A partial order approach to branching time logic model checking

Citation for published version (APA):

Gerth, R. T., Kuiper, R., Peled, D., & Penczek, W. (1994). *A partial order approach to branching time logic model checking*. (Computing science reports; Vol. 9453). Technische Universiteit Eindhoven.

Document status and date:

Published: 01/01/1994

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Eindhoven University of Technology
Department of Mathematics and Computing Science

A Partial Order Approach to
Branching Time Logic Model Checking

by

R. Gerth, R. Kuiper, D. Peled and W. Penczek

94/53

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 94/53
Eindhoven, December 1994

A Partial Order Approach to Branching Time Logic Model Checking

Rob Gerth* and Ruurd Kuiper
Eindhoven University of Technology
Eindhoven, The Netherlands

Doron Peled
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA

Wojciech Penczek†
Eindhoven University of Technology‡
Eindhoven, The Netherlands

December, 1994

Abstract

Partial order techniques enable reducing the size of the state graph used for model checking, thus alleviating the 'state space explosion' problem. These reductions are based on selecting a subset of the enabled operations from each program state. So far, these methods have been studied, implemented and demonstrated for assertional languages that model the executions of a program as computation sequences, in particular the logic LTL (linear temporal logic). The present paper shows, for the first time, how this approach can be applied to languages that model the behavior of a program as a tree. We study here partial order reductions for branching temporal logics, e.g., the logics CTL and CTL* (all logics with the next-time operator removed) and process algebras such as CCS. Conditions on the subset of successors from each node to guarantee reduction that preserves CTL properties are given. Provided experimental results show that the reduction is substantial.

1 Introduction

Partial order (or more accurately, commutativity-based) methods are useful for tackling the exponential blowup in the memory required for the automated verification by model-checking of concurrent programs. They exploit the fact that many properties are insensitive to the order in which concurrent actions are executed. Fixing one out of many such orders can then be used to reduce the memory and time needed to check such properties. Such methods were studied so far [5, 10, 20, 23, 24] in conjunction with specifications that assert about the set of interleaved executions of the program; e.g., that use linear temporal logic without the next-state operator (**LTL-X**).

State-based algorithms for model checking a system are patterned after a depth-first search of the system's configurations or states, thus generating a state graph that allows checking whether a concurrent finite state program P satisfies a temporal logic property φ . Partial order reductions are aimed at constructing a reduced state graph, based on exploring for each visited state only a *subset* of the enabled operations, so that only some of the successors of that state are expanded and, hence, specifications can be verified in less space and time. The correctness of the reduced state graph generation algorithm is based on employing a set of constraints that limit the choice of such subsets of operations to those that guarantee that the evaluation of specifications is preserved.

*Partially supported by ESPRIT project P6021: "Building Correct Reactive Systems (REACT)".

†Partially supported by De stichting informatica-onderzoek in Nederland (SION).

‡On leave from Institute of Computer Science, PAS, Warsaw, Poland.

The next step is to try to extend these methods to handle other types of specifications. Natural candidates are specification languages based on branching models, in particular, branching time temporal logics. Such logics, as opposed to **LTL-X**, can distinguish the state where a nondeterministic choice is made in the execution of the program. We are guided by three main reasons for our pursuit of a reduction that preserves branching-time logics. The first one is achieving greater expressiveness, e.g., by using a logic such as **CTL*-X**, which, besides being able to distinguish the nondeterministic choices, can express all **LTL-X**-properties. The second one is the existence of some interesting restricted versions of branching time logics such as **CTL-X**. Although **CTL-X** does not include **LTL-X** (and vice versa), it can, by virtue of the branching operators, describe many interesting properties of programs. Moreover, due to its restrictions, it has a model-checking algorithm that is linear in the size of the checked formula [2], as opposed to the exponential algorithm for **LTL-X** [14]. The third motivation for such a reduction lies within the fact that branching temporal properties are preserved by bisimulation [1]; besides basing our correctness proof on this fact, checking that two states are bisimulation equivalent is itself important for process-algebra style correctness. Thus, our reductions can be used to improve the time required and the size of the state graph and can be used in conjunction with process-algebra based tools such as PSF and AUTO [15, 19].

The paper starts out investigating the proper constraints on the subset that is chosen to be explored at each visited state. Not unexpectedly, the set of constraints turns out to be strictly stronger than the one needed for **LTL-X**. Indeed, **CTL*-X** is more expressive than **LTL-X** is, so that branching points due to nondeterministic choices should be preserved in the reduced graph. Of course, this also means that reduction for **LTL-X** can produce smaller state graphs, and thus be more efficient in space and time. This is compensated by the fact that some branching time logics such as **CTL-X** have model checking algorithms that are linear rather than exponential in the size of the checked property.

The proof of the correctness of our algorithm is novel in that it is rather different from the one used for **LTL-X** reductions [20]: instead of using traces [16], i.e., equivalence classes of sequences, we show stuttering equivalence between the full and the reduced state graph [1]. This equivalence was proved in [1] to be a necessary and sufficient condition for ensuring that the two stuttering equivalent structures satisfy the same **CTL*-X** formulas.

CTL*-X is the most expressive of the logics we discuss and, consequently, the same result holds for logics that are included in it, namely **ACTL*-X**, **ACTL-X**, and **CTL-X**.

Experimental results show that even with the additional constraint on selecting subsets of the enabled operations, the reduction is still substantial. We demonstrate the reduction on various algorithms and protocols and compare it to the reduction obtained for **LTL-X**. The simplicity of the reduction algorithm, and the small overhead in time and memory it incurs, suggests that one can obtain significant improvement for state-based model checking, by using the suggested reduction algorithm, with a relatively small investment. We also investigate using our algorithm as part of a branching bisimulation checker. Experiments indicate that it is more efficient to use our reduction strategy to generate a state graph to be checked than it is to generate and check the full state graph.

2 Basic Notions

Syntax of **CTL*-X**

Let PV be a finite set of propositions. The set of state formulas and the set of path formulas are defined inductively:

- S1. every member of PV is a state formula.
- S2. if φ and ψ are state formulas, then so are $\neg\varphi$ and $\varphi \wedge \psi$.

S3. if φ is a path formula, then $A\varphi$ is a state formula.

P1. any state formula φ is also a path formula.

P2. if φ, ψ are path formulas, then so are $\varphi \wedge \psi$ and $\neg\varphi$.

P3. if φ, ψ are path formulas, then so is $U(\varphi, \psi)$.

The modal operator A has the intuitive meaning: “for all paths”. U denotes the standard Until.

CTL*-X consists of the set of all state formulae.

The following abbreviations will be used:

$$E\varphi \stackrel{def}{=} \neg A\neg\varphi, \quad F\varphi \stackrel{def}{=} U(true, \varphi), \quad G\varphi \stackrel{def}{=} \neg F\neg\varphi.$$

Sublogics of CTL*-X

CTL-X. The state modalities E and A and the path modalities U , F and G may only appear paired, i.e., in the combinations EU , EF , EG , AU , AF and AG .

ACTL*-X. The modality E is prohibited, and negation can be applied only to subformulas that do not contain modalities.

ACTL-X. The sublogic of **CTL-X** in which the modality E is prohibited, and negation can be applied only to subformulas that do not contain modalities.

LTL-X. Restriction to formulas of the form $A\varphi$, where φ does not contain A and E . We usually write φ instead of $A\varphi$ if confusion is unlikely.

Semantics of CTL*-X

Let T be a set of labels. A *model* for **CTL*-X** is a pair (F, V) , where $F = (W, (\xrightarrow{a})_{a \in T}, w^0)$ is a directed, rooted, edge-labeled graph with node set W and initial node w^0 , while V is a valuation function that assigns to each node w the set of propositions $V(w) \subseteq PV$ that are ‘true’ in w . The edge relation is assumed to be total: i.e. $\forall u \exists v. a \quad u \xrightarrow{a} v$. The labels on the edges in the definition of the graph are only used in the sequel for the benefit of the description of the suggested algorithm, but are ignored by the interpretation of the temporal logics.

Let $M = ((W, (\xrightarrow{a})_{a \in T}, w^0), V)$ be such a model and let $\pi = (v_0, v_1, \dots)$ be an infinite path starting at $v_0 \in W$ such that $v_i \xrightarrow{a_i} v_{i+1}$ for every i . Let π_i denote the suffix $(v_i, v_{i+1}, \dots)$ of π . Satisfaction of a formula φ in a state v of M , (written $M, v \models \varphi$ or just $v \models \varphi$), is defined inductively as follows:

S1. $v \models q$ iff $q \in V(v)$, for $q \in PV$.

S2. $v \models \neg\varphi$ iff not $v \models \varphi$. $v \models \varphi \wedge \psi$ iff $v \models \varphi$ and $v \models \psi$.

S3. $v \models A\varphi$ iff $\pi \models \varphi$ for every path π starting at v .

P1. $\pi \models \varphi$ iff $v_0 \models \varphi$ for any state formula φ .

P2. $\pi \models \neg\varphi$ iff not $\pi \models \varphi$. $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$.

