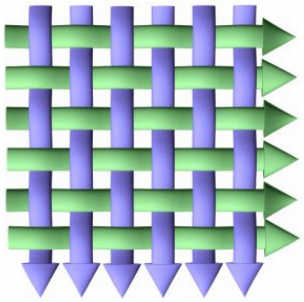
Sonderforschungsbereich 559
Modellierung großer Netze in der Logistik



Technical Report 06007

ISSN 1612-1376

Trace Analysis – Gain Insight through Modelchecking and Cycle Reduction

Teilprojekt M2:
Prof. Dr. Peter Kemper
Universität Dortmund
Informatik IV
August-Schmidt-Str. 12
44227 Dortmund

Teilprojekt M1/M2:
Carsten Tepper
Universität Dortmund
Informatik IV
August-Schmidt-Str. 12
44227 Dortmund

Dortmund, 23. Juni 2006

Trace Analysis - Gain Insight through Modelchecking and Cycle Reduction *

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 models are crucial steps to the success of a model-based development of systems. Traces are a common and powerful mean to document the dynamic behavior of a model and are generated from many sources, ranging from simulation engines to modelcheckers. In this paper, we describe techniques and corresponding tool support that helps a modeler to gain insight in the dynamic behavior of a complex model 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 develop a novel technique to reduce the sequence of events leading to such states. The proposed techniques are implemented in Traviando, a stand alone tool with an open XML interface to import traces from many modeling environments, e.g., the APNN toolbox and Mobius among others. A subtle modeling error in a multi-class server system serves as an application example to illustrate the effectiveness of our approach in revealing the cause of an observed effect.

Keywords: cycle reduction, 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.

Traces of interesting systems usually contain large volumes of data, so an automated analysis is important to extract relevant pieces of information. 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 [5] 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 [10] 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.

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. [7, 8] 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 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

*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.

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 message sequence charts (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 to this basic functionality, we propose a novel reduction method based on the identification and removal of cycles in a trace. This reduction is particularly useful in models of cyclic communicating processes, which are common in performability modeling where a finite number of cyclic workload processes are composed with cyclic failure and repair models for the availability of resources. 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 [6].

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. Section 4 proposes the new reduction technique and algorithms for the identification and selection of cycles in traces. We describe and compare three different greedy algorithms that select a set of non-intersecting cycles in a trace for subsequent reduction. In Section 5 we introduce our new tool for trace visualization, its architecture and main functionality. In Section 6, we show how to make good use of our tool to debug an example performability model of a server with two classes.

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 "=". For example, if $s \in S \subseteq \mathbb{N}$ is the state of an automaton, $=$ may be the usual equality among integer values, if $s \in S$ is a marking of a Petri net, then $=$ is the equality of markings (integer vectors), if $s \in S$ is the description of a term of a process algebra, then $=$ may be defined as a weak or strong bisimulation, and similarly for other formalisms with some notion of bisimulation for states like stochastic well-formed nets (SWNs), and the multi-paradigm models of Mobius. 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.

Let us define some common operations for sequences. The length of σ is defined as $len(s_0 e_1 s_1 \dots s_n) = n$. The concatenation $\circ$ of two sequences $\sigma = s_0 e_1 s_1 \dots s_n$ and $\sigma' = s'_0 e'_1 s'_1 \dots s'_m$ where $s_n = s'_0$ is defined as $\sigma \circ \sigma' = s_0 e_1 s_1 \dots s_n e'_1 s'_1 \dots s'_m$. Obviously, if $\sigma'' = \sigma \circ \sigma'$ then $len(\sigma'') = len(\sigma) + len(\sigma')$. For $\sigma = s_0 e_1 s_1 \dots s_n$ and $0 \le i < j \le n$, we define a projection or substring operation as $sub(\sigma, i, j) = s_i e_{i+1} \dots s_j$. If states of σ have some compositional structure, we may use that for a visualization as MSCs.

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), [12], 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 $i, j \in \{1, 2, \dots, n\}$.

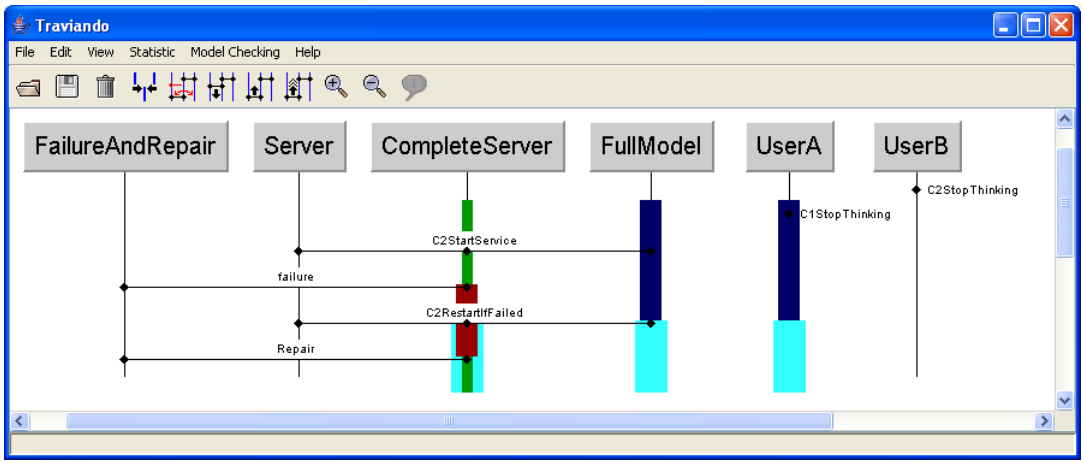


Figure 1. MSC of Example Trace

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 example is discussed in further detail in Section 6. The vertical order of events (top-down) follows the order of events in a visualized sequence σ.

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 [5], 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 $s_i \models g$.

