Traviando - a Trace Analyzer to Debug Simulation Models [*]

Peter Kemper, Carsten Tepper
Informatik IV, Universität Dortmund
D-44221 Dortmund, Germany
Email: peter.kemper@udo.edu
carsten.tepper@udo.edu

Abstract

Verification and validation of simulation models are crucial steps to the success of a simulation project. Traces are a common and powerful mean to document the dynamic behavior of a model and are generated by most simulation engines. In this paper, we describe techniques and corresponding tool support that helps a modeler to gain insight in the dynamic behavior of complex simulation models based on trace analysis. We propose to visualize traces by message sequence charts. We use a common modal logic, namely the linear time logic (LTL), to identify states of interest and a pattern system to make specification of formulas more productive. The proposed techniques are implemented in Traviando, a stand alone tool with an open XML interface to import traces from various modeling environments, e.g., the APNN toolbox, the ProC/B toolset and Mobius.

Keywords: modelchecking, debugging simulation models, visualization and analysis tool

1 Introduction

Traces that document the dynamic behavior of a system by a sequence of states and events are considered in many different settings. For instance, they are used as input for a trace-driven simulation, as data to obtain stochastic representations of work load characteristics as distributions, markovian arrival processes and alike. Traces are also analyzed with respect to some qualitative behavior in runtime verification, where a system could be a multi-threaded java program that is monitored at runtime. Finally traces are generated from models by a simulator, a model checker, or some monitoring tool for verification, validation, debugging or animation purposes.

We focus on traces that are generated from a simulation model by discrete event simulation. Such a trace contains different pieces of information, information with respect to time and with respect to a causal order of events. While timing information is usually used for estimates of performance figures, the causal order and qualitative properties are of interest in the search for flaws in a verification and validation phase of a simulation project. Qualitative properties of dynamic behavior are usually described in a modal logic and much research has gone into the development of a variety of useful and expressive logics for dynamic systems in general. Common examples are computational tree logic (CTL) as a branching time logic and linear time logic (LTL) as a linear time logic, see Clarke et al [4] as a recent textbook. Since a trace is based on a single and finite sequence of observations, branching time logics like CTL are not a natural choice and many concerns with respect to the infinite behavior of a system do not matter. For trace analyis, some work has been done in the area of runtime verification. Havelund et al describe in [8] how to perform model checking for linear time logic (LTL) formulae on a trace by specially adjusted algorithms while Alur and Yannakakis describe in [1] how a canonical LTL modelchecking approach would assess a trace of a concurrent model. Uchitel et al [13] provide tool support for scenario-based specification, synthesis and model-checking of models of dynamic systems with the help of message sequence charts (MSCs).

For practical applications, modelchecking has a number of challenges. First, a human computer interface is required that is easy to use and that helps a practitioner to be confident on the property he or she specified in a formal notation like LTL. For instance, a formula like $G((q \wedge \neg r \wedge Fr) \rightarrow (\neg pUr))$ with atomic propositions p, q, and r is not crystal clear to a non-expert[1]. Dwyer et al. [6, 7] recognized regularities in the use of logic formulas for modelchecking and organized those into a pattern system. Second, given a model and a formula, efficient algorithms are necessary to decide whether a model fullfills that formula or not. Much

[*]This material is based upon work supported in part Deutsche Forschungsgemeinschaft, SFB 559.

[1]It describes that p is false between q and r; for the semantics of LTL operators see Section 3.

research has gone into that problem and impressive results have been obtained by so-called symbolic modelchecking techniques. However, in case of traces, modelchecking retains rather simple from a practical point of view since there is only a single and finite sequence that has to be analyzed. Third, once a result has been obtained, it must be represented or visualized in a way that maximizes the insight it may give to a modeller.

In this paper, we contribute to those challenges in the following way. We describe a new tool that visualizes traces by a variant of MSCs. It provides a pattern system with a formula editor for LTL to describe properties of interest. For a given set of formulas and propositions, it evaluates which states fulfill which formulas and visualizes those properties in the MSC representation. In addition, we provide a number of visualization features that help a user to retain an overview on the dynamic activity [12], and to identify fragments of traces that are critical for common performance measures in process-oriented simulation models [10]. The tool we present has an open XML interface for traces and has been applied with several modeling frameworks including the ProC/B toolset [2], the APNN toolbox [3] and the multi-paradigm multi-solution framework Mobius [5].