P3. $\pi \models U(\varphi, \psi)$ iff there is an $i \geq 0$ such that $\pi_i \models \psi$ and $\pi_j \models \varphi$ for all $0 \leq j < i$.

Observe that satisfaction over our models coincides with the standard definition as provided in, e.g., [1].

Programs, State Graphs and Independence

For purposes of state graph generation and model checking, the specific syntactic structure of programs is not important. Instead, a *finite-state program* P is viewed as a 4-tuple (Q, T, q_0, I) where Q is a *finite* set of *states* giving, e.g., the values of the variables and the contents of the message queues, T is a finite set of operations such as assignments and send and receive actions to and from the message queues, $q_0 \in Q$ is the *initial state*, and I is a so-called *independence* relation on the program's actions that will be discussed below (see Definition 2.1). The enabling condition $en(q) \subseteq T$ is the set of operations that can be executed from a state q . Each operation $a \in T$ is identified with a partial function $a: Q \rightarrow Q$ (its denotation) that needs to be defined at least for each q such that $a \in en(q)$. We assume that $en(q) \neq \emptyset$ for any q .

Ignoring the independence relation for a moment, programs can directly be represented as state graphs, namely by taking Q as nodes and T for edges. The aim for reduction prompts to also consider subgraphs; a homomorphism is used in making the connection to the program.

A *state graph*, G_P , for a program P is a directed, rooted and edge labeled graph $(S, (\xrightarrow{a})_{a \in T}, s^0)$, with nodes S and initial node s^0 for which there is a homomorphism $st: S \rightarrow Q$ that maps nodes to program states such that (1) $st(s^0) = q_0$ and (2) if $s \xrightarrow{a} t$ then $a \in en(st(s))$ and $st(t) = a(st(s))$. Hence, a state graph explicitly represents computations of the program as paths through the graph. In fact, because actions are functional (i.e., their denotations are), computations, i.e., paths in a state graph that start in the initial state, are uniquely determined by the sequence of actions that occur along them. Observe that a state graph need not contain all states of the corresponding program. If it does contain all reachable states and, moreover, if st is an isomorphism with respect to the set of reachable states, we call it the *full state graph*. We shall assume state graphs not to contain unreachable states.

A program P together with a valuation function $V: Q \rightarrow 2^{P^V}$ defines models $(G_P, \hat{V})$ for any state graph G_P for P and function $\hat{V}: S \rightarrow 2^{P^V}$ defined by $\hat{V}(s) = V(st(s))$. In the sequel, we shall not distinguish between V and $\hat{V}$.

Partial order reduction exploits concurrency in programs and the fact that truth of specifications is often insensitive to the order in which so-called *independent* actions from different concurrent components occur in computations. Such independent actions can be, e.g., assignments to variables that are local to different components and send-actions in different components that affect separate message queues. The information as to which actions are independent can be given in an abstract way as follows:

Definition 2.1 *An independence relation is an irreflexive and symmetric relation $I \subseteq T \times T$ such that for each pair of operations $(a, b) \in I$ (called independent operations), for each $q \in Q$,*

- *If $a \in en(q)$ (i.e., a is enabled from q), then $b \in en(a(q))$ iff $b \in en(q)$.*
- *If $a, b \in en(q)$ then $a(b(q)) = b(a(q))$.*

If $(a, b) \notin I$, then a and b are called dependent.

3 Stuttering Equivalence

The correctness of the reduction method will be based on the notion of stuttering equivalence.

Let $M = ((W, (\xrightarrow{a})_{a \in T}, w^0), V)$ and $M' = ((W', (\xrightarrow{a'})_{a \in T}, w^{0'}), V')$ be two finite models.

Definition 3.1 ([1]) *A relation $\sim \subseteq W \times W'$ is a stuttering equivalence between M and M' if the following conditions hold:*

1. $w^0 \sim w^{0'}$.

2. if $w \sim w'$, then $V(w) = V'(w')$ and for every path π of M that starts at w , there is a path π' in M' that starts at w' , a partition $B_1, B_2 \dots$ of π , and a partition $B'_1, B'_2 \dots$ of π' such that for all $j \geq 0$, B_j and B'_j are nonempty and finite, and every state in B_j is related by ' $\sim$ ' to every state in B'_j and
3. the same condition as (2) interchanging π and M with π' and M' .

In [1] stuttering equivalence is defined using approximants $\sim^n$. Because our models are finite, it is easy to see that the two definitions are equivalent.

Theorem 3.2 ([1]) *Let φ be a CTL*-X formula with the set of atomic propositions PV . Let M and M' be two models with the same set of atomic propositions PV and let the relation $\sim$ be a stuttering equivalence between M and M' . If $w \sim w'$, then $M, w \models \varphi$ iff $M', w' \models \varphi$.*

The reverse of this theorem, essentially stating that no coarser equivalence preserves the truth value of CTL*-X formulas, also holds [1]. This means that as far as the equivalence is concerned, stuttering equivalence enables maximal reduction.

4 The Algorithm

The reduction algorithm is based upon a modified depth-first-search algorithm. It generates a reduced state graph G' for the checked program P such that the correctness of any checked property φ under G' is the same as under the full state graph G of P . This is guaranteed by ensuring that the model corresponding to G is stuttering equivalent with the model corresponding to G' (see Theorem 3.2).

The idea of the reduction is that from each state in the reduced state graph the set of enabled operations is examined, and only a subset of it is used to generate successors. This contrasts with the construction of the full state graph, where all of the enabled operations are expanded. The subset of the operations $E(q)$ taken from a state q satisfies restrictions **C0**, ..., **C3** below, in order to preserve stuttering equivalence between the full and the reduced model.

Figure 1: Possibilities of reduction by not expanding a .

To explain the restrictions imposed on the set $E(q)$ let's assume first that the full model M does not contain loops except for self loops. Definition 3.1 describes the cases in which the model M' resulting after removing a transition from M is stuttering equivalent with M . In Figure 1 we have indicated the two situations in which the a -labeled transition need not be expanded from state q in M . This is the case when the states q'' and q' are stuttering equivalent (denoted $q'' \sim q'$) or when the states q and q' are stuttering equivalent ($q \sim q'$) in the full model M .

If $q'' \sim q'$ or $q \sim q'$, then for each path π with the prefix qq'' , there is a path π' with the prefix qq' such that two partitions of π and π' satisfying Condition (2) of Definition 3.1 exist. Notice that it is because of the absence of non-trivial self-loops in M that the path π' cannot contain the transition

$q \xrightarrow{a} q''$. If it could, M would not need to be stuttering equivalent with M' since $q \xrightarrow{a} q''$ is not present in M' . Therefore, the full model M remains stuttering equivalent with the model M' obtained after removing the transition $q \xrightarrow{a} q''$ from M .

As for ensuring $q'' \sim q'$, we do not know of any efficiently checkable condition that would imply this. Indeed, the general problem is PSPACE-hard in the number of program operations, as it depends on the subgraphs of nodes reachable from q'' and q' . Therefore, we concentrate on the second case: ensuring that $q \sim q'$.

First observe that by repeatedly applying the argument above, it follows that if $q \sim q'$ then it suffices to only have the transition $q \xrightarrow{b} q'$ from q in M' , i.e., the subtree of other transitions (indicated by the triangle in the figure) can be ignored. Hence, to reduce most effectively, we shall require the algorithm to use singleton sets whenever reduction is possible; thus we obtain

C0 $E(q)$ contains either all operations enabled in state q , or exactly one of these; i.e., $E(q)$ is a singleton.

The next condition will make sure that the execution of b does not change any propositional variable used in φ assigned to q and q' , which is a necessary condition for $q \sim q'$.

To present it, the following definition is needed:

Definition 4.1 An operation $a \in T$ is visible if it can change the truth value of some proposition in φ ; i.e., if $V(q) \neq V(a(q))$ for some state q such that $a \in en(q)$. Let Vis be the set of visible operations in T .

A good practical approximation for calculating Vis is as follows: Consider each program operation that can change one of the propositions as visible. (This approximation might not calculate the minimal set of visible operations, therefore allowing less reduction. However, it is safe in the sense that it would not miss a visible operation.)

In keeping with extant literature [21], this condition is called **C2**:

C2 If $E(q) \neq en(q)$, the operation in $E(q)$ is not visible.

In formulating the subsequent conditions we use the fact that we have already imposed conditions **C0** and **C2**.

The general problem of showing that $q \sim q'$ still is PSPACE-hard in the number of program operations. So, we aim for a stronger condition: for every path π starting in q there is a path π' starting in q' that is the same upto invisible actions. Now, consider π . As long as the actions along π are independent of b there is no problem in constructing π' because independent actions commute, so that these π -actions can still occur, in the required order, after the (invisible) b -action. Note that, like before, absence of non-trivial loops matters here. Dependent actions do cause a problem because there is no way to ensure that such actions can still occur without exploring all paths starting in q' . So, we disallow this situation by stipulating that such actions can only occur *after* the b -action has occurred.