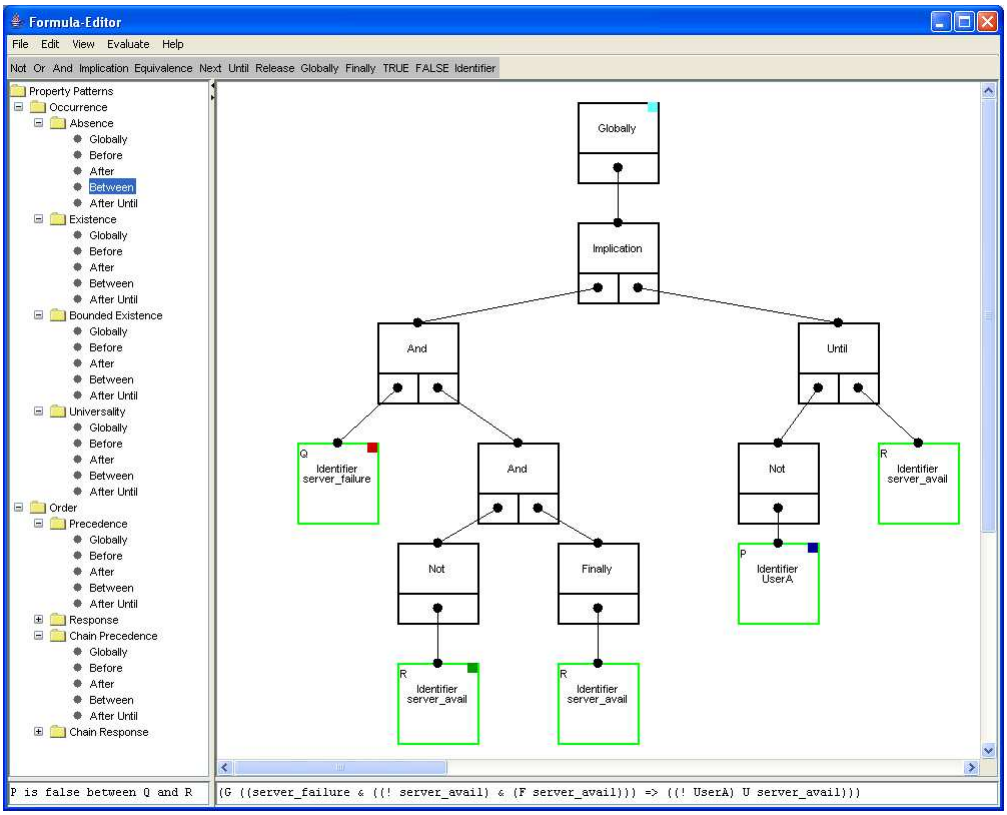


Figure 2. Graphical formula editor with pattern system

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. [8] 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 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 $f = G((q \wedge \neg r \wedge Fr) \rightarrow (\neg pUr))$ 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 [10] 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. In that scenario, usually the question arises how a particular state s_i could be reached. If s_i is not located in the beginning of the trace σ, one may want to have a shorter trace σ' that leads to s_i more directly. This is the goal of the reduction operation we introduce next.

4 Cycle Detection and Reduction

The operation that we introduce in this section is a reduction operation that removes repetition. For $\sigma = s_0 e_1 s_1 \dots e_n s_n$ with $0 \leq i < j \leq n$ and $s_i = s_j$, we define a reduction operation $red(\sigma, i, j) = sub(\sigma, 0, i) \circ sub(\sigma, j, n)$.

The reduction operation is based on the repetition of states which can be understood as a cycle or loop. For a given sequence σ, we can compute all cycles and subsequently remove step by step as many cycles as possible to obtain a sequence σ'. Sequence σ' is obviously shorter than σ and leads to s_n more directly. The potential of this operation to assist a user in his understanding of a trace is twofold, first to recognize cycles and what events give a cyclic behavior and second why and how a particular state or sequence of states like s_n can be reached. In this way, we distinguish between cyclic and progressing fragments of a trace.

From a modeling point of view, the usefulness of the reduction operation is based on a key assumption: the state of a system s_i must be sufficient to define the subsequent behavior that is present in the trace and starting at s_i. This assumption is usually fulfilled in untimed automata if $s \in S$ describes the state of an automaton completely and also in Markov models since the current state defines the potential future behavior in a Markov process. However, in discrete event simulation of non-Markovian models in general, a selection of state variables is usually not sufficient and rather the current event list – the state of the simulator – would be necessary to describe the state of the simulation, which would obviously minimize the possibility of cycles.

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

In the following, we investigate the algorithmic side of cycle reduction in three steps, namely detection of all cycles, selection of a subset of cycles for removal and removal of those cycles. It is clear that an on-the-fly approach would not necessarily consider all cycles in a trace. Nevertheless, before we describe solution algorithms, it is worthwhile to investigate the implications of each of the three steps for a better understanding.

Detection of Cycles. For a given sequence σ of length n, we want to identify a set $\mathcal{I}$ of all pairs $(i,j), 0 \leq i < j \leq n$ with $s_i = s_j$ in σ. $\mathcal{I}$ contains all cycles of σ. Entries of $\mathcal{I}$ can be also understood as an interval of integer values on the index range of states in σ. Set $\mathcal{I}$ could be the basis for a subsequent selection and removal of cycles. However, certain states may occur once, twice or many times in σ. Figure 3 gives the number of occurrences for states of a trace σ that we analyze in more detail in Section 6. States are ordered by decreasing number of occurences. For instance, the first state in the graph occurs 339 times. Note that this is unfortunate for an explicit enumeration of entries of $\mathcal{I}$. If a state occurs k times, that state alone generates $k \cdot (k-1)/2$ elements of $\mathcal{I}$. The calculation reflects the number of possible selections of 2 states among k states where permutations count only once. For $k = 339$, that state alone contributes 57291 intervals to $\mathcal{I}$. A more concise representation of $\mathcal{I}$ is a set $\mathcal{J}$ of tuples $(s, i_1, \ldots, i_k)$ where $s_{i_1} = s_{i_2} = \ldots s_{i_k}$ and k is the number of occurrences of s in σ. Clearly, the value of k depends on s. The generation of $\mathcal{J}$ for σ is straightforward. Starting from $\mathcal{J} = \emptyset$ the generation algorithm runs once through all states of σ and inserts each state s_i into $\mathcal{J}$. There are two cases possible for the insertion operation. If no tuple with $s = s_i$ exists in $\mathcal{J}$, a new tuple (s_i, i) is created. Otherwise an existing tuple $(s, i_1, \ldots, i_k)$ is extended to $(s, i_1, \ldots, i_{k+1})$ and i_{k+1} stores the index value of i. The complexity is $O(n\ log\ n)$ since n states are inserted and if we assume that insertion requires $O(log\ n)$. All elements with $k = 1$ (states that occur only once) are subsequently removed from $\mathcal{J}$. $\mathcal{J}$ allows for an efficient determination of a maximum interval (i_1, i_k) and subintervals that shall fit into given limits for the lower or upper index value.

Selection of Cycles. Based on a given set $\mathcal{I}$ (possibly implemented by set $\mathcal{J}$ that gives access to elements of $\mathcal{I}$), we consider the problem of selecting a subset of cycles $\mathcal{C} \subseteq \mathcal{I}$ such that cycles do not overlap and that a maximal number of states of S is covered. Formally, let $ol \subseteq \mathcal{I} \times \mathcal{I}$ be a relation that indicates which intervals overlap and two intervals (i_1, i_2) and (i_3, i_4) are in ol, if neither $i_2 \leq i_3$ nor $i_4 \leq i_1$ holds. The problem of finding a maximal set of cycles can be described as $max\ \sum_{i \in \mathcal{I}} g_i \cdot x_i$ where $g_i = i_2 - i_1$ for tuple $i = (i_1, i_2)$ and x_i is an indicator variable with values 0 and 1. The problem has side conditions, one for each element $(i,j) \in ol$, namely $x_i + x_j \leq 1$. We denote the problem of finding $\mathcal{C}$ as sequence reduction problem (SRP) for short. Due to the type of restrictions, the SRP is similar but not necessarily equivalent to the well-known NP-complete knapsack problem. In any case, we obtain difficulties in two ways. We have to solve a combinatorial optimization problem and that problem is defined on $\mathcal{I}$ whose cardinality is subject to a combinatorial explosion as well. In conclusion, we proceed in a pragmatic manner with heuristic and computationally inexpensive algorithms in the following. We propose three heuristic strategies.