The rest of the paper is structured as follows. In Section 2, we briefly define basic terminology for traces and MSCs. Section 3 is devoted to LTL modelchecking of traces and a pattern system for the specification of formulas. In Section 4 we introduce our new tool for trace visualization, its architecture and main functionality.

2 Definitions

We are interested in the analysis of finite traces. Formally, we consider a trace as a sequence $\sigma = s_0 e_1 s_1 \dots e_n s_n$ of states $s_0, \dots, s_n \in S$ and events $e_1, \dots, e_n \in E$ over some (finite or infinite) sets S, E for an arbitrary but fixed $n \in \mathbb{N}$. For elements of S, we assume an equivalence relation denoted by "$=$". Note that events are irrelevant in the following formal treatment, but events are important pieces of information in a trace in order to document not only the state of the system but also what happens. Hence, we keep events within our considerations.

Message sequence charts. If a trace results from a set P of interacting, concurrent processes, we can assume that a state $s \in S$ is a vector of local states $s_p \in S_p$ of processes $p \in P$, i.e., let $S \subseteq \times_{p \in P} S_p$ and events may not necessarily change the state of all processes. Hence we can distinguish so-called local events that change the state of only a single process from those that change two or more processes. By adding more information, we can distinguish among send and receive events and synchronized events. For that scenario, message sequence charts (MSCs), [9], or UML sequence diagrams are common and related graphical representations. In the following, we focus on a variant of MSCs that adds (undirected) synchronized events to the usual send, receive and local events of MSCs and that adds state information.

Definition 1. *An MSC M is defined as a tuple $M = (V, <, P, M, K, N, S, G, s_0, m)$, where V is a finite set of events, $< \subseteq V \times V$ is an acyclic relation, P is a set of (MSC) processes, M is a set of message names, $L : V \to P$ is a mapping that associates each event with a process, $K : V \to \{s, r, u, l\}$ is a mapping that describes the kind of each event as send, receive, synchronized or local, $N : V \to M$ maps every event to a name, $m = m_{sr} \cup m_u$ is a relation called matching with $m_{sr} \subseteq V \times V$ that pairs send and receive events. Each send is paired with exactly one receive and vice versa. Events v_1 and v_2 can be paired, only if $N(v_1) = N(v_2)$. $m_u \subseteq 2^V$ relates up to $|P|$ processes by a synchronized event. S is a finite set of states with $s_0 \in S$ as initial state and $G : V \to S$ determines the successor state after event $v \in V$.*

Note that events cannot repeat due to $<$, but names of events and states may occur multiple times in a trace. We restrict ourselves to well-defined MSCs in the sense that we assume a total order on $V = \{v_1, v_2, \dots, v_n\}$ such that $\sigma = s_0 v_1 s_1 \dots s_n$ with $G(v_i) = s_i$ that is consistent with "$<$", i.e., if $v_i < v_j$ then $i < j$ for all $i, j \in \{1, 2, \dots, n\}$. The restriction does not exclude any relevant cases in practice, since we would start from a totally ordered sequence σ and represent it by an MSC. Synchronized events are represented by undirected edges (therefore we choose u for notation), while send-receive events yield directed arcs. The definition of $MSCs$ is a union of what is usually needed by different modeling formalisms. We currently use local events, send and receive events for traces generated by the ProC/B toolset, and local and synchronized events for traces generated by the APNN toolbox and Mobius. Figure 1 shows an example trace of six processes with local and synchronized events, which was generated from a Mobius model for a server system with two customer classes A and B where the server is subject to failure and repair. The model is compositional and each submodel matches a process in the MSC. A process is visualized by a vertical line with its name on top. Local events are represented by small dots, synchronized events by horizontal lines, and a state $s = G(v)$ is visualized in an additional window for a selected event v. Line thickness and colorings reflect results of modelchecking a particular formula. The vertical order of events (top-down) follows the order of events in a visualized sequence σ.