C1 No operation $a \in T \setminus E(q)$ that is dependent on the operation in $E(q)$ can be executed in P before the operation from $E(q)$ is executed.

Now, consider the first action c along π that depends on b and let $\bar{q}$ be the state on π from which c is taken. Since the b -action must have occurred along π before reaching state $\bar{q}$, commutativity of independent actions implies that the constructed prefix of π' ends in state $\bar{q}$, from which c and, indeed, the whole sequence of subsequent actions along π can be taken.

Condition **C1** occurs in many variations in LTL-X preserving reduction methods [12, 20, 23, 6].

The final restriction [20] is needed in case M does have non-trivial loops and restricts omitting operations along such loops. Consider the figure on the right, essentially extending Figure 1 with a non-trivial loop. Choosing $E(q) = \{b\}$, $E(q') = \{b'\}$, which satisfy **C0**, **C1** and **C2** (when only a is visible), yields an incorrect reduction, where a is absent. **C3** prohibits completely omitting such operations occurring along a nontrivial loop. In the algorithm it is detected, (as closing a cycle on the search stack) when a node is part of a loop. It is at this node that the operation is added.

C3 If $E(q) \neq en(q)$, then the operation in $E(q)$, when applied to the current state q , does not close a cycle on the search stack (i.e., we don't allow that an open node with value $a(q)$ exists on the search stack).

Note that in the example **C3** only requires to add one of the two a 's, namely the one from q to q'' . This in turn necessitates that q'' together with the corresponding b and b' are added to the reduced model. So only the a operation from q to q'' can be removed.

The conditions **C1**, **C2**, **C3** are sufficient to guarantee that the reduced state graph will preserve any checked *linear* temporal logic property φ [21]. The condition **C0** is newly introduced for branching temporal logics.

Figure 2: Correct **CTL*-X** reductions

Examples of reductions that use a subset satisfying the conditions **C0**, ..., **C3** are shown in Figure 2. The dotted and solid transitions together constitute the full model. The reduced model consists of only the solid transitions.

The reduced state graph generation program is given in Figure 3. Each node s represents some program state $st(s)$. The construction starts with a node whose state is the initial state q_0 of the program. The main program consists of a depth-first search (lines 4-17). Each new state is marked by the flag *open* (lines 2 and 12), and when its expansion is finished (i.e., it is removed from the search stack), it is marked by *closed* (line 16). For each new state, the subset of successors to be used is calculated by the procedure *ample* (the procedure call is in line 5, the procedure body is in lines 18-25). This procedure returns either a single (invisible) operation that satisfies conditions **C0**, ..., **C3** (line 22) or the set of all enabled operations (line 24).

Checking that a singleton set $\{a\}$ satisfies condition **C1** is not detailed in the algorithm given in Figure 3. Because finding optimal *ample* sets is NP-complete ([20]), any implementation of *ample*

will use heuristics that may also depend on the specific programming language used (to define a finite state program in our sense).

Such heuristics are based on checking the type of the operation a (e.g., a local assignment, a synchronous receive operation, etc.) and some conditions on the rest of the program, and the state of the current node s : according to the type of the operation, there are certain conditions whose satisfaction in the current state s guarantee that $\{a\}$ satisfies **C1**. For example, the simplest condition is that a is a local assignment, and is not within a non-deterministic choice with other operations. A slightly more complicated condition applies when a is a non-synchronous receive. Then **C1** is guaranteed if there is no other receive operation from the same queue in any other process (this holds vacuously when a communication queue can be shared only by a pair of processes), and the queue is not empty in the state $st(s)$. A more complete description of checking **C1** appears in [11].

```

1  create_node(s, q0);
2  set open(s);
3  expand_node(s);

4  proc expand_node(s);
5      working_set(s) := ample(st(s)); /* find set of operations to expand from s */
6      while working_set(s) ≠ ∅ do
7          a := some operation of working_set(s);
8          working_set(s) := working_set(s) \ {a};
9          succ_state := a(st(s)); /* the a-successor of st(s) */
10         if new(succ_state) then
11             create_node(s', succ_state); /* s' has value succ_node */
12             set open(s'); /* set s' to open, i.e., on the search stack */
13             expand_node(s') fi /* expand the successors of s' */
14             create_edge(s, a, s');
15         end while;
16         set closed(s); /* close s, i.e., s is not on the search stack */
17     end expand_node.

18  proc ample(x):element of T;
19      foreach a ∈ T \ Vis /* for every invisible (due to C2) operation */
20          if {a} satisfies C1 for x and
21             not exists s' with st(s') = a(x) and open(s') /* Check C3 */
22          then return({a}) fi /* singleton set, due to C0 */
23      end foreach;
24      return(en(x)); /* cannot find good operation; choose all enabled ones */
25  end ample;

```

Figure 3: A reduced state graph expansion algorithm

Let M_P be the model corresponding to the full state graph G_P of given program P and M_r be the model corresponding to the reduced state graph G' generated by our algorithm. We have the following:

Theorem 4.2 M_P is stuttering equivalent with M_r .

The correctness of the algorithm is based on the relation found between **CTL*-X** validity and a stuttering equivalence of models [1]. Proving the correctness of the above algorithm is based on constructing an equivalence relation $\sim$ on the states of the full state graph M_P and showing that it

is a stuttering equivalence. By construction, the relation $\sim$ relates only pairs of states that have the same set of atomic propositions. Moreover, for each $v \sim w$ such that w remain in the reduced state graph M_r , if π is a finite or infinite sequence from v , then there exists a sequence π' from w such that

- π' contains only edges that remain in the reduced state graph M_r , and
- π and π' can be partitioned into segments that pairwise correspond according to the conditions 2 and 3 of Definition 3.1.

Since the initial state w^0 remains in the reduced state graph, the above conditions guarantee that the reduced state graph M_r simulates the full state graph M_P . The other direction of the bisimulation stems from the fact that $\sim$ is an equivalence relation, hence reflexive. Thus, it follows from Theorem 3.2 that the reduction preserves the validity of the checked formula.

The construction of the relation $\sim$ and the detailed proof are delegated, due to lack of space, to an appendix.

4.1 Complexity of the Algorithm

The time complexity of the algorithm is $O(n_r \cdot C + m_r)$, where n_r is the number of states in the generated state graph, m_r is the number of edges and C is the complexity of computing an ample set. This is obvious as the algorithm is a modified depth-first search through the state graph. Computing ample sets can be done in constant time along the lines of the earlier explanation. As to the amount of space, this is clearly linear in the number of states and edges. Hence we obtain an $O(n_r + m_r)$ space and time complexity for the algorithm.

5 Experimental Results

5.1 Reduced State Space for Various Algorithms and Protocols

The algorithm described in this paper was implemented by Gerard Holzmann in SPIN [9] and run on several examples. The table in figure 4 below contains the number of states and edges, memory used in bytes, and time in seconds of generating full state graphs and reduced state graphs for both **LTL-X** and **CTL-X**. The reduction for **CTL-X** contains an additional restriction, namely **C0**, on selecting the subset of successors. This restricts the subsets of successors to be either the full set of the enabled operations, or a singleton set. In order to make the comparison unbiased towards any particular checked property, all operations were considered invisible during the tests. The set of properties that can be checked *without* making any program operation visible includes properties such as deadlock and termination.

All measurements were made on a Sparc-10 workstation with 128Mbyte of RAM. The runtimes are the sum of system-time and user-time. The algorithms checked are as follows: *leader* is a leader election algorithm for an unidirectional ring [3], *sorting* is a pipeline distributed sorting algorithm, *urp* is AT&T universal receiver protocol, *dtp* is a data transfer protocol, *snoopy* is a cache coherence protocol, *pftp* is a file transfer protocol [9] and *tpc* is a model of a telephone switch. For the first two examples, *leader* and *sorting*, the reduction with and without the additional restriction **C0** are the same. Both give (when repeated with different numbers of processes) an exponential reduction of the state graph. For *urp* and *dtp*, the reduction in space and time is very similar with and without the additional restriction. For *snoopy*, the **CTL-X** reduction generates a state graph that is about 25% bigger in space, and takes about 50% more time. For *pftp*, the reduction is about twice better in space and time for the **LTL-X** reduction, and for *tpc*, it is about 2.5 times better in space and more than three times better in time.