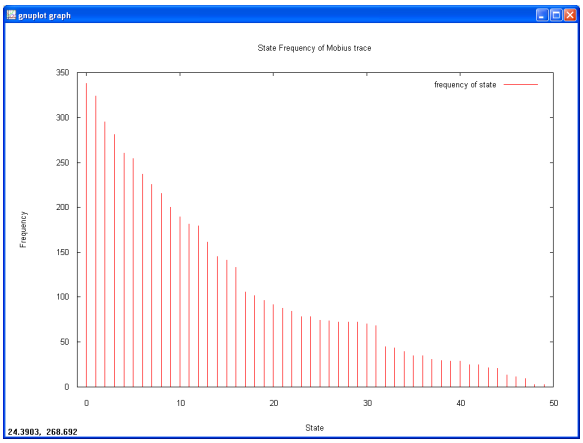


Figure 3. Number of occurrences of states

The first strategy is a gready strategy that selects largest elements first, i.e. $i = (i_1, i_2) \in \mathcal{I}$ with maximal value of $i_2 - i_1$. For a selected element i, the approach iterates on a reduced set $\mathcal{I}' = \mathcal{I} \backslash \{j | (i,j) \in ol\}$. Note that identification of large intervals can be implemented efficiently with $\mathcal{J}$. We denote this strategy as greedy by size (GS). It requires knowledge of all intervals, i.e., set $\mathcal{J}$ has to be computed.

The second strategy is a greedy strategy that selects cycles in the order of occurence, resp. identification. We denote this strategy as greedy by order (GO). It is an on-the-fly approach that runs through σ from the beginning, creates a set S' and removes cycles from σ (and corresponding states from S') as soon as a cycle is identified.

For the third strategy, we make use of a greedy algorithm for a related problem, the minimal weighted vertex cover (MWVC), which is NP-complete. We denote this strategy as greedy by vertex cover (GVC). The MWVC problem is defined as follows. Let $G(V, E)$ be an undirected graph with a finite set of nodes V and a finite set of edges E. Let $g : V \rightarrow \mathbb{N}$ be a weight function that assigns positive weights to nodes in V. The WMVC problem is to find a subset of nodes $V' \subseteq V$ such that $\sum_{v \in V'} g(v)$ is minimal and for all edges $(v,w) \in E$ either v or w (or both) are an element of V'. We define a σ-graph as $V = \mathcal{I}$, edges

$E = \{(i,j)|(i,j) \in ol\}$ and weights $g(i) = i_2 - i_1$ for $i = (i_1, i_2) \in \mathcal{I}$ for a sequence σ.

Theorem 1. *For any given σ-graph $G(V,E)$ with $V', V'' \subseteq V$ and $V'' = V \backslash V'$ holds that if V' is a solution for WMVC then V'' is a solution for SRP.*

Proof. Note that $g((i_1, i_2)) \geq 0$ by definition of $\mathcal{I}$. For any $v, w \in V''$, $(v,w) \notin E$ since all edges are covered by V' if V' is a valid solution of WMVC. Hence V'' must be a legal solution of SRP, i.e., no two intervals in V'' overlap.
It remains to show that it is maximal. We assume the contrary, let $v \in V'$ such that $V'' \cup \{v\}$ is a better solution to SRP due to the positive weight of v. If v has no edge adjacent to it, then $v \notin V'$ because it is not necessary for a solution of WMVC and hence $v \in V''$ holds already and the assumed better solution makes no difference. So v must have at least one edge and $v \in V'$. Since V' is a solution of WMVC each node must be exclusively covering at least one edge, so there must be an edge (v,x) with $x \notin V'$. However, that x must be in $V'' = V \backslash V'$ and thus $V'' \cup \{v\}$ cannot be a solution of SRP due to $(v,x) \in E$. Hence such x and also v can not exist. □
The theorem implies that we can use all(!) known algorithms for WMVC to solve our SRP. In this setting, we select only one and evaluate a simple greedy algorithm that iterates on two steps. It starts with a σ-graph $G = (V,E)$ and sucessively selects an edge (step 1) and removes it together with its adjacent nodes and edges (step 2). The algorith selects an edge $e = (i,j)$ (from the remaining edges) that has minimal weight $g(i) + g(j)$. The GVC algorithm is the weighted version of a classical greedy algorithm for WMVC [9]. We avoid the combinatorial explosion in the generation of $V = \mathcal{I}$ by makeing use of a subset $\mathcal{I}' \subseteq \mathcal{I}$ instead. We use $\mathcal{I}' = \{(i_j, i_{j+1})|(s, i_1, \ldots, i_k) \in \mathcal{J}, 1 \leq j < k\} \cup \{(i_1, i_k)|(s, i_1, \ldots, i_k) \in \mathcal{J}\}$. The subset contains sufficient elements of $\mathcal{I}$ to retain the correct solution (the first set of the union of two suffices); the second set of the union is used to make it simpler for GVC to avoid large intervals and leave them for the solution of the SRP.

Removal of Cycles. Removal of a set of cycles $\mathcal{C}$ from σ is straightforward. However, note that if $\mathcal{C}$ covers a large fraction of σ and has many elements, it can be simpler and more efficient to generate a new and reduced trace σ' from the few subsequences that retain instead of transforming σ by a sequence of removal operations.

Note that all three greedy algorithms GS, GO and GVC are based on heuristics, are computationally inexpensive, but follow rather different selection strategies. It requires some further experimental analysis if those algorithms work sufficiently well, if there are significant differences such that one may be identified as superior or if they are rather complementary and a tool may apply all and select the best result for a particular trace σ.

Evaluation. We exercised the greedy algorithms on a set of traces taken from a number of different examples. Table 1 gives the resulting values for $max \sum_{i \in \mathcal{C}} (i_2 - i_1)$ for the calculated set $\mathcal{C}$.

Row *Courier* refers to a trace of length $n = 1000$ that is generated from a stochastic Petri net model of the Courier protocol model by Li and Woodside [15]. The model is intended for performance analysis and generates a recurrent, finite Markov chain. The initial marking is chosen such that the model has a small state space and is expected to show a lot of cyclic behavior. We used the APNN toolbox [3] for modeling the Courier protocol and its simulation engine to generate the trace. The resulting values do not differ much and range between 949 for GS, 953 for GO and 949 for GVC. GVC operates on a graph with 859 nodes and 67445 edges and weights range from 12 to 948 with an average of 168.

Row *Server* refers to a Mobius model of a server with failure and repair that is considered in detail in Section 6. The model is intended for performability analysis and has a cyclic behavior since the workload of the server is given by a finite number of customers of two classes. However, the variant of the model we consider has a defect which implies that the trace can be partitioned into two parts where each can contain cycles but there is none that overlaps with both parts. All three algorithms show similar results.

Row *Prodcell* refers to a large model of a production cell by Heiner et al [11]. The model contains hundreds of places and transitions and considers control of a production cell which consists of a feeding belt, a rotating table, a robot with two arms, a press, a second belt to remove processed parts and a crane. The crane is used to obtain a closed-circuit that has a finite number of parts that are processed over and over again. The model has been imported in the APNN toolbox and analyzed [4]; we used the APNN simulator to generate a trace with 1000 events. GS, GO, and GVC give identical results. GVC operates on a graph with 91 vertices and 3406 edges, weights range from 162 to 810.

Row *Store* refers to model of a storage area described in [13]. It models the transfer of goods into a store and out of a store by trucks that allocate ramps and that are loaded or unloaded with the help for forklifts that are manned with workers. The model describes an open system and is developed with the ProC/B toolset [2]. The state representation that is chosen for the trace abstracts from certain details of the ProC/B simulation model. In particular, identities of entities are not revealed. The model has a defect in the sense that it reaches a situation, where the loading/unloading operations are all blocked due to a partial deadlock and the model shows only activites owing to the fact that it is an

open model and new entities can be generated. This model shows a significant advantage for GO.