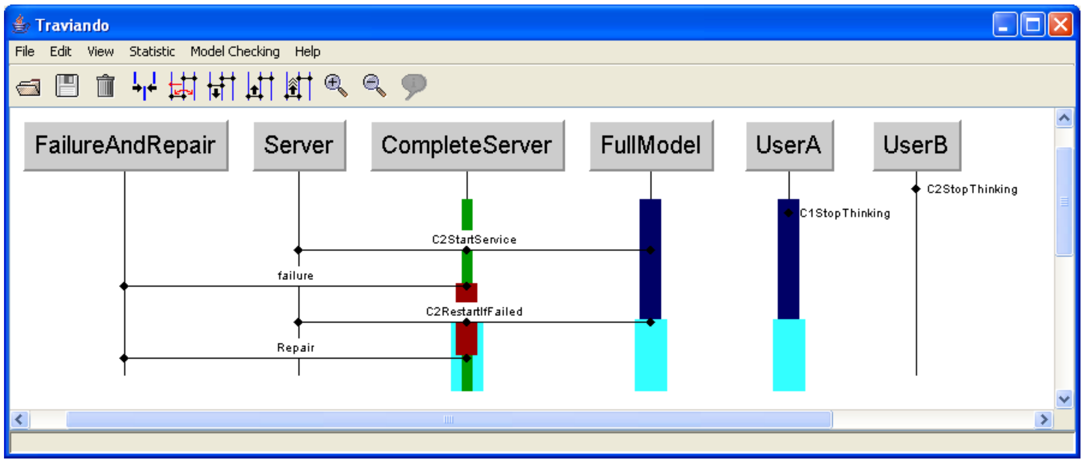


Figure 1. MSC of Example Trace

3 LTL modelchecking

Given a sequence σ, it is interesting to identify what properties of the system are present or absent. Modal logics of various kinds have been developed to analyze dynamic systems. In our case, a single and finite sequence would be considered a trivial case in concurreny theory and a linear time logic like LTL seems a reasonable choice for our purpose.

We assume a set of atomic propositions AP, such that an $a \in AP : S \rightarrow \{tt, ff\}$ evaluates states $s \in S$ to true or false or in other terms $s \models a$ iff $a(s) = tt$. Since we allow for a relation $=$ on S, AP has to be consistent with $=$, i.e., we require for $s, s' \in S$ that if $s = s'$ then $a(s) = a(s')$ for all $a \in AP$. This requirement is trivial for equality; it becomes relevant if some form of bisimulation is considered as “=”.

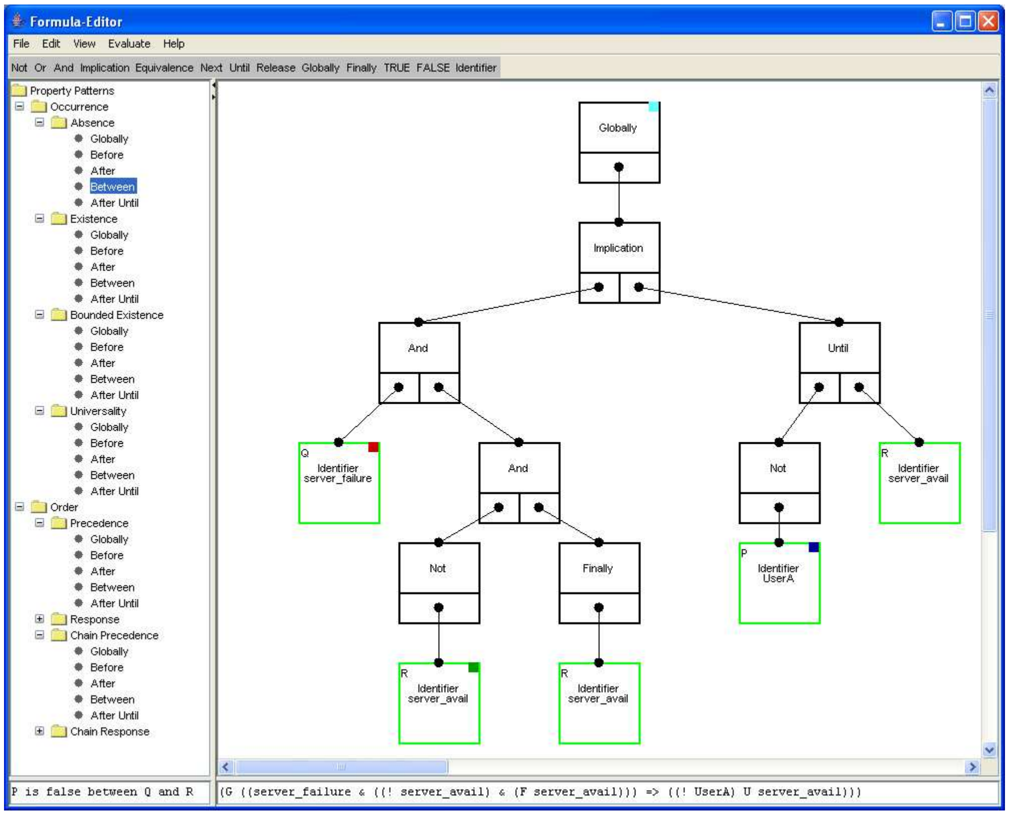
LTL considers the behavior of a system as set of sequences and has formulas of the form Af where f is a path formula. Following [4], an LTL path formula can be of two kinds.