Algorithm	Reduction	States	Transitions	Memory	Time
leader	Non	382,064	1,847,294	88,029,016	97.1
	LTL-X	94	94	1,111,896	(<0.1)
	CTL-X	94	94	1,116,304	(<0.1)
sorting	Non	659,683	3,454,989	113,629,016	145.5
	LTL-X	182	182	1,120,088	(<0.1)
	CTL-X	182	182	1,124,496	0.1
urp	Non	4,329	16,563	1,943,416	0.9
	LTL-X	1,149	2,160	1,542,008	0.2
	CTL-X	1,231	2,388	1,598,304	0.3
dtp	Non	251,409	648,467	36,369,976	32.2
	LTL-X	16,459	17,603	4,290,104	1.5
	CTL-X	17,100	18,320	4,540,224	1.6
snoopy	Non	191,498	938,719	24,446,520	48.2
	LTL-X	34,544	65,468	6,080,056	5.1
	CTL-X	46,329	99,733	7,620,416	7.5
pftp	Non	1,067,787	4,545,778	170,440,248	625.5
	LTL-X	136,208	257,019	23,565,880	30.1
	CTL-X	259,252	597,819	43,120,448	59.5
tpc	Non	3,918,286	11,762,426	251,827,768	7202.9
	LTL-X	391,534	461,534	26,863,160	31.0
	CTL-X	977,454	1,727,034	64,472,896	98.1

Figure 4: CTL-X versus LTL-X reductions

These results demonstrate that the inclusion of the reduction algorithm is beneficial for all the above examples. A substantial reduction can be achieved with relatively small cost, as the implementation of the reduction algorithm is simple and incurs only very small overhead (for further implementation details, refer to [11], where an efficient LTL-X implementation is described). Even in the cases where the reduction is not very big (in comparison to some other reductions, as for the *leader* algorithm), such as in the *tpc* algorithm, where the gain in space is a factor of four, one can obtain a considerable benefit: since the algorithm is complicated enough to consume a large amount of memory, even the fourfold memory reduction could reduce the execution time from over two hours to about a minute and a half (avoiding needless memory swaps).

5.2 Verifying Branching Bisimulation

The reduction method described in Section 4 can be further exploited in the context of process algebra. It can be used to verify whether two states of a program are branching bisimilar [15, 19].

Let *Invis* denote the set of invisible operations, i.e., $Invis = T \setminus Vis$. As we identify each operation from *Invis* with a silent step, the definition of branching bisimulation can be formulated as follows:

Definition 5.1 *A relation $\sqsubseteq \subseteq W \times W'$ is a branching simulation between M and M' if it satisfies the following conditions:*

(a) $w^0 \sqsubseteq w'^0$ and

(b) if $w \sqsubseteq w'$, then if $w \xrightarrow{a} v$, then either $a \in Invis$ and $v \sqsubseteq w'$ or there are states $w' = w'_0, w'_1, \dots, w'_{n+1} = v'$ such that for each $0 \leq i < n$, $w'_i \xrightarrow{a_i} w'_{i+1}$, $a_i \in Invis$, $w \sqsubseteq w'_{i+1}$, and

$$w'_n \xrightarrow{a'} v', v \sqsubseteq v'.$$

A relation $\sim$ is a branching bisimulation ([18]) if both $\sim$ and $\sim^T$ (the transpose of $\sim$) are branching simulations.

M and M' are branching bisimilar if there is a branching bisimulation relation between M and M' .

We have the following:

Theorem 5.2 *Given a program P , M_P is branching bisimilar with M_r .*

The proof is given in the Appendix.

Verifying whether or not two states are branching bisimilar is done by constructing the minimal branching bisimilar state graph of the program. The most efficient algorithm for this was published by Vaandrager and Groote in [8] and has time and space complexity of $O(n^2 + nm)$ and $O(n + m)$, respectively, where n and m are the number of states and edges in the full state graph. For determinate programs, [22] presents a more efficient algorithm with time and space complexity $O(n \log n + m)$ and $O(n + m)$, respectively. In [8] it was conjectured that the same time complexity suffices for minimizing arbitrary state graphs.

By Theorem 5.2, our algorithm generates a state graph that is branching bisimilar to the full state graph. Moreover it has time and space complexity $O(n_r + m_r)$, where n_r and m_r are the numbers of generated states and edges. This raises the possibility of using our algorithm as a preprocessing phase to constructing a minimal branching bisimilar state graph, thus allowing the minimization algorithm to run in time $O(n_r^2 + n_r m_r)$ using $O(n_r + m_r)$ space. The algorithms [8, 22] can be applied to a reduced state graph, instead of the full one.

The ‘benchmark’ example used in literature, and by us, is Milner’s scheduler as described in [17]. This is a simple token ring consisting of k cyclic processes C_i , which, on having received the token, communicate with some system and then concurrently wait for acknowledgment and wait to pass on the token. Process C_i is described by the CCS equation $C_i = \bar{t}_i \cdot c_i \cdot (\bar{a}_i \mid t_{i \bmod k + 1}) \cdot C_i$. The complete scheduler on k processes is described by $Sch_k = (t_1 \cdot nil \mid C_1 \mid \dots \mid C_k) \setminus t_1 \dots \setminus t_k$, where the first process starts C_1 . The operator ‘ $\mid$ ’ denotes a concurrent composition, and ‘ $\cdot$ ’ means sequencing. Letters correspond to operations. Two letters, where one is barred, e.g., c and $\bar{c}$, may synchronize, thus producing an invisible action. The operator ‘ $\setminus$ ’ is the restriction operator which, in this case, forces the $t_i, \bar{t}_i$ synchronizations to occur. nil is the idle process that does nothing.

In Figure 5 we have collected some results for various sizes k of the token ring. The measurements were done on a Sparc1⁺-workstation with 16MB of memory. The actual generation of both the reduced and the full state graphs are achieved by a script written in PERL, an interpreted language with heavy use of pattern matching, and the absolute times should be interpreted accordingly. A C implementation can be expected to run at least an order of magnitude faster. The number of states and edges in the full state graph is given in the 2nd and 3rd columns. We consider both the case that only the communication actions (c_i) are visible and the case that both c_i and the acknowledgment actions ($\bar{a}_i$) remain visible. For both cases we give the sizes of the state graphs as generated by our algorithm and the minimal state graphs (as given by an implementation of the Vaandrager/Groote algorithm, part of the PSF [15] toolkit). The *time* column gives the time in seconds that our algorithm needs to generate the state graph, where only c_i is visible. We see that in this case not only that the resulted reduced state graphs is small but the time to generate them is small as well. This should be contrasted with the figures in the 2nd to last column that give the times it takes to generate the full state graph. The time for the actual minimalization of the reduced state graph is negligible for these sizes of state graphs. In other words, one gains considerably here by generating the state graph using our reduction algorithm.

Milner's scheduler			only c_i visible						both c_i and $\bar{a}_i$ visible					
			PO			minimal			PO		PO & τ -removal		generation time*	
k	states(s)	edges(e)	s	e	time*	s	e	s	e	s	e	full	PO	PO- τ
4	97	241	13	13	0	4	4	97	193	64	160	4	3	3
5	241	721	16	16	1	5	5	241	561	160	480	9	9	8
6	577	2017	19	19	1	6	6	577	1,537	384	1,344	25	24	23
7	1,345	5,377	22	22	1	7	7	1,345	4,033	896	3,584	68	61	60
8	3,073	13,825	25	25	1	8	8	3,073	10,241	2,048	9,216	175	156	155
9	6,913	34,561	28	28	1	9	9			these are also the minimal br. bisim. state graphs				
10	15,361	84,481	31	31	1	10	10							
11	33,793	202,753	34	34	2	11	11							
50			151	151	28	50	50							
100			301	301	100	100	100							

*Using a PERL script for state graph generation.

Figure 5: Verifying branching bisimulation

A second experiment shows that even in the case when our algorithm does not substantially reduce the state graph, the overhead of doing the reduction is negligible. In this case, both c_i and $\bar{a}_i$ are made visible and the reduction of the number of edges is only between 11% ($k=4$) and 26% ($k=8$). Here, more than a half of the operations are visible, which defies most of the reduction. This is fortunately untypical. Furthermore, one can see that the minimal state graph also grows exponentially, producing a minimal state graph that is only about 50% smaller than the full state graph. The 3rd to last and 2nd to last columns, marked as “full” and “PO”, show the time it takes to generate the full state graph and reduced state graph, respectively. One can see that even though in this case the reduction is small, the overhead that our algorithm incurs is minimal when compared to generating the full state graph; in fact, the algorithm still runs a little faster.

Although minimizing a state graph w.r.t. branching bisimulation is a global process, certain equivalence preserving transformations can be done locally during state graph generation. For instance, states that have precisely one outgoing transition can be removed if that transition is invisible¹. The column labeled ‘PO & τ -removal’ shows the result of augmenting our partial order algorithm with invisible-step removal. There is now a reduction in the number of states as well as a more substantial reduction in the edges. Interestingly, the resulting state graphs are in fact the minimal branching bisimilar ones. The last column shows that there is no time penalty. In fact, the running times are almost the same, which is not surprising because the algorithm has to visit the same number of nodes as before. Note however that the minimization algorithm will run in time and space proportional to the size of the reduced graph. Hence, invisible-step removal is advantageous for the minimization phase.

6 Conclusions

We have presented an algorithm for generating reduced state graphs to be used for model-checking branching temporal properties. The usual DFS expansion algorithm was modified so that only subsets of the successors from each state are expanded. This allows reducing the number of states and edges,

¹More formally: the rewrite rule $x \cdot \tau \rightarrow x$ preserves branching bisimulation.

and thus allows reducing the space and time used for this construction and for model checking. The branching time logics include the temporal logic **CTL*-X**, which is more expressive than the linear time logic **LTL-X**. They also include the logic **CTL-X** which has a model-checking algorithm that is linear in the size of the checked property [2]. These advantages in either expressiveness or efficiency can now be combined with the ability to reduce the state graph using partial order methods.