Model	GS	GO	GVC
Courier	949	953	949
Server	5464	5466	5464
Prodcell	811	811	811
Store	417	506	420

Table 1. Results of Selection Algorithms

Since the differences among algorithms are usually small but we would like to gain more empirical data on their performance, so we currently apply all 3 algorithms in Traviando and then use the most effective for a trace reduction.

5 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 [14], 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 [14] 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 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 [13], 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.

In this paper, we focus on the most recent advance of Traviando towards traces generated from the multi-formalism multi-solution framework Mobius [6]. 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. Note that Mobius has the option to internally apply bisimulations to make states appear as equal for trace analysis which is to the benefit of the cycle reduction approach discussed in the previous section. In Section 6, we analyze a trace of a Mobius model whose structure is shown in Fig. 4.

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. First, a 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. Trace reduction techniques help to separate repetitive from progressing parts of a trace and can substantially cut down on the number of events considered for trace analysis.

The upper right window shows the formula 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.

6 Example

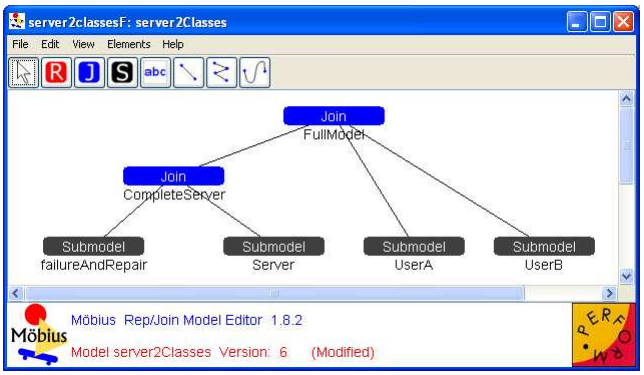


Figure 4. Composed Model

As an example application, we consider a performability model of a server that is subject to failure and repair. Failures happen mainly due to software failures and are handled by rebooting the system and restarting tasks that are performed. The workload can be partitioned into two types of tasks according to their service demands. The user population is limited, such that we decide to model the system by a closed model with two customer classes A and B and corresponding finite population $N(A)$ and $N(B)$. Class B is used to describe "normal" customers that give the baseline utilization and class A customers are particular ones with different service requirements. We adjust a thinktime between end of service and next service request for tasks of each class to match the interarrival times between request of the system under study. The goal of the study is to identify a threshold for $N(A)$ such that the server is not able to comply with certain quality of service (QoS) for that amount of customers. QoS is acceptable if 90% of all class A tasks are served within a given time limit. We do not provide further details of the timing because we will focus on debugging a corresponding model.

We make use of Mobius and its discrete event simulation engine to model this system and conduct a simulation study. Mobius offers several modeling formalisms. We choose the SAN formalism to model each aspect of our system individually and the composition operation that is based on shared variables to join the individual atomic models into an overall model. Figure 4 shows the compositional structure of the model. The *FullModel* combines a *CompleteServer* submodel of the server with submodels *UserA* and *UserB*, which describe the user behavior. Users interact with the server by putting tokens on input and output places of the server, those places are shared via the *FullModel*. The *CompleteServer* submodel encapsulates how a service is performed and a submodel *failureAndRepair* that models availability of the server. The number of customers of class A is a parameter that we evaluate over a range of values. In the following, we briefly describe the individual atomic models and their composition, namely the atomic model for failure and repair, the atomic model for the server, and the two atomic models customer classes A and B. The failure and repair model describes a cyclic behavior and switches between "available" and "failed" states to indicate the status of the server. It shares a boolean state variable "avail" with the server model. The server models a queue with random scheduling and no preemption for 2 customer classes. If a customer is served when a failure occurs, its service is interrupted and it is positioned back into the queue. Its service time is not memorized. Customers cycle between their own class-dependent submodel that delays them for a thinktime and the server. A simulation run reveals performance measures that in the long run deviate from what is expected. In particular the throughput of class B is slightly too small, for A slightly too high. At one point, we suspect that the model contains an error. Clearly, there is a variety of options for debugging simulation models. In this context, we focus on the use of trace analysis and the visualization of MSCs. We first check traces for $N(A) = 1$, $N(B) = 0$ (and vice versa) to see whether the one-class case works, but could not find any error. We select a minimal population $N(A) = N(B) = 1$ next to check for functional correctness for the case of two classes and make the server failure free. For that case, we cannot find any error. Finally, we generate a trace of some thousand events for the complete model including failures. We first check individual processes if all transitions in fact occur. For rare events like failures, we use a coloring mechanism to simplify their identification. We start with short traces and increase its

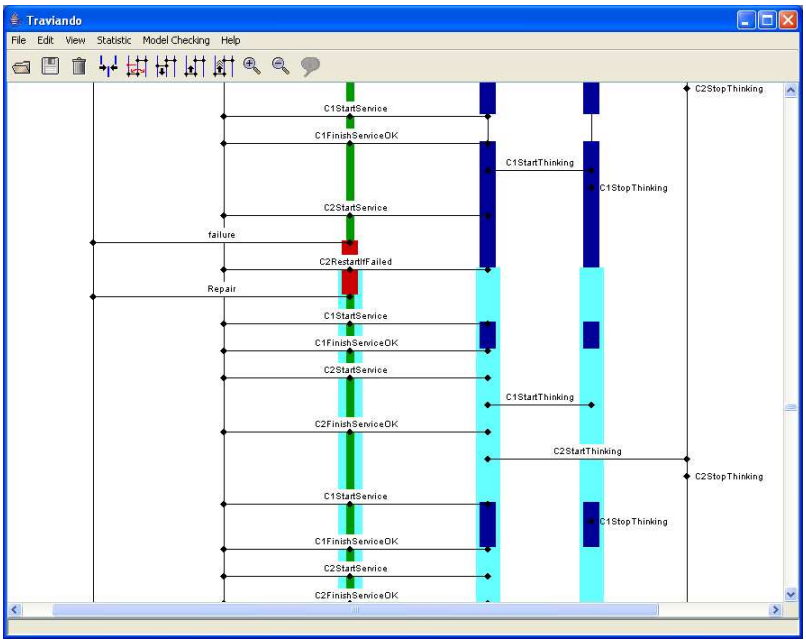


Figure 5. Results for LTL modelchecking ahead of cycle reduction

length if appropriate. A trace of 5475 elements happens to be long enough to contain all events and three failures with subsequent repair. We suspect that failures may be a cause of the problem and formulate different properties that should hold during a failure and repair cycle. We make use of LTL to formulate that if a server fails, all customers shall be either in the queue or located in their corresponding submodel and thinking. Since we want to be warned if that constraint is violated, we formulate the negated property as a formula of the pattern *absence* with scope *between* that is shown in Fig. 2; the formula has been discussed in Section 3 in the paragraph on patterns for illustration purposes. In particular, a light blue coloring indicates if the population of customers of class A outside the server is not matching $N(A)$, i.e. the class A population would be corrupted. Fig. 5 shows the relevant fragment of the trace to which the LTL modelchecking and coloring of the MSC traces guides us. We check the state information that is displayed in an additional window which is not shown here and recognize that the number of customers increases for that class. We use the cycle reduction on the prefix of the trace that leads to this situation (where the overall formula is true for the first time) and automatically obtain the trace represented in Fig. 1. By following the few events and checking the state variables, we recognize that the removal of a class B customer from service back into the queue in case of a server failure creates an additional class A customer in the queue. The error was introduced when class B was added to the model as a copy of class A. By knowing the error, one immediately sees simpler ways for its identification, for instance by defining invariants on the customer populations. Nevertheless, at first hand, any automated supported that is able to guide a user to critical fragments of a trace is a productivity enhancer in the debugging process. Finally, we want to explicitly note that the type of error we identified could happen in any modeling environment and that it is in no way related to the fact that we use Mobius as a modeling and analysis framework.