- If $p \in AP$, then p is a path formla.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$,Xf,Ff,Gf,fUg, and fRg are path formulas.

The logic connectives implication and equivalence follow from the boolean operators in the usual manner, $f \rightarrow g \equiv \neg f \vee g$ and $f \leftrightarrow g \equiv (f \rightarrow g) \wedge (g \rightarrow f)$.

We mildly adjust the semantics of LTL for the case of a single finite sequence that is usually a prefix of a potentially much longer or infinite sequence. Hence for operators that refer to future behavior, we can be optimistic or pessimistic which we formally address by an additional artifical pair $e_{n+1}s_{n+1}$ that is attached to σ and we assume that atomic propositions are defined for s_{n+1} as well. For a desired property $\Psi \in AP$, we say we are optimistic if we define $s_{n+1} \models \Psi$ and pessimistic if $s_{n+1} \not\models \Psi$. With that extension we define the semantics of LTL operators for a sequence $s_0, \ldots, s_n$ and states $s_i, i = 0, 1, ..., n+1$ as follows:
$s_i \models \neg f$ iff $s_i \not\models f$,
$s_i \models f \vee g$ iff $s_i \models f$ or $s_i \models g$,
$s_i \models f \wedge g$ iff $s_i \models f$ and $s_i \models g$.
If $i \leq n$, we define
$s_i \models Xf$ iff $s_{i+1} \models f$,
$s_i \models Ff$ iff there exists a j, $i \leq j \leq n+1$ such that $s_j \models f$,
$s_i \models Gf$ iff for all j, $i \leq j \leq n+1$ holds $s_j \models f$,
$s_i \models fUg$ iff there exists a k, $i \leq k \leq n+1$ such that $s_k \models g$ and for all j , $i \leq j < k$ holds $s_j \models f$,
$s_i \models fRg$ iff for all k, $i \leq k \leq n+1$, for all j, $i \leq j < k$ holds $s_j \not\models f$ then $s_k \models g$.
Finally, we define for $i = n+1$
$s_{n+1} \models Xf$ iff $s_{n+1} \models f$,
$s_{n+1} \models Ff$ iff $s_{n+1} \models f$
$s_{n+1} \models Gf$ iff $s_{n+1} \models f$
$s_{n+1} \models fUg$ iff $s_{n+1} \models g$
$s_{n+1} \models fRg$ iff $s_{n+1} \models f \wedge g$

Patterns. Dwyer et al. [7] suggest a pattern system that integrates several logics and provides a common taxonomy. It classifies properties into two classes, one for *occurrence* and one for *ordering* (of states or events or properties). Both classes are further refined into four subclasses each. *occurrence* contains *absence*, *universality*, *existence* and *bounded existence*. *ordering* consists of *precedence*, *response*,*chain precedence*, and *chain response*. Patterns are elements of those classes and each pattern has a scope that defines where



```
P is false between Q and R
```

```
(G ((server_failure & ((! server_avail) & (F server_avail))) => ((! UserA) U server_avail)))
```