On the other hand, we have shown that, in general, the reduction of the state graph for preserving branching properties is more restricted than the one for **LTL-X**: an additional restriction was added, limiting the subset of successors taken from each state to be either the full set of successors or a singleton set.

Experimental results show that the suggested algorithm results in a substantial reduction in both space and time over the traditional full state graph exploration. Also, the algorithm proved to be the preferred way to generate state graphs to verify branching bisimulation.

Acknowledgments The authors would like to thank Gerard Holzmann for implementing the additional constraint into the partial order version of SPIN, and helping with the experiments. Also, hats off for Larry Wall's *Perl*, which proved to be an excellent rapid prototyping tool as well.

References

- [1] M.C. Browne, E.M. Clarke, O. Grumberg, Characterizing Finite Kripke Structures in Propositional Temporal Logic, *Theoretical Computer Science* 59 (1988), Elsevier, 115–131.
- [2] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach, *ACM Transactions on Programming Languages and Systems*, 8(2), 1986 244–263.
- [3] D. Dolev, M. Klawe, M. Rodeh, An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle, *Journal of Algorithms*, 3 (1982), 245–260.
- [4] E.A. Emerson, J.Y. Halpern, “Sometimes” and “not never” revisited: on branching versus linear time temporal logic, *Journal of the ACM*, Vol 33, 1986, 151–178.
- [5] P. Godefroid, Using partial orders to improve automatic verification methods, in E.M. Clarke, R.P. Kurshan (eds.), *Computer Aided Verification 1990*, DIMACS, Vol 3, 1991, 321–339.
- [6] P. Godefroid, D. Pirotin, Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract), in C. Courcoubetis (ed.), *Computer Aided Verification 1993*, LNCS 697, 1993, 438–449.
- [7] P. Godefroid, P. Wolper, A Partial Approach to Model Checking, *Proc. of LICS*, 1991, 406–415.
- [8] J.S. Groote, F. Vaandrager, An efficient algorithm for branching bisimulation and stuttering equivalence, *Proc. of ICALP 1990*, 626–638.
- [9] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1992.
- [10] G.J. Holzmann, P. Godefroid, D. Pirotin, Coverage preserving reduction strategies for reachability analysis, *Proc. IFIP Symp. on Protocol Specification, Testing, and Verification*, June 1992, Orlando, U.S.A., 349–364.
- [11] G.J. Holzmann, D. Peled, An improvement in formal verification, to appear in *Proc. of FORTE'94*, 1994.
- [12] S. Katz, D. Peled, Verification of distributed programs using representative interleaving sequences, *Distributed Computing* 6 (1992), 107–120.
- [13] L. Lamport, What good is temporal logic, in R.E.A. Mason (ed.), *Proceedings of IFIP Congress*, North Holland, 1983, 657–668.

- [14] O. Lichtenstein, A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification, *11th ACM POPL*, 1984, 97–107.
- [15] S. Mauw, G. J. Veltink. A process specification formalism. *Fundamenta Informaticae XIII*, 85–139, 1990.
- [16] A. Mazurkiewicz, Trace semantics, in W. Brauer, W. Reisig, G. Rozenberg (eds.), *Advances in Petri Nets 1986*, LNCS 255, Springer, 1987. 279–324.
- [17] R. Milner, *A Calculus of Communicating Systems*. LNCS 92, Springer, 1980.
- [18] R. deNicola, F. Vaandrager, Three Logics for Branching Bisimulation, *Proc. LICS'90*, 1990, 118–129.
- [19] R. de Simone, D. Vergamini, Aboard AUTO. Technical report 111, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1989.
- [20] D. Peled, All from one, one for all, on model-checking using representatives, *5th international conference on Computer Aided Verification*. Greece. 1993, LNCS, Springer, 409–423.
- [21] D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, *6th international conference on Computer Aided Verification*. Stanford, California. 1994, LNCS 818, Springer, 377–390.
- [22] H. Qin, Efficient Verification of Determinate Processes. *Proc. CONCUR'91*. LNCS 527, 470–479, Springer, 1991.
- [23] A. Valmari, Stubborn sets for reduced state graph generation. *10th International Conference on Application and Theory of Petri Nets*. Vol. 2. 1989. 1–22. Bonn.
- [24] A. Valmari, A Stubborn attack on state explosion. in E.M. Clarke, R.P. Kurshan (eds.), *CAV'90*, DIMACS, Vol 3. 1991. 25–42.

A Appendix: Correctness of the Algorithm

Let $M_P = ((W, \longrightarrow, w^0, L), V)$ be a model, where $(W, \longrightarrow, w^0, L)$ is the full state graph of P and let $M_r = ((W_r, \longrightarrow_r, w^0, L_r), V_r)$ be a model with a reduced state graph as generated by the algorithm, where $w^0 \in W_r \subseteq W$, $\longrightarrow_r \subseteq \longrightarrow$, $L_r = L|_{\longrightarrow_r}$ (reads L restricted to the states that participate in the reduced transition relation $\longrightarrow_r$) and $V_r = V|_{W_r}$. Because of Theorem 3.2, we just have to prove that M_P is stuttering bisimilar with M_r .

To do so we assign to each state $w \in W$ in M_P the set of operations $E(w) \subseteq T$ such that

- if $w \in W_r$, then $E(w)$ is equal to the set of operations expanded by the algorithm from w ; i.e., $E(w)$ is equal either to $en(w)$ or to one invisible operation from $en(w)$ satisfying the conditions **C1–C3**.
- if $w \in W - W_r$, then $E(w) = en(w)$; i.e., $E(w)$ is equal to the set of enabled operations at w .

In the sequel, let $M' = ((W', \longrightarrow', w^{0'}, L'), V')$ be the model obtained from M_P by removing all the transitions $w \xrightarrow{a} w'$ such that $a \notin E(w)$. Hence, $w^{0'} = w^0$. One can easily notice that the reachable part of M' is equal to M_r . So, to prove that M_P is stuttering bisimilar with M_r , we just have to show that M' is stuttering bisimilar with M_P .

We now discuss how such a stuttering equivalence is obtained. The following notation will be used, where $\sim$ is an equivalence to be fixed in the sequel:

- $w \xRightarrow{a} w'$ denotes $w \xrightarrow{a} w'$ and $w \sim w'$.
- $w \xrightarrow{a}_E w'$ denotes $w \xrightarrow{a} w'$ and $a \in E(w)$.
- $w \Rightarrow_E w'$ denotes $w \xRightarrow{a} w'$ and $a \in E(w)$.
- when convenient, we will sometimes omit labels of transitions writing $w \longrightarrow_E w'$, $w \Rightarrow_E w'$ etc.

As M' is a submodel of M_P , we shall simplify notation by constructing a stuttering equivalence on M_P . We start out by building a relation $\sim$ in M_P which, as shown in Lemma A.7, only relates states with identical valuations. Furthermore, as shown in Lemmas A.11 and A.13, $\sim$ satisfies the following properties:

- P1) if $v \sim w$, $w \in W' \subseteq W$ and $v \longrightarrow r'$ and $\neg (r \sim r')$, then there is a non-empty sequence of transitions $w = w_0 \Rightarrow_E w_1 \Rightarrow_E \dots \Rightarrow_E w_k \longrightarrow_E w'$ in M_P such that $r' \sim w'$,
- P2) if $v \sim w$ and there is an infinite stutter path $v = v_0 \Rightarrow v_1 \Rightarrow v_2 \dots$, then there is an infinite stutter path $w = w_0 \Rightarrow_E w_1 \Rightarrow_E \dots$.

The following theorem shows that $\sim$ is a stuttering equivalence consisting only of transitions which are not removed from M_P .

Theorem A.1 *The relation $\sim$ is a stuttering equivalence on M_P such that if $v \sim w$, $w \in W' \subseteq W$ and π is a path starting at v , then a corresponding path π' (as in Definition 3.1) can be taken, starting with w , and consists of $\longrightarrow_E$ transitions only.*

Proof. By definition of $\sim$, $w^0 \sim w^0$. Because $\sim$ is symmetrical, it suffices to establish Clause 2 of Definition 3.1. Let $v \sim v'$. Then $V(v) = V(v')$ holds by definition. Next, let π be a path $v = v_0 \longrightarrow v_1 \longrightarrow \dots$ starting at v . Partition π into blocks $B_0, B_1, \dots$, where every B_j is a maximal subsequence of π of $\sim$ -related states. Every B_i is finite, except for possibly the last block. If B_0 is finite with last state v_k , then $v_k \sim v'$ and $v_k \not\sim v_{k+1}$. By Lemma A.11, Property P1 holds so that there

is a non-empty sequence of transitions $v' = v'_0 \Rightarrow_E v'_1 \Rightarrow_E \dots \Rightarrow_E v'_l \xrightarrow{-E} v'_{l+1}$ s.t. $v_{k+1} \sim v'_{l+1}$. We take $B'_0 = v'_0 \dots v'_l$ and have $v_i \sim v_j$ for every $v_i \in B_0$ and $v'_j \in B'_0$. Because $v_{k+1} \sim v'_{l+1}$, this sets us up to construct $B'_1, B'_2, \dots$: as long as the corresponding B_j 's are finite. Finally, let some B_n be infinite with starting state v_r and let B'_{n-1} have v'_{s-1} as last state, so that $v_r \sim v'_s$.