7 Conclusion

We propose LTL modelchecking for the analysis of traces in combination with a trace visualization by MSCs. In addition to that, we introduce a reduction operation based on the removal of cycles and algorithms for the detection and selection of cycles for removal. The overall approach is implemented in Traviando, a new software tool for Trace visualization and analysis from Dortmund University. We demonstrate the usefulness of the approach by detecting a subtle modeling error in a performability model of a server system with 2 classes. More information on Traviando can be found at http://www4.cs.uni-dortmund.de/Traviando/

Acknowledgements We would like to thank W.H. Sanders and his student 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] Peter Buchholz and Peter Kemper. On generating a hierarchy for GSPN analysis. *SIGMETRICS Performance Evaluation Review*, 26(2):5–14, 1998.

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

[6] 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.

[7] 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.

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

[9] M. R. Garey and David S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[10] 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.

[11] M. Heiner and P. Deussen. Petri net based design and analysis of reactive systems. In *Proc. 3rd Workshop on Discrte Event Systems (WoDES96)*, pages 308–313, 1996.

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

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

[14] 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.

[15] C. Murray Woodside and Yao Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *PNPM*, pages 64–73, 1991.

A Document Type Definition (DTD) of trace

```
<!ELEMENT Trace (Comment, Process+, Interactions*, Sequence+)>
<!ATTLIST Trace model CDATA #IMPLIED>
<!ATTLIST Trace generator CDATA #IMPLIED>

<!ELEMENT Comment (#PCDATA)>

<!-- Process section -->
<!ELEMENT Process (Action*, Var*)>
<!ATTLIST Process id CDATA #REQUIRED>
<!ATTLIST Process name CDATA #REQUIRED>

<!ELEMENT Action EMPTY>
<!ATTLIST Action id CDATA #REQUIRED>
<!ATTLIST Action name CDATA #REQUIRED>

<!ELEMENT Var EMPTY>
<!ATTLIST Var id CDATA #REQUIRED>
<!ATTLIST Var name CDATA #REQUIRED>

<!-- Interactions section -->
<!ELEMENT Interactions (Diraction*, Undiraction*)>

<!ELEMENT Undiraction (Touch+)>
<!ATTLIST Undiraction id CDATA #REQUIRED>
<!ATTLIST Undiraction name CDATA #REQUIRED>

<!ELEMENT Touch (#PCDATA)>

<!ELEMENT Diraction (From, To)>
<!ATTLIST Diraction id CDATA #REQUIRED>
<!ATTLIST Diraction name CDATA #REQUIRED>

<!ELEMENT From (#PCDATA)>

<!ELEMENT To (#PCDATA)>

<!-- Sequence section -->
<!ELEMENT Sequence (S,A*)>
<!ATTLIST Sequence type CDATA #IMPLIED>

<!ELEMENT S (V+)>
```

```
<!ELEMENT A (V*)>
<!ATTLIST A id CDATA #REQUIRED>
<!ATTLIST A time CDATA #IMPLIED>
<!ATTLIST A inst CDATA #REQUIRED>

<!ELEMENT V EMPTY>
<!ATTLIST V id CDATA #REQUIRED>
<!ATTLIST V val CDATA #REQUIRED>
```

B Superposed Generalized Stochastic Petri net model

The first example is a model of the APNN-Toolbox. The APNN-Toolbox allows the modelling of Generalized Stochastic Petri nets (GSPNs). It is possible to specify partitions on the places, so that the model is divided into partitions.

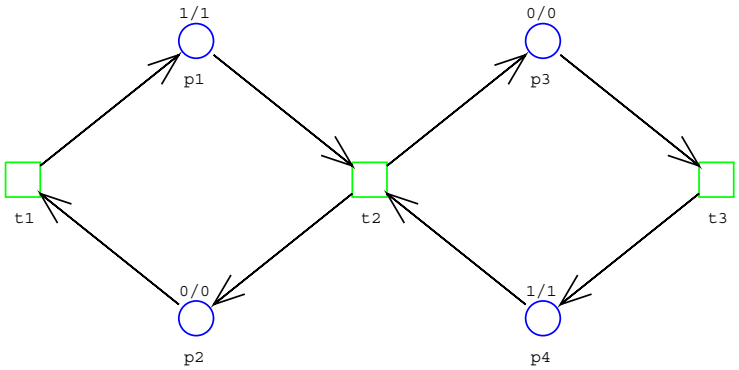


Figure 1: SGSPN model

The example in Fig. 1 contains two partitions $P^1 = \{p_1, p_2\}$ and $P^2 = \{p_3, p_4\}$. Partition P^1 contains the two variables (Var) p1 $\rightarrow$ v_1, p2 $\rightarrow$ v_2 and Partition P^2 p3 $\rightarrow$ v_3, p4 $\rightarrow$ v_4. Both partitions contain one local activity (Action) t1, respectively t3. The transition t2 is a synchronized transition which regards both partitions (Interactions). The interaction is undirected (Undiraction) because it is not clear which partition is the sender or receiver.
The example trace contains first a specification block which is not conform to the XML standard. Due to the size of the trace we have decided to specifiy first the structural part of the model followed by the dynamical part (Sequence). The structural part contains the definition of processes (Process) and interactions (Interactions) between the processes. The dynamical part is sequence (Sequence) of events. Every event (A) has a time stamp which is specified as global time and a list of variables which values have changed. The order of the list of variables is arbitrary.
Example trace for model in Fig. 1:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Trace file of model SGSPN -->
<Trace model="SGSPN" generator="APNNsim" >
<Comment> Purpose of this trace: demonstration </Comment>
```

```
<!-- declaration of processes -->
<Process id="0" name="P1" >
<Action id="a_1" name="t1" />
<Var id="v_1" name="p1" />
<Var id="v_2" name="p2" />
</Process>
<Process id="1" name="P2" >
<Action id="a_3" name="t3" />
<Var id="v_3" name="p3" />
<Var id="v_4" name="p4" />
</Process>
<Interactions>
<Undiraction id="a_2" name="t2">
<Touch>0</Touch><Touch>1</Touch></Undiraction>
</Interactions>
<!-- process sequence -->
<Sequence type="StateActionType">
<S>
<V id="v_1" val="1" />
<V id="v_2" val="0" />
<V id="v_3" val="0" />
<V id="v_4" val="1" />
</S>
<A id="a_2" t="0.5">
  <V id="v_1" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="1" />
  <V id="v_4" val="0" />
</A>
<A id="a_1" t="1.3">
  <V id="v_2" val="0" />
  <V id="v_1" val="1" />
</A>
<A id="a_3" t="2.4">
  <V id="v_3" val="0" />
  <V id="v_4" val="1" />
</A>
<A id="a_2" t="2.5">
  <V id="v_1" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="1" />
  <V id="v_4" val="0" />
</A>
<A id="a_3" t="3.3">
```