Figure 2. Graphical formula editor with pattern system

it applies, i.e., a scope gives the first and last state where a formula (as a refinement of a corresponding pattern) should be evaluated. Scopes are categorized into the following five classes: *global*, *before*, *after*, *between* and *after-until*. Dwyer et al. derived their pattern system from a substantial empirical study on the use of modal logics in verification and validation of systems. They analyzed about 500 example specifications taken from at least 35 different application areas and experienced that very few kinds of formulas occur in practice. They observed that patterns *response*, *universality*, and *absence*, all with scope *global*, cover about 80% of the considered cases.

This motivated us to integrate that taxonomy into an editor for LTL formulas. Fig. 2 shows the editor window in the pattern system of Traviando with 3 representations of formula f of pattern *absence* with scope *between*. Formula $f = G((q \wedge \neg r \wedge Fr) \rightarrow (\neg p U r))$ has atomic propositions $p = User\_A$, $q = server\_avail$, and $r = server\_failure$ that are formulated by equalities/inequalities on arithmetic expressions with state variables, e.g., $server\_avail$ is defined as $avail = 1$ where $avail$ is a state variable of submodel $CompleteServer$. p is defined as $C1WaitsForServer + C1WaitsForUser + C1Thinking = 1$. So state variables like $C1Thinking$ are used to make state information $s \in S$ accessible to a specification of properties in atomic propositions. The upper right window in Fig. 2 shows the formula in a graphical, tree-type representation that is used to create and refine formulas, to assign atomic propositions to the leafs of the structure, and to select colors for its graphical representation. The lower right window gives the common formal representation for users that are more familiar with that type of notation. Finally, the lower left window describes the formula in technical prose that is initially derieved from predefined phrases associated with patterns and that can subsequently be worked on by a user with further comments. Note that those representations describe the same information, but should complement each other for human understanding. The tree-type representation is what is used as input for the modelchecking algorithms.

Algorithm. Since we consider the special case of a single and finite σ, an algorithmic treatment of LTL path formulas is straightforward. Atomic propositions are evaluated for individual states. X, F, G, U, R operators are evaluated backwords starting at state s_{n+1}. Note that by definition the

evaluation at s_{n+1} for X, F, G, U, R is immediate. With known results at s_{n+1}, we can make use of the following properties:
$s_i \models Ff$ iff $s_i \models f$ or $s_{i+1} \models Ff$
$s_i \models Gf$ iff $s_i \models f$ and $s_{i+1} \models Gf$
$s_i \models fUg$ iff $s_i \models g$ or ($s_i \models f$ and $s_{i+1} \models fUg$)
$s_i \models fRg$ iff ($s_i \models f \wedge g$) or ($s_i \models g \wedge s_{i+1} \models fRg$)
From an implementation point of view, an evaluation can be performed with the help of two arrays in the length of all subformulas and by sweeping through σ in a backward manner. A similar approach is presented by Havelund et al in [8] for runtime verification of systems.

Visualization. Figure 1 presents the visualization of the formula $f = G((q \wedge \neg r \wedge Fr) \rightarrow (\neg pUr))$ as discussed before. and as specified in the formula editor in Fig. 2. We can assign a color to each atomic proposition (leafs of the formula tree) and subformulas and use those colors to highlight in the MSC representation where a formula or subformula holds. As shown in Fig. 1, highlighting of formulas takes place at those processes whose state varibales are used in the atomic propositions or subformulas. In case of subformulas, all relevant processes must be determined with respect to every atomic proposition that is used in a subformula. By coloring each subformula in a different color, it is easy to identify when a subformula is valid across the trace. Note that in Fig. 2, we decided to use colors for only 4 nodes of the formula, namely a light blue color for the overall formula, a red color for the atomic proposition that shows a failure of a server, a green color for the atomic proposition that shows an operational, available server and a blue color for the atomic proposition that shows that all customers of class A are not present at the server.[2]

The modelchecking algorithm computes additional information for each node in the tree of a formula. That information includes the total number of states that fulfill a (sub)formula as well as the position of first and last occurrence of those states for ease of navigation. Buttons are provided that make the visualizer scroll to the first or last occurence. This supports the intended usage of modelchecking: the modeller can specify properties of interest and the modelchecker guides the modeller to that particular fragment of the trace.