By Lemma A.13, $\sim$ satisfies Property P2, so that there is an infinite path $v'_s = v'^0_s \Rightarrow_E v'^1_s \Rightarrow_E \dots$. As all states in B_n and the constructed path starting in v'_s are $\sim$ -related, we can partition these suffixes into finite blocks any way we want. This concludes the proof as an infinite block obviously is the last block. $\square$

Corollary A.2 M_P is stuttering equivalent with M_r .

Proof. We have M_P is stuttering equivalent with M' . This is immediate from Theorem A.1 and the fact that all $\xrightarrow{-E}$ steps are present in M' . As the reachable part of M' and M_r are isomorphic as models, M_r and M' are stuttering equivalent. The Corollary follows by transitivity. $\square$

Thus, by corollary A.2, the reduction is correct.

The proof proceeds by first defining an equivalence that satisfies weaker properties.

Definition A.3 Let $M = ((W, \xrightarrow{-}, w^0, L), V)$ and $M' = ((W', \xrightarrow{-}', w'^0, L'), V')$ be two finite models.

A relation $\sqsubseteq \subseteq W \times W'$ is a divergence blind stuttering simulation between M and M' if it satisfies the following conditions:

(a) $w^0 \sqsubseteq w'^0$ and

(b) if $w \sqsubseteq w'$, then $V(w) = V'(w')$, and if $w \xrightarrow{-} v$ then there are states $w' = w'_0, w'_1, \dots, w'_n = v'$ such that for each $i < n$, $w'_i \xrightarrow{-'} w'_{i+1}$ and $w \sqsubseteq w'_i$, and $v \sqsubseteq v'$. This is called the transfer property.

A relation $\sim$ is a divergence blind stuttering equivalence ([18]) if both $\sim$ and $\sim^T$ (the transpose of $\sim$) are divergence blind stuttering simulations.

M and M' are divergence blind stuttering equivalent if there is a divergence blind stuttering equivalence relation between M and M' .

The desired divergence blind stuttering equivalence is constructed as follows:

Definition A.4 Let $\sqsubseteq_c \subseteq W \times W'$ be a relation in M_P defined as $\bigcup_{b \in T} \bigcup_{i < \omega} F_b^i(\sqsubseteq_c^b)$ where

(a) $\sqsubseteq_c^b = \{ (u, v) \mid u = v \text{ or } u \in W_r, u \xrightarrow{b} v, b \in E(u) \neq \epsilon n(u) \}$,

(b) for $R \subseteq W \times W'$, $F_b^0(R) = R$, $F_b^{i+1}(R) = F_b^i(R) \cup F_b(F_b^i(R))$, and

(c) $F_b(R) = \{ (u, v) \mid \exists u', v', a \ (u', v') \in R, u' \xrightarrow{b} v', u' \xrightarrow{a} u, v' \xrightarrow{a} v, a \neq b \}$

The next lemma gives two properties of $\sqsubseteq_c$ needed in the rest of the proof.

Lemma A.5 The relation $\sqsubseteq_c$ has the following properties.

(a) If $w \sqsubseteq_c z$ and $w \neq z$, then for some b , $w \xrightarrow{b} z$ and $(a, b) \in I$ for all $a \in \epsilon n(w) - \{b\}$,

(b) $\sqsubseteq_c$ is a divergence blind stuttering simulation in M_P .

Proof. (a) We prove by induction on k the property $\forall u, v, b \ \forall k \ P_b^k(u, v)$, where $P_b^k(u, v)$ is defined as:

if $(u, v) \in F_b^k(\sqsubseteq_c^b)$ we have both $V(u) = V(v)$ and if $u \neq v$ then there are u_0 and v_0 such that $u_0 \in W_r$, $u_0 \xrightarrow{b} v_0$, $b \in E(u_0) \neq en(u_0)$ (hence, $u_0 \sqsubseteq_c^b v_0$), $(u, v) \in F_b^k(\{(u_0, v_0)\})$, $u \xrightarrow{b} v$ and $\forall a \in en(u) - \{b\} \ (a, b) \in I$.

Obviously, this implies the first part of the Lemma. For the proof below, fix some action b and nodes u, v such that $(u, v) \in F_b^k(\sqsubseteq_c^b)$.

For $k = 0$, we have $u \sqsubseteq_c^b v$ so that $u = v$ or $u \xrightarrow{b} v$ for some invisible (by C3) b . In either case $V(u) = V(v)$. To establish the remainder, take $u_0 = u$ and $v_0 = v$. Since we may assume $u \neq v$, then $u \in W_r$. As $E(u) \neq en(u)$, we have $\forall a \in en(u) - E(u) \ (a, b) \in I$ by Property C1. This suffices as $E(u) = \{b\}$ by Property C0.

For $k > 0$, we may assume that $(u, v) \in F_b^k(\sqsubseteq_c^b) - F_b^{k-1}(\sqsubseteq_c^b)$. If $u = v$ then $V(u) = V(v)$, so assume otherwise. Since $(u, v) \in F_b^k(\sqsubseteq_c^0) \cap F_b^{k-1}(\sqsubseteq_c^0)$, $u \neq v$. There are u', v' and a such that $(u', v') \in F_b^{k-1}(\sqsubseteq_c^b)$, $u' \xrightarrow{b} v'$, $u' \xrightarrow{a} u$, $v' \xrightarrow{a} v$ and $a \neq b$. As $u \neq v$, we must have $u' \neq v'$, by definition of a model. Since $P_b^{k-1}(u', v')$ holds by induction, we obtain u_0 and v_0 such that $u_0 \in W_r$, $u_0 \xrightarrow{b} v_0$, $b \in E(u_0) \neq en(u_0)$, $(u', v') \in F_b^{k-1}(\{(u_0, v_0)\})$ and $u' \xrightarrow{b} v'$; moreover, $\forall a \in en(u') - \{b\} \ (a, b) \in I$. Hence, $u \xrightarrow{b} v$ and $V(u) = V(v)$ because b is invisible. It follows that $(u, v) \in F_b^k(\{(u_0, v_0)\})$ and $(a, b) \in I$ for any $a \in en(u) - \{b\}$ by Property C1 and because $en(u) \cap E(u_0) = \{b\}$ by Property C0. We conclude that $P_b^k(u, v)$ holds.

(b) As for proving that $\sqsubseteq_c$ is a branching simulation, clearly $w^0 \sqsubseteq_c w^0$. Now, let $u \sqsubseteq_c v$ and $u \xrightarrow{a} u'$. This means that $P_b^k(u, v)$ holds for some k and b so that $V(u) = V(v)$. If $u = v$ then the transfer property (of Definition A.3 (b)) holds because $v \sqsubseteq_c v$. Otherwise, for some b , $u \xrightarrow{b} v$ and $\forall c \in en(u) - \{b\}$ we have $(b, c) \in I$. If $a = b$ then the transfer property holds, again, because $v \sqsubseteq_c v$. Finally, if $a \neq b$ then $(a, b) \in I$ and, hence, $v \xrightarrow{a} v'$ for some v' . We obtain $u' \sqsubseteq_c v'$ by definition of $\sqsubseteq_c$ and this establishes the transfer property and completes the proof. $\square$

Definition A.6 The relation $\sim \subseteq W \times W$ is the symmetric and transitive closure of $\sqsubseteq_c$.

Next we will prove that $\sim$ is a divergence blind stuttering equivalence.

Lemma A.7 The relation $\sim$ is a divergence blind stuttering equivalence in M_P .

Proof. Firstly we prove that $\sqsubseteq_c^T$ is a divergence blind stuttering simulation. Clearly $w^0 \sqsubseteq_c^T w^0$. Now, let $u \sqsubseteq_c^T v$ and $u \xrightarrow{a} u'$. So, $v \sqsubseteq_c u$. This means that $P_b^k(v, u)$ holds for some k so that $V(u) = V(v)$. If $u = v$ then the transfer property (of Definition A.3 (b)) holds because $v \sqsubseteq_c^T v$. Otherwise, by Lemma A.5(a), for some b , $v \xrightarrow{b} u$. Therefore, the transfer property holds, again, because $v \xrightarrow{b} u \xrightarrow{a} u'$, $u \sqsubseteq_c^T u$ and $u' \sqsubseteq_c^T u'$.

Therefore, by definition, $\sqsubseteq_c$ is a divergence blind stuttering equivalence and its symmetric and transitive closure is as well. $\square$

In order to prove that $\sim$ satisfies the property P1, we need two auxiliary Lemmas A.8 and A.9. The first lemma states that if a transition labeled a is removed from some state, then a sequence of non-removable stuttering transitions (with labels independent on a) followed by a non-removable transition labeled a will start at that state.