```
  <V id="v_3" val="0" />
  <V id="v_4" val="1" />
</A>
<A id="a_1" t="5.14">
  <V id="v_2" val="0" />
  <V id="v_1" val="1" />
</A>
<A id="a_2" t="6.55">
  <V id="v_1" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="1" />
  <V id="v_4" val="0" />
</A>
</Sequence>
</Trace>
```

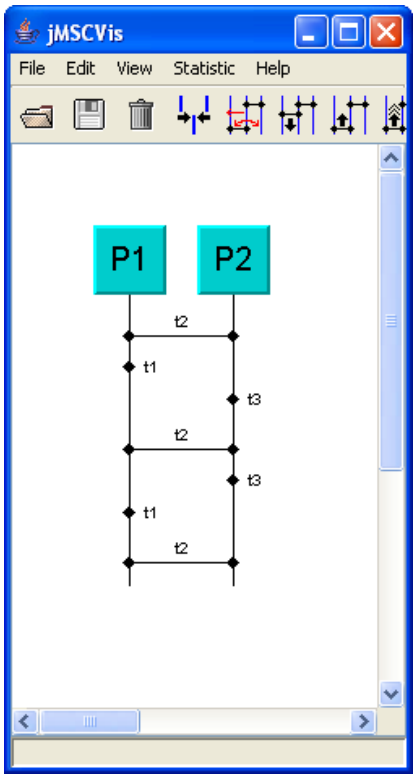


Figure 2: MSC of example trace

C ProC/B model

A modelling language which is especially designed to the needs of logistic networks is the ProC/B formalism, which is accompanied by a corresponding software toolset including a graphical interface for specification and a set of analysis tools.
The main structuring elements of ProC/B models are *Functional Units* (FUs), which encapsulate one or more *Chains*. A chain can be viewed as a structured and measured set of activities starting with *Sources* for process creations (demand), denoted by a circle with midpoint $\odot$, followed by a chronological sequence of *Process Chain Elements* (PCEs) describing activities, denoted by arrow-like hexagons, and the chain is accomplished by a sink, denoted by $\otimes$. Horizontal connections between PCEs indicate a sequential behavioural pattern of processes. Branches into and merges from alternative sub-chains are allowed and represented by vertical b ars. *Process Chains* can be hierarchically structured. PCEs may invoke sub-chains from so-called *subordinated FUs*. Subordinated FUs and sub-chains are described in the same manner as their super-ordinated FUs and chains (self-similarity). Actually, two types of subordinated FUs are distinguished: user-defined FUs and simple, predefined FUs of type *Server* or *Counter*. Servers are used to model active, possibly shared resources, i.e. machines, assembly lines, workers. In principle, servers correspond to single stations in queueing networks. Counters are used to describe passive resources, i.e. stores and waiting areas of usually restricted capacity.

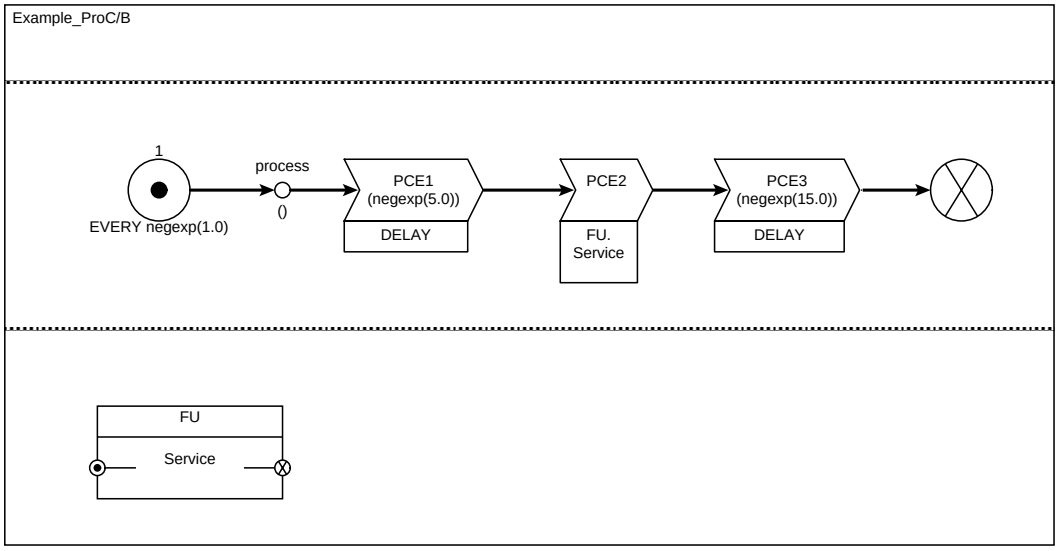


Figure 3: ProC/B model

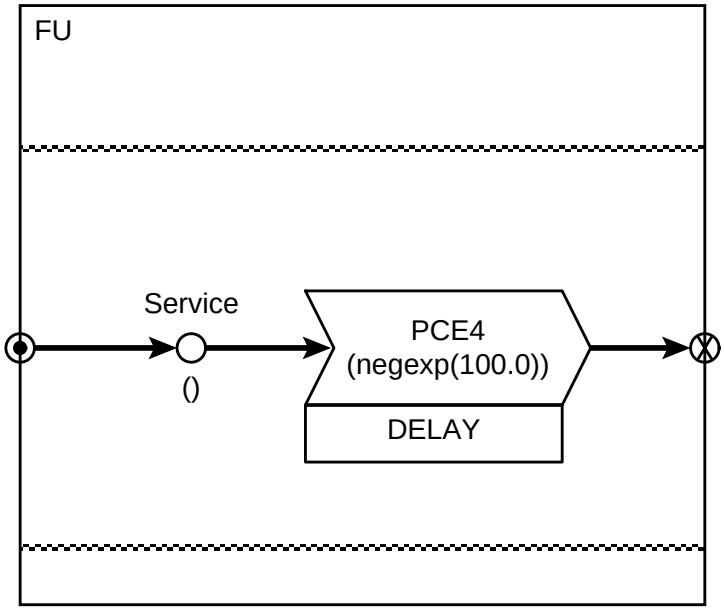


Figure 4: FU of ProC/B model in Fig. 3

In this example trace the sender and receiver can be dedicated. Therefore the interactions between the processes are directed (Diraction). Variables are the PCEs PCE1 → v_11, PCE3 → v_19 and PCE4 → v_28. Theses variables count the number of process instances which are waiting for proceed at these PCEs. The last differnce to the example trace of the SGSPN model is that the actions in the squence have the optinal attribute *i*. This attribute is necessary for identifing the process instance which has generated the event. The encoding of this attribute is the id of the process instance followed by a dot plus the id of the PCE where the process instance has waited before proceeding.

Example trace for model in Fig. 3:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Trace file of model Example_ProC/B -->
<Trace model="Example_ProC/B" generator="ProcessVis" >
<Comment> Purpose of this trace: demonstration </Comment>
<!-- declaration of processes -->
<Process id="0" name="process" >
<Action id="a_5" name="Source_process" />
<Action id="a_10" name="PCE1" />
<Action id="a_18" name="PCE3" />
<Action id="a_21" name="Sink_process" />
<Var id="v_11" name="PCE1" />
<Var id="v_23" name="PCE2" />
```