4 Tool

Traviando supports the visualization and analysis of traces of interacting processes. The visualization is based on MSCs and its particular purpose is to make information accessible to a human being, information that is otherwise hidden in large trace file. MSCs are promising since they focus on the interaction of processes and we believe that this is crucial: in modeling as well as programming with interacting submodels, components, processes or threads, the behavior of an individual process is rather simple to analyze and debug but the overall effect of interactions contributes much to the complexity of such systems. So the latter requires adequate tool support. Traviando supports a number of operations that are helpful for trace analysis. In [11], we discuss the role and potential of fundamental operations on MSCs for trace analysis in more detail, for instance, the role of grouping a subset of processes into a single process to manage the level of abstraction and to set a focus on particular processes (while others are aggregated into one or few groups, "environments"). Traviando also supports moderate manipulations on the total order of events which is still consistent with the partial order of an MSCs and which can substantially improve the visualization and make it easier to understand for a human observer, see [11] for further details. A key advantage to a direct animation of some modeling formalism is that the MSC visualization in Traviando allow to track down causes in a backword manner while retaining an overview of a number of events. Most animations of dynamic behavior like the token game for Petri nets, some animated simulation model, or some debugging facility usually do only show the current state and a human user is forced to memorize its history while navigating forward.

Traviando imports sequences in an open XML format that consists of two parts, namely a prefix and the sequence of events that constitutes the trace. The prefix contains definitions for processes, events, their type and association with processes, state variables and more. The prefix helps to keep the sequence of events concise in its description and also allows for some preprocessing and consistency check for the trace based on the given structural information. The sequence of events in the second part of a trace can be enhanced in many ways by additional information, for instance by time stamps and by information on changes to state variables.

Traviando is implemented in Java and is able to work on traces of different kinds and from different sources. It operates on traces generated with the APNN toolbox [3]. Those traces result from Petri net models and a notion of process is introduced by partitioning the set of place (state variables). Many partitions are possible, the finest partition has one place per process, the coarsest has one process with all places. The APNN toolbox particularly supports superposed GSPNs that are composed of Petri nets by synchronized transitions; that class of nets comes with a partition of places and hence has a natural mapping to processes in MSCs. The resulting MSCs have synchronized events for transitions whose pre- or postset covers different processes.

Traviando operates on traces generated with the ProC/B toolset [2] which follows the common process interaction

[2]Colors are visible in the pdf file of this paper.

approach for simulation modeling. The ProC/B notation is based on a notion of hierarchy with function calls. So the resulting traces make use of send-receive events (represented as directed actions). This allows us to infer a notion of population and response times for function calls to (server) processes. In [10], we describe three additional visualization operations that are particularly designed to track down what happens in a trace of a ProC/B model and with respect to time. Highlighting and colorings are useful to illustrate more information on the empirical distribution of response times and populations derived from a trace. Visualizing the number of open function calls also helps to identify the root of blocking and deadlock situations in open models.

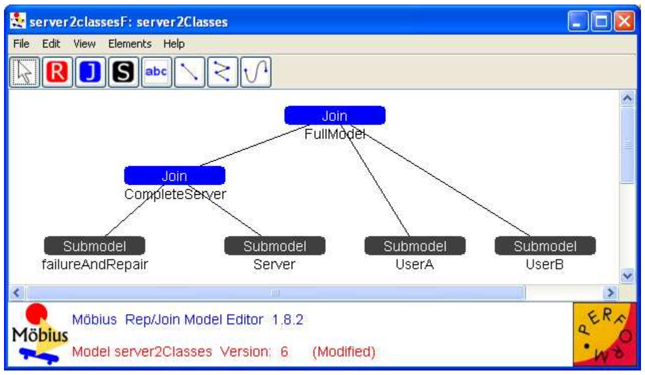
The most recent advance of Traviando is towards traces generated from the multi-formalism multi-solution framework Mobius [5]. Models are structured in a hierarchical manner such that a decomposition into processes according to the composition of atomic and composed models is a natural choice. Fig. 3 shows a Mobius model of a server with failure and repair for a performability study. We generated a trace from this model that serves as a running example for the visualization in Fig. 1 and and the formula in Fig. 2.