Lemma A.8 Let $w \in W' \subseteq W$ and $a \in en(w)$. Then, there is a (possibly empty) sequence of transitions $w = w_0 \xrightarrow{b_1}_E w_1 \cdots \xrightarrow{b_k}_E w_k$ and the transition $w_k \xrightarrow{a}_E v'$ in M_P such that $w_i \sqsubseteq_c w_{i+1}$, $w_i \neq w_{i+1}$, $b_{i+1} \neq a$ for $i < k$, and $b_1 \cdots b_k a \equiv_I ab_1 \cdots b_k$.

Proof. By induction on the order that the algorithm ‘closes’ nodes, i.e., removing the node from the search stack, after its expansion is finished, setting it to *closed* at line 16 in Figure 3. We claim that the above property holds for each node when it closes, and continues to hold throughout the execution. At the end, all the nodes of M_r are closed, which proves the Lemma. Now, to prove the claim, distinguish between two cases:

1. $E(w) = en(w)$ (all the operations are taken). In this case, the Lemma holds trivially, with an empty sequence of b ’s. This case also correspond to the base of the induction.
2. $E(w) \subset en(w)$; i.e., $E(w)$ is a proper subset of enabled operations. Choose some $b \in E(w)$, which is invisible by C2. Then by Definition A.4, there is a transition $w \xrightarrow{b}_E w'$. Since by C3, this transition cannot close a cycle it holds that $w \neq w'$. It is a property of the DFS (depth-first-search) algorithm, that if this transition does not close a cycle, then w' is closed before w is closed. Thus, by the inductive hypothesis, when w is closed, w' already satisfies the condition of the Lemma; i.e., there is a path $w' = w_0 \xrightarrow{b_1}_E w_1 \cdots \xrightarrow{b_k}_E w_k \xrightarrow{a}_E v'$ such that $w_i \sqsubseteq_c w_{i+1}$, $w_i \neq w_{i+1}$ and $b_{i+1} \neq a$ for $i < k$. Now, according to C1, all the operations that are executed before a from w are independent of a , which means that the path $w \xrightarrow{b}_E w_0 \xrightarrow{b_1}_E w_1 \cdots \xrightarrow{b_k}_E w_k \xrightarrow{a}_E v'$ satisfies the conditions of the Lemma.

□

The next lemma says that if at some state a sequence of stuttering transitions starts, then also a sequence of non-removable stuttering transitions starts such that the ending states of these sequences are connected by a sequence of stuttering transitions.

Lemma A.9 *Let $w \in W' \subseteq W$. If there is a non-empty sequence of transitions $w = w_0 \xrightarrow{a_1} w_1 \cdots \xrightarrow{a_n} w_n$, then there is a non-empty sequence of non-removable transitions $w = w'_0 \Rightarrow_E w'_1 \cdots \Rightarrow_E w'_l$ and there is a (possibly empty) sequence of transitions $w_n = w_n^0 \Rightarrow w_n^1 \cdots \Rightarrow w_n^m = w'_l$ such that $w_n^i \sqsubseteq_c w_n^{i+1}$ and $w_n^i \neq w_n^{i+1}$ for $i < m$.*

Proof. By induction on n . Consider $n = 1$. If $a_1 \in E(w_0)$, then we are done. Assume that $a_1 \notin E(w_0)$. By Lemma A.8, there is a sequence of transitions $w = w'_0 \xrightarrow{b_1}_E w'_1 \cdots \xrightarrow{b_k}_E w'_k$ and the transition $w'_k \xrightarrow{a_1}_E v'$ such that $w'_i \sqsubseteq_c w'_{i+1}$, $w'_i \neq w'_{i+1}$ for $i < k$, and $b_1 \cdots b_k a_1 \equiv_I a_1 b_1 \cdots b_k$. Since a_1 is independent of all b_i , there are states w_1^i such that $w_1 = w_1^0 \xrightarrow{b_1} w_1^1 \cdots \xrightarrow{b_k} w_1^k = v'$ and $w'_i \xrightarrow{a_1} w_1^i$, for $i \leq k$. Because of $w'_i \sqsubseteq_c w'_{i+1}$ for $i < k$, it follows from the definition of $\sqsubseteq_c$ that $w_1^i \sqsubseteq_c w_1^{i+1}$ for $i < k$.

Notice that, by assumption, $w_0 \sim w_1$ and by definition of $\sim$, $w_1 \sim v'$. Thus, by transitivity of $\sim$, $w_0 \sim v'$. Next, by definition of $\sim$, $w_0 \sim w'_k$. It follows from the definition of $\sim$ that $w'_k \sim v'$.

If any transition labeled by b_i is a self-loop, then we remove the state w'_{i-1} from the sequence $w_1 = w_1^0 \xrightarrow{b_1} w_1^1 \cdots \xrightarrow{b_k} w_1^k = v'$, which completes the proof for $n = 1$.

Inductive step. Consider a sequence $w = w_0 \xrightarrow{a_1} w_1 \cdots \xrightarrow{a_{k+1}} w_{k+1}$. By the inductive hypothesis, there is a non-empty sequence of non-removable transitions $w = w'_0 \Rightarrow_E w'_1 \cdots \Rightarrow_E w'_l$ and there is a (possibly empty) sequence of transitions $w_k = w_k^0 \xrightarrow{d_1} w_k^1 \cdots \xrightarrow{d_m} w_k^m = w'_l$ such that $w_k^i \sqsubseteq_c w_k^{i+1}$ and $w_k^i \neq w_k^{i+1}$ for $i < m$.

There are now two possibilities: either $a_{k+1} \neq d_i$ for all $i \leq m$ or there is some $d_j = a_{k+1}$. If $a_{k+1} \neq d_i$ for all $i \leq m$, then from Lemma A.5(a) we have $(a_{k+1}, d_i) \in I$ for all $i \leq m$ and therefore

we can complete the “rectangle” such that $w_k^m \xrightarrow{a_{k+1}} w_{k+1}^m$ and there is a sequence of transitions $w_{k+1} = w_{k+1}^0 \xrightarrow{d_1} w_{k+1}^1 \cdots \xrightarrow{d_m} w_{k+1}^m$. By definition of $\sqsubseteq_c$, $w_{k+1}^i \sqsubseteq_c w_{k+1}^{i+1}$ for $i < m$. If any of the transitions labeled by d_i is a self-loop, then we remove the state w_{k+1}^{i-1} from the sequence. For a_{k+1} from w_k^j we apply the construction shown for the case $n = 1$.

Assume now that j is the smallest index such that $d_j = a_{k+1}$. Then, we can build the “rectangle” until w_{k+1}^{j-1} such that $w_k^{j-1} \xrightarrow{a_{k+1}} w_{k+1}^{j-1}$ and there is a sequence of transitions $w_{k+1} = w_{k+1}^0 \xrightarrow{d_1} w_{k+1}^1 \cdots \xrightarrow{d_{j-1}} w_{k+1}^{j-1}$. By definition of $\sqsubseteq_c$, $w_{k+1}^i \sqsubseteq_c w_{k+1}^{i+1}$ for all $i < j - 1$. If any of the transitions labeled by d_i is a self-loop, then we remove the state w_{k+1}^{i-1} from the sequence. Since $d_j = a_{k+1}$ and there are no two different transitions with the same label, $w_k^j = w_{k+1}^{j-1}$ and we take the existing sequence from w_k^j . This completes the inductive step. $\square$

Corollary A.10 *Let $w \in W$. If there is an infinite sequence of transitions $w = w_0 \xrightarrow{a_1} w_1 \cdots \xrightarrow{a_n} w_n \xrightarrow{a_{n+1}} \cdots$, then there is an infinite sequence of non-removable transitions $w = w'_0 \Rightarrow_E w'_1 \cdots \Rightarrow_E w'_n \Rightarrow_E \cdots$ and $w_i \sim w'_i$ for all i .*

Proof. A consequence of the induction step in the proof of Lemma A.9 and the definition of $\Rightarrow$. $\square$

Now, we are ready to prove that $\sim$ satisfies P1.