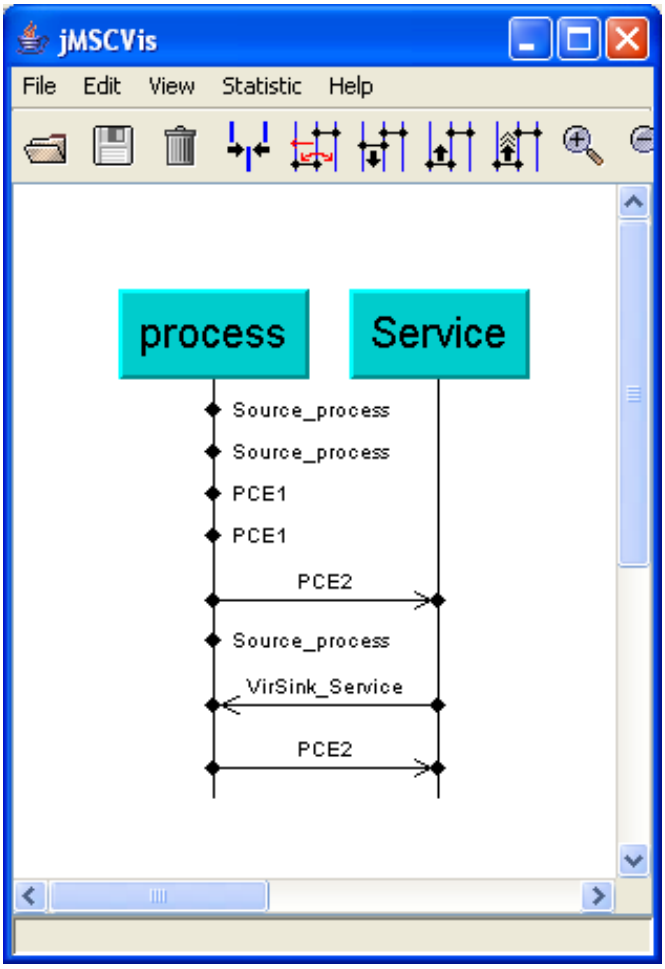


Figure 5: MSC of example trace

```
<Var id="v_19" name="PCE3" />
</Process>
<Process id="1" name="Service" >
<Action id="a_25" name="PCE4" />
<Action id="a_30" name="Sink_Service" />
<Var id="v_28" name="PCE4" />
</Process>
<Interactions>
<Diraction id="a_15" name="PCE2">
<From>0</From><To>1</To></Diraction>
<Diraction id="a_16" name="VirSink_Service">
<From>1</From><To>0</To></Diraction>
</Interactions>
```

```
<!-- process sequence -->
<Sequence type="StateActionType">
<S>
<V id="v_11" val="0" />
<V id="v_19" val="0" />
<V id="v_28" val="0" />
</S>
<A id="a_5" t="0.5" i="process1.5">
  <V id="v_11" val="1" />
</A>
<A id="a_5" t="1.3" i="process2.5">
  <V id="v_11" val="2" />
</A>
<A id="a_10" t="5.5" i="process1.18">
  <V id="v_11" val="1" />
  <V id="v_23" val="1" />
</A>
<A id="a_10" t="6.3" i="process2.18">
  <V id="v_11" val="0" />
  <V id="v_23" val="2" />
</A>
<A id="a_15" t="6.7" i="process1.25">
  <V id="v_23" val="1" />
  <V id="v_28" val="1" />
</A>
<A id="a_5" t="7.14" i="process3.5">
  <V id="v_11" val="1" />
</A>
<A id="a_16" t="7.88" i="process1.21">
  <V id="v_28" val="0" />
</A>
<A id="a_15" t="9.34" i="process2.25">
  <V id="v_23" val="0"/>
  <V id="v_28" val="1"/>
</A>
</Sequence>
</Trace>
```

D Möbius: Join model

The 3 atomic models in Fig. 6 has two shared variables/places p2 and p3. The place p1 contains one intinial token.

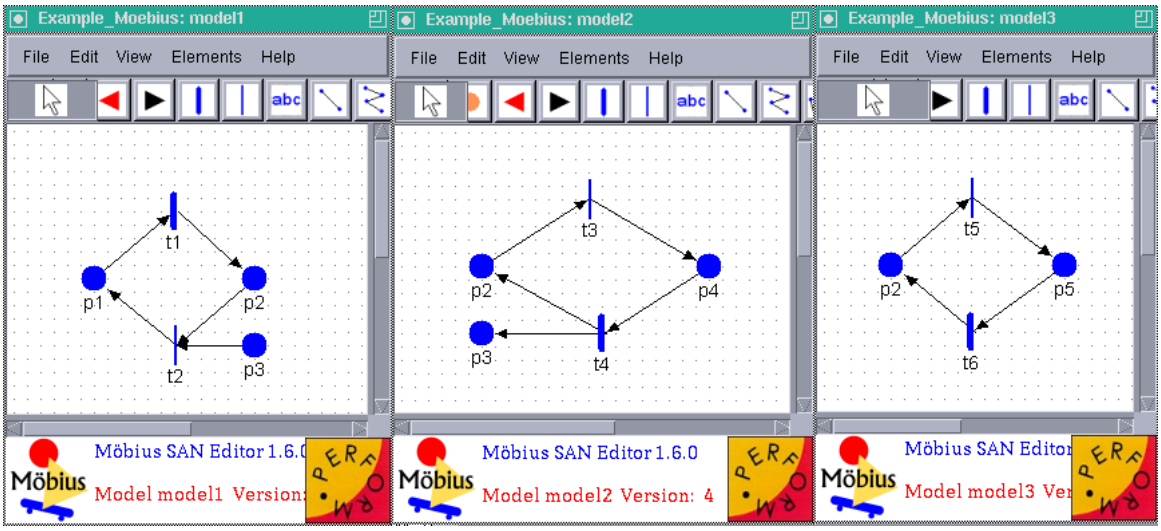


Figure 6: 3 atomic models

In Fig. 7 is depicted the hierarchy of the composed model. In this example, every node contains one variable. It is possible that a node has more than one or zero variables. If a node has zero variables then this node must not described as a process because the node cannot have any interactions with other processes.