Traviando's new visualization features include a LTL modelchecking of traces that is supported by a pattern system. Atomic propositions are build by arithemic expressions on state variables and equalities and inequalities. As illustrated in Fig. 2, we combine three descriptions of a formula to make LTL modelchecking more accessible to a user. The upper right window shows the formula in a graphical, tree-type representation the lower left window describes the formula in technical prose, and the lower right window gives the common formal representation. Note that those representations describe the same information, but should complement each other for human understanding.

The graphical, tree-type representation with one node per logical operator and atomic propositions as leafs is used to derive formulas by refinement. Atomic propositions can be selected by mouse-clicks from a menu that is derived from a currently defined set atomic propositions. A set of atomic propositions is defined in an additional editor window. Colors are associated with nodes in the formula editor to define the coloring and highlighting used for the visualization of the modelchecking results on a MSC, for instance the colors selected in Fig. 2 match those in Fig. 1. Novel trace reduction techniques, which are not presented here for lack of space, help to separate repetitive from progressing parts of a trace and can substantially cut down on the number of events considered for trace analysis.

5 Conclusion

We propose LTL modelchecking for the analysis of traces in combination with a trace visualization by MSCs.



Möbius Rep/Join Model Editor 1.8.2
Model server2Classes Version: 6 (Modified)

Figure 3. Composed Model

The approach is implemented in Traviando, a new software tool for Trace visualization and analysis from Dortmund University. More information on Traviando can be found at http://www4.cs.uni-dortmund.de/Traviando/

Acknowledgements We would like to thank W.H. Sanders, Tod Courtney and Michael Mc Quinn for supporting us with an appropriate XML trace output of Mobius.

References

[1] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th Intl. Conf. on Concurrency Theory*, pages 114–129. Springer Verlag, 1999.

[2] F. Bause, H. Beilner, M. Fischer, P. Kemper, and M. Völker. The ProC/B toolset for the modelling and analysis of process chains. In T. Field, P.G. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation, Modelling Techniques and Tools*, Springer LNCS 2324, pages 51–70, 2002.

[3] Falko Bause, Peter Buchholz, and Peter Kemper. A toolbox for functional and quantitative analysis of DEDS. In Ramón Puigjaner, Nunzio N. Savino, and Bartomeu Serra, editors, *Computer Performance Evaluation (Tools)*, volume 1469 of *Lecture Notes in Computer Science*, pages 356–359. Springer, 1998.

[4] E. M. Clarke, Jr. O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[5] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Trans. Software Eng.*, 28(10):956–969, 2002.

[6] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark A. Ardis and Joanne M. Atlee, editors, *FMSP*, pages 7–15. ACM, 1998.

[7] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[8] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), LNCS 2280*, pages 342–356. Springer Verlag, 2002.

[9] ITU-T Recommendation Z.120. *Message Sequence Charts (MSC'96)*, 1996.

[10] P. Kemper and C.Tepper. Trace based analysis of process interaction models. In *Proc. of the 2005 Winter Simulation Conference*, pages 427–436, 2005.

[11] P. Kemper and C. Tepper. Visualizing the dynamic behavior of ProC/B models. In T. Schulze et al., editor, *Simulation und Visualisierung*, pages 63–74. SCS Publishing House, 2005.

[12] Peter Kemper and Carsten Tepper. Visualizing the dynamic behavior of ProC/B models. In Thomas Schulze, Graham Horton, Bernhard Preim, and Stefan Schlechtweg, editors, *SimVis*, pages 63–74. SCS Publishing House e.V., 2005.

[13] Sebastián Uchitel, Robert Chatley, Jeff Kramer, and Jeff Magee. Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 597–601. Springer, 2003.