

The sequence of the time line is established via an ordering relation. 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 the event at its start and/or end. Furthermore it is labeled with a string label, stating the actual constraint expression.

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

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 *e1* and ends at *e2*, then *e1* must be smaller than *e2* 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 first four constraints allow for a meaningful model definition, while the last two allow for a simpler definition of the model transformation and to restrict the complexity of any given rule. For example, if in the rule defined in figure 2 the fail event *delete* was added, meaning we don't want to delete the file we are working on, it would require to check for both fail events at runtime. The same result can be accomplished with two independent rules, which are also more easy to understand.

The model transformation are based on the assumption these constraints are verified.

3. Model transformation to state automaton

The transformation from the TimeLine model to state automaton is defined using AToM^3 model transformation. These can be specified by defining a left-hand side (LHS) for matching objects in the model and right-hand side (RHS) for defining the result of the transformation of the found objects.

The resulting model is composed of three Meta-models: the TimeLine, the finite state automaton and the Generic Graph used for tracing links between the two models. The expected meaning of the model is that it will accept a sequence of events if it violates the specification.

The transformation is performed in eight sequential transformation stages.

1. *Add an end event.* The end event is inserted after the last event in the sequence.
2. *Add states.* For each event one state is created, which reflects the point in time immediately before the associated event occurs.

The state changes depending on the type of event, for example the state associated with a required event is defined as an accepting state, as the monitor has not seen the required event yet. The start event is not associated with a state.

3. *Marking the initial state.* The initial state of the automaton is specified as the state connected to the event immediately after the start event.
4. *Adding transitions.* For each state one or more transitions are added depending on the next event.
5. *Folding constraints.* For any constraint that links to two states which are not immediate successors, it is divided between all the events in-between the linked ones.
   To do so, the single constraint is replaced with new constraints linked to every immediate successor in the sequence between the two original events.
6. *Applying the constraints.* The constraints are applied to the transition between states and to the self-loop transitions if it is included on the associated state.
7. *Implement semantics of fail events.* The fail state is represented as two states, one is the same as the other events and represents the point in time immediately before the associated event occurs. The other is an accepting state, where the automaton stops if the fail event occurs.
8. *Removing the events.* These leaves only the resulting finite state automaton.

As an example of the model transformation, in figure 4 the implementation of step 4 is shown in the AToM^3 environment.

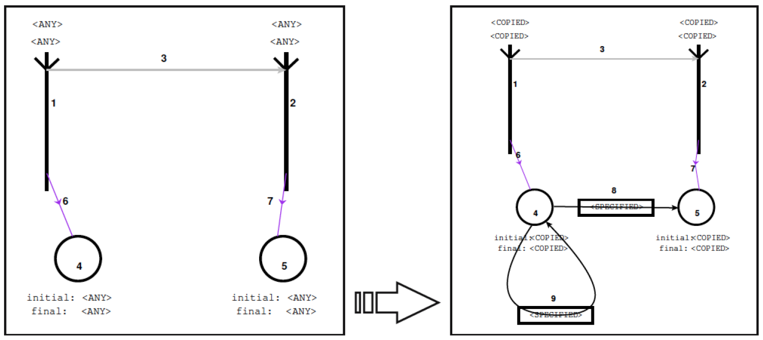

Figure 4: The "Creating transition" step

The detailed model transformation representation for each step is available in the original paper from Bodden et al.[2].

The application of this sequence of transformation to the model in figure 2 gives the automaton in figure 5 as a result.

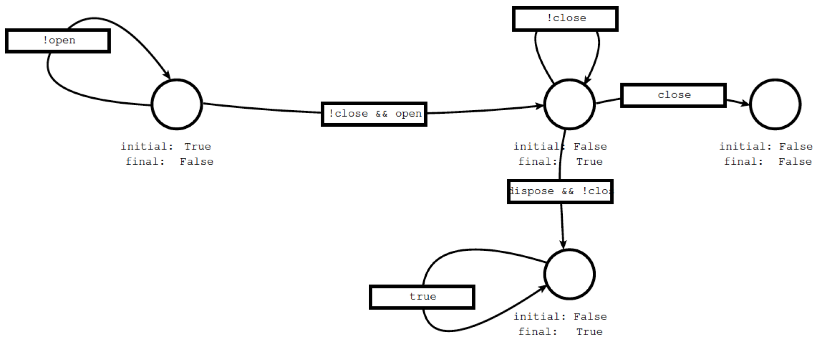

Figure 5: Resulting automaton for dispose and close requirements

This automaton can be converted to a monitor for runtime monitoring, which can then be applied using aspect-oriented programming with AspectJ[4]. Aspect-oriented programming allows for the definition of concerns, which need to be applied to all the components of a system. An common example is logging, which is a cross-cutting concern that can be implemented once and then applied to all classes and methods using this paradigm. This application is done during the compilation of the source code, allowing for an easy generation of monitors for any java program.

4. TimeLine as an Abstraction of Regular Expressions

For this project, I will implement the TimeLine formalism, by defining a domain-specific language in AToMPM[5]. Compared to the version of AToM^3 used in previous works, it enables for a more complete definition of the transformation rules, thanks to negative application conditions (NACs) and the definition of the transformation schedule.

The expected result is the generation of finite state automaton, which can be used to parse the trace of a program to check that one specific program executions the defined specification is verified. It is not as powerful as runtime monitoring, but it can be applied to any program with a decent use of event tracing. The case study for this project will be a communication protocol of a client-server 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.