Lemma A.11 *The relation $\sim$ satisfies the property P1, i.e., if $v \sim w$, $w \in W' \subseteq W$ and $v \rightarrow v'$ and $\neg(v \sim v')$, then there is a non-empty sequence of transitions $w = w_0 \Rightarrow_E w_1 \Rightarrow_E \cdots \Rightarrow_E w_k \rightarrow_E w'$ in M_P such that $v' \sim w'$.*

Proof. Let $v \sim w$, $v \rightarrow v'$ and $\neg(v \sim v')$. From Lemma A.7 the relation $\sim$ is a divergence blind stuttering equivalence. Then, by definition of divergence blind stuttering equivalence and transitivity of $\sim$, there is a non-empty sequence of transitions $w = u_0 \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_n \rightarrow u'$ such that for all $i \leq n$, $v \sim u_i$ and $v' \sim u'$. It follows from transitivity of $\sim$ that $\neg(u_n \sim u')$. Let a be the label such that $u_n \xrightarrow{a} u'$. Then, by Lemma A.9, there is a non-empty sequence of non-removable transitions $w = w'_0 \Rightarrow_E w'_1 \cdots \Rightarrow_E w'_l$ and there is a (possibly empty) sequence of transitions $u_n = w_n^0 \xrightarrow{d_1} w_n^1 \cdots \xrightarrow{d_m} w_n^m = w'_l$ such that $w_n^i \sqsubseteq_c w_n^{i+1}$ and $w_n^i \neq w_n^{i+1}$ for $i < m$. We show now that $a \neq d_i$ for all $i \leq m$. If $a = d_1$, then $u' = w_n^1$. Since $u_n \sim w_n^1$, $u_n \sim w'$. Contradiction with the assumption that $\neg(u_n \sim u')$. If $a \neq d_1$, then by Lemma A.5(a), $(a, d_1) \in I$ and $a \in \text{en}(w_n^1)$. Let $w_n^1 \xrightarrow{a} w_{n+1}^1$. Then, by definition of $\sqsubseteq_c$, $u' \sqsubseteq_c w_{n+1}^1$. By definition of $\sim$, $\neg(w_n^1 \sim w_{n+1}^1)$. Therefore, the same argument can be repeated to show that $a \neq d_2$ and then that $a \neq d_i$ for each $i > 2$.

Therefore, by Lemma A.5(a), a is independent with all d_i . So, $a \in \text{en}(w'_l)$. Let $w_n^i \xrightarrow{a} w_{n+1}^i$ for $i < m$ and denote w_{n+1}^m by u . Then, by definition of $\sqsubseteq_c$, $w_{n+1}^i \sqsubseteq_c w_{n+1}^{i+1}$ for $i < m$. Notice that $u' = w_{n+1}^0$ and $u = w_{n+1}^m$. Therefore, by definition of $\sim$, $u' \sim u$.

If $a \in E(w'_l)$, then we are done. If not, then by Lemma A.8, there is a sequence of transitions $w'_l = v_0 \xrightarrow{b_1} v_1 \cdots \xrightarrow{b_k} v_k$ and the transition $v_k \xrightarrow{a} w''$ such that $v_i \sqsubseteq_c v_{i+1}$, $v_i \neq v_{i+1}$, $b_{i+1} \neq a$ for $i < k$, and $b_1 \dots b_k a \equiv_I a b_1 \dots b_k$. As before, using properties of $\sqsubseteq_c$ and $\sim$, we can show that $u \sim w''$. Therefore, by transitivity of $\sim$, $u' \sim w''$ and $v' \sim w''$, which ends the proof. $\square$

In order to prove that the relation $\sim$ satisfies also the property P2, the following auxiliary lemma is used to show that if at some state an infinite sequence of stuttering transitions starts, then also an infinite sequence of stuttering transitions starts at each state related by $\sim$.

Lemma A.12 *Let $w \sim v$. If there is an infinite sequence $w \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_i \Rightarrow \dots$, then there is an infinite sequence $v \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \dots \Rightarrow v_i \Rightarrow \dots$.*

Proof. Let $\sim_c^1 = \sqsubseteq_c \cup \sqsubseteq_c^T$ and $\sim_c^{k+1} = \sim_c^k; \sim_c$. The proof is by induction on $\text{length}(w \sim v)$, where $\text{length}(w \sim v) = \min\{k > 0 \mid w \sim_c^k v\}$.

For the base case $w \sim_c v$. If $w = v$, then we are done. So, assume otherwise. Then, according to Lemma A.5(a), either $v \rightarrow w$ or $w \rightarrow v$.

In case $v \rightarrow w$, this fact together with $w \sim v$ gives $v \Rightarrow w$. Concatenation this edge to $w \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_i \Rightarrow \dots$ provides the desired sequence.

To facilitate discussion of the second case where $w \Rightarrow v$, we make some transition labels explicit.

Let $w \xrightarrow{a} v$ and $w \xrightarrow{b_1} w_1 \xrightarrow{b_2} w_2 \xrightarrow{b_3} \dots \xrightarrow{b_i} w_i \xrightarrow{b_{i+1}} \dots$. Also assume w and v to have index 0 wherever applicable.

We build a sequence of transitions $v_{i-1} \xrightarrow{c_i} v_i$ ($i \geq 1$) starting at v , using the transitions $w_{i-1} \xrightarrow{b_i} w_i$ from the sequence starting at w .

As long as $a \neq b_i$, proceed as follows: start from v . If $a \neq b_1$ then, by Lemma A.5(a), $(a, b_1) \in I$ and hence $b_1 \in \text{en}(v)$. So, for some v_1 , $v \xrightarrow{b_1} v_1$. Note that if $v_1 = v$ this is a self-loop and therefore, by Definition A.4(a), also $v \sim_c v$ and thus $v \Rightarrow v$; hence the self-loop yields the desired sequence and we are done in this case. If $v_1 \neq v$, because $(a, b_1) \in I$, also $w_1 \xrightarrow{a} v_1$. By the Definition A.4 (c), then $w_1 \sim_c v_1$. Hence, by transitivity of $\sim$, $v \sim v_1$. This establishes $v \Rightarrow v_1$.

As mentioned, $w_1 \xrightarrow{a} v_1$, thus making the situation at v_1 with respect to w_1 exactly the same as at v with respect to w . This shows that, as long as $a \neq b_i$ the above construction can be applied iteratively.

Now when $a = b_j$ is encountered, this means that $v_{j-1} = w_j$. But that means that from v_{j-1} on one can take as transitions $v_{k-1} \xrightarrow{c_i} v_k$, $k \geq j$, just the transitions $w_{k-1} \xrightarrow{b_i} w_k$, yielding the desired sequence.

For the induction step let $w \sim_c^{k-1} z \sim_c c$ for some z . With respect to $w \sim_c^{k-1} z$, the induction hypothesis provides a sequence $z \xrightarrow{d_1} z_1 \xrightarrow{d_2} z_2 \xrightarrow{d_3} \dots \xrightarrow{d_i} z_i \xrightarrow{d_{i+1}} \dots$. In precisely the same manner as in the base case, a sequence $v \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \dots \Rightarrow v_i \Rightarrow \dots$ can now be constructed. This concludes the proof. $\square$

Lemma A.13 *The relation $\sim$ satisfies the property P2, i.e., if $v \sim w$ and there is an infinite stutter path $v = v_0 \Rightarrow v_1 \Rightarrow v_2 \dots$, then there is an infinite stutter path $w = w_0 \Rightarrow_E w_1 \Rightarrow_E \dots$.*

Proof. Let $v \sim w$ and $v \Rightarrow v_1 \Rightarrow v_2 \dots v_i \Rightarrow \dots$ be an infinite stutter sequence starting at v . By Lemma A.12, there is an infinite sequence $w \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_i \Rightarrow \dots$ starting at w . By applying Corollary A.10, we obtain an infinite stutter sequence $w = w'_0 \Rightarrow_E w'_1 \Rightarrow_E \dots w'_i \Rightarrow_E \dots$ $\square$

Finally, we prove that the algorithm produces a reduced state graph that is branching bisimilar to the full state graph, i.e., Theorem 5.2 from Section 5.2:

Proof. From the proofs of Lemmas A.5(b), A.7, and A.11 one immediately concludes that $\sqsubseteq$ is a branching bisimulation between M_P and M_r . $\square$

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt
H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen
L.A.M. Schoenmakers	Parallel Programs for the Recognition of <i>P</i> -invariant Segments, p. 16.
91/04	E. v.d. Sluis
A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse
P.J. de Bruin
P. Hoogendijk
G. Malcolm
E. Voermans
J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse
P.J. de Bruin
G.Malcolm
E.Voermans
J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts
K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J. Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic process creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottom-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for F _ω , p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ -calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.

93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxonomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.

94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π -conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Markov Decision Processes to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Rosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. Verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Klocks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Computational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.

94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β -Reduction in Church's $\lambda \rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$ CTL*, $\exists$ CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$ -free and W_4 -free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefănescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π -conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma R. Gerth W. Janssen S. Katz M. Poel C. Rump	J. Davies S. Graf B. Jonsson G. Lowe A. Pnueli J. Zwiers
94/45	G.J. Houben	Verifying Sequentially Consistent Memory, p. 160
94/46	R. Bloo F. Kamareddine R. Nederpelt	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/47	R. Bloo F. Kamareddine R. Nederpelt	The λ -cube with classes of terms modulo conversion, p. 16.
94/48	Mathematics of Program Construction Group	On Π -conversion in Type Theory, p. 12.
94/49	J.C.M. Baeten J.A. Bergstra	Fixed-Point Calculus, p. 11.
94/50	H. Geuvers	Process Algebra with Propositional Signals, p. 25.
94/51	T. Kloks D. Kratsch H. Müller	A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.
94/52	W. Penczek R. Kuiper	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
		Traces and Logic, p. 81