Join1	p2
Join2	p3
model1	p1
model2	p4
model3	p5

Example trace for model in Fig. 7:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Trace file of model Example_Moebius -->
<Trace model="Example_Moebius" generator="manually_generated" >
<Comment> Purpose of this trace: demonstration </Comment>
<!-- declaration of processes -->
<Process id="0" name="Join1" >
<Var id="v_2" name="p2" />
</Process>
<Process id="1" name="Join2" >
```

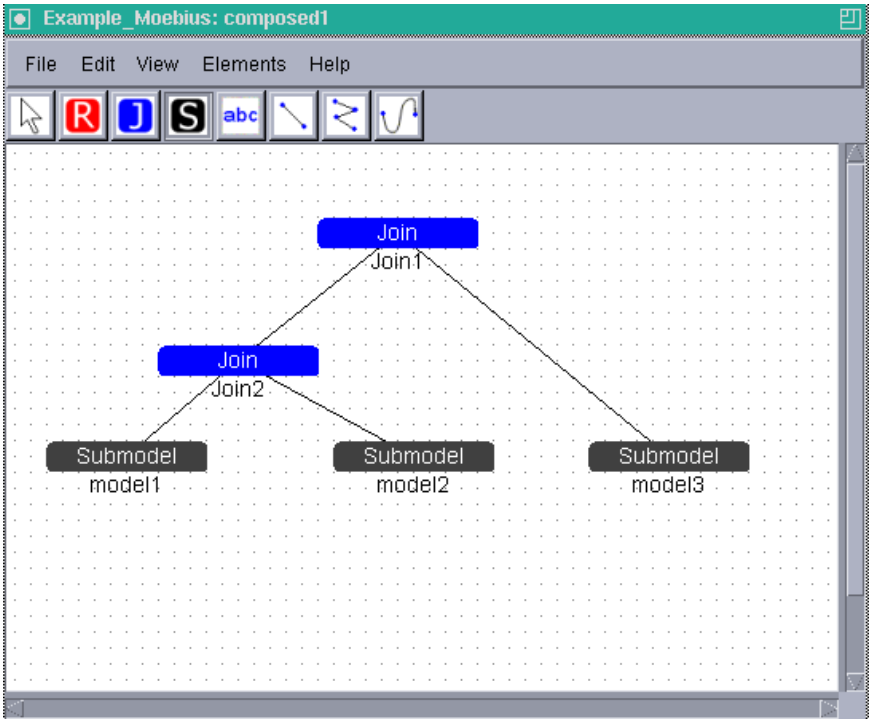


Figure 7: Composed model

```
<Var id="v_3" name="p3" />
</Process>
<Process id="2" name="model1" >
<Action id="a_1" name="t1" />
<Var id="v_1" name="p1" />
</Process>
<Process id="3" name="model2" >
<Action id="a_4" name="t4" />
<Var id="v_4" name="p4" />
</Process>
<Process id="4" name="model3" >
<Action id="a_6" name="t6" />
<Var id="v_5" name="p5" />
</Process>
<Interactions>
<Undiraction id="a_2" name="t2">
<Touch>0</Touch><Touch>1</Touch><Touch>2</Touch></Undiraction>
<Undiraction id="a_3" name="t3">
<Touch>0</Touch><Touch>3</Touch></Undiraction>
<Undiraction id="a_5" name="t5">
```

```
<Touch>0</Touch><Touch>4</Touch></Undiraction>
</Interactions>
<!-- process sequence -->
<Sequence type="StateActionType">
<S>
<V id="v_1" val="1" />
<V id="v_2" val="0" />
<V id="v_3" val="0" />
<V id="v_4" val="0" />
<V id="v_5" val="0" />
</S>
<A id="a_1" t="0.0159">
  <V id="v_1" val="0" />
  <V id="v_2" val="1" />
</A>
<A id="a_3" t="0.0159">
  <V id="v_2" val="0" />
  <V id="v_4" val="1" />
</A>
<A id="a_4" t="0.044">
  <V id="v_4" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="1" />
</A>
<A id="a_5" t="0.044">
  <V id="v_2" val="0" />
  <V id="v_5" val="1" />
</A>
<A id="a_6" t="0.6367">
  <V id="v_5" val="0" />
  <V id="v_2" val="1" />
</A>
<A id="a_3" t="0.6367">
  <V id="v_2" val="0" />
  <V id="v_4" val="1" />
</A>
<A id="a_4" t="1.0676">
  <V id="v_4" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="2" />
</A>
<A id="a_3" t="1.0676">
  <V id="v_2" val="0" />
  <V id="v_4" val="1" />
```

```
</A>
<A id="a_4" t="2.53">
  <V id="v_4" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="3" />
</A>
<A id="a_3" t="2.53">
  <V id="v_2" val="0" />
  <V id="v_4" val="1" />
</A>
<A id="a_4" t="3.28">
  <V id="v_4" val="0" />
  <V id="v_2" val="1" />
  <V id="v_3" val="3" />
</A>
<A id="a_5" t="3.28">
  <V id="v_2" val="0" />
  <V id="v_5" val="1" />
</A>
<A id="a_6" t="3.35">
  <V id="v_5" val="0" />
  <V id="v_2" val="1" />
</A>
<A id="a_2" t="3.35">
  <V id="v_2" val="0" />
  <V id="v_3" val="2" />
  <V id="v_1" val="1" />
</A>
<A id="a_1" t="4.24">
  <V id="v_1" val="0" />
  <V id="v_2" val="1" />
</A>
</Sequence>
</Trace>
```

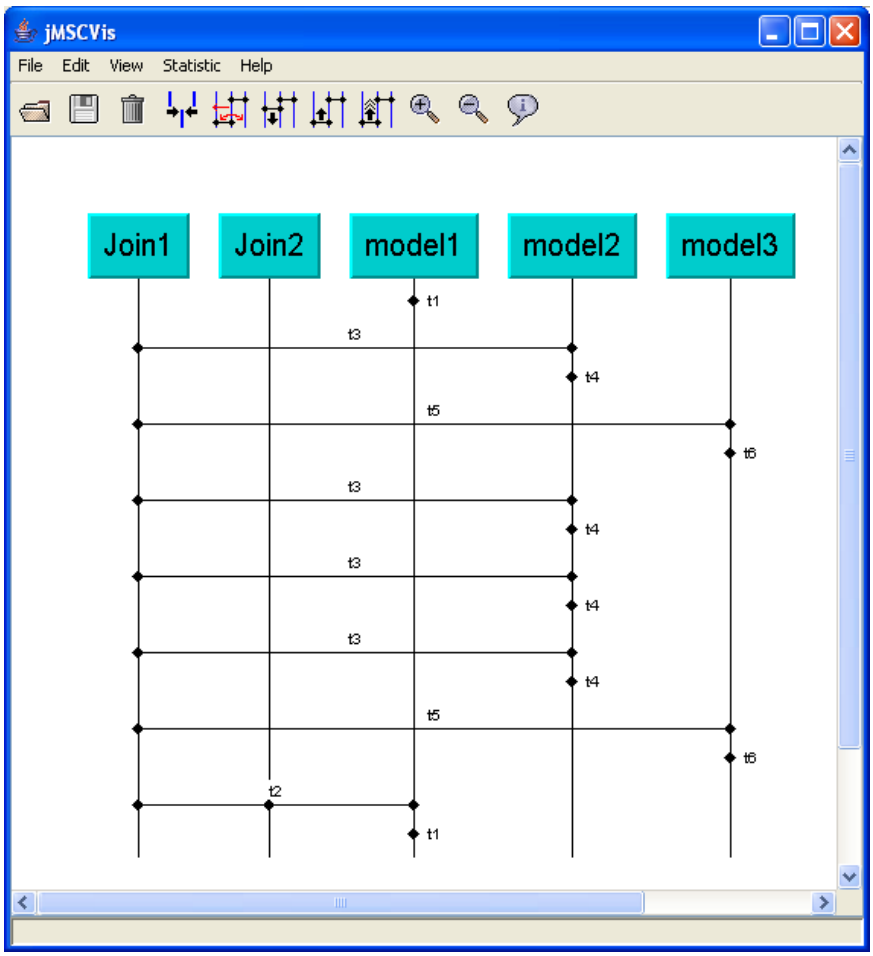


Figure 8: MSC of example trace