Compressing Transitions for Model Checking

Robert Kurshan[1], Vladimir Levin[2], and Hüsnü Yenigün[3]

[1] Cadence Design Systems, New Providence, NJ 07974
rkurshan@cadence.com
[2] Lucent Technologies, Mount Olive, NJ 07828
vlevin@lucent.com
[3] Sabancı University, İstanbul, Turkey
yenigun@sabanciuniv.edu

Abstract. An optimization technique is presented that compresses a chain of transitions into a single jump transition, thus making a model smaller prior to model checking. We give compression algorithms, together with conditions that allow such compressions to preserve next-time-free LTL. Experimental results are presented and discussed.

1 Introduction

In model checking a multi-component system, memory or/and time necessary to explore the system's state space may often grow too fast, even exponentially in the number of system's components. This is known as the state space explosion problem. Methods to fight it, often called "reductions", constitute one of the most important directions in formal verification and their effectiveness largely determines the size of systems manageable by model checking.

This paper suggests a reduction method that attempts to compress a sequence of transitions into a single transition, eliminating the interim states. A simple example of the compression method in the case of a sequential program is to replace the consecutive assignments $x := 1; x := x + 1$ with one assignment $x := 2$. Although this substitution might seem always possible, it is not so. Consider the property **always**$(x = 2)$: it holds on the compressed code, but fails on the original code. The paper formulates rules for correct compression and suggests simple reduction algorithms based on those rules.

When applied to a multi-process system with interleaving semantics, this compression method can augment partial order reduction. Partial order reduction gets its effect by ignoring redundant interleavings. Given two or more executions that only differ in the interleaving orders of their transitions (for example, a, b and b, a), it suffices to check the property on only one of those executions provided that its truth value can be guaranteed to be the same on any other one. The conditions which guarantee this kind of insensitivity to a particular interleaving order make up the heart of partial order reduction. In general ([16,4,13,10]), these conditions require that the transition selected to be executed be irrelevant to the property and independent (i.e. commutative) with all other transitions, and also, it cannot close a state-transition cycle. However, the very same conditions suggest that interim states between the transitions (e.g., between a and b) are also irrelevant to the property. Hence, there is no reason to execute the selected sequence (a, b) in steps, transition-by-transition. Instead, it can be executed as an atomic jump transition. Thus, the compression method completes the partial order reduction approach.

The compression is performed as a program transformation of the same sort required for static partial order reduction (SPOR) [10,7]. The model checker operates on the compressed program, which is equivalent (relative to the property being checked) to the original program.

SDLCheck [12] is a model checking tool developed for verification of SDL programs [6]. Given an SDL program and a property in a subset of Linear Temporal Logic (LTL) [15], they are both translated into S/R, the input language of the model checking tool COSPAN [9]. SDLCheck treats each SDL statement (an assignment, an output, etc.) as an atomic transition and reflects this view in the S/R model it produces: global states are generated before and after each SDL statement is executed. SDLCheck implements SPOR in the translation phase from SDL to S/R. Thus, the S/R model is optimized to have partial order reduction realized in the model checking phase. The combined tool, SDLCheck+COSPAN, utilizes all enhancements of COSPAN, including BDD-based symbolic verification. SDLCheck also implements the compression technique given in this paper.

SPOR is used to analyze the SDL system, and gather the information required for the compression.

For instance, it can detect if x is an important variable for the correctness of the verification, and if it is not, then the compression algorithm replaces the consecutive assignments $x := 1; x := x+1$ with $x := 2$. COSPAN, therefore, does not generate the interim state after executing $x := 1$, avoiding many possible interleavings that would be caused by the transitions of the other processes at this interim state. After transitions are compressed, SPOR is applied again, but this time, to the compressed S/R model.

2 Preliminaries

2.1 Modeling Programs by Transition Systems

We model a finite state program by *a transition system* $M = (S, \varsigma, T, \mathcal{P}, L)$ where S is a finite set of states, $\varsigma \in S$ is the initial state, T is a set of deterministic transitions, each transition $t \subseteq S \times S$, $\mathcal{P}$ is a finite set of propositional variables, and $L : S \mapsto 2^{\mathcal{P}}$ is an interpretation function: for every state s, $L(s)$ is the set of all propositional variables which are *true* at s.

For a transition t, the set $start(t)$ (the set $final(t)$) include all states that appears in a pair $(s_1, s_2) \in t$ as s_1 (respectively, s_2). Since transition t is assumed deterministic, we associate it with a function $t : start(t) \mapsto final(t)$ such that $t(s_1) = s_2$ iff $(s_1, s_2) \in t$. For a state s, the set of transitions *enabled* at s is

$enabled(s) = \{t \in T \mid s \in start(t)\}$. We assume for simplicity that for every state s, $enabled(s) \neq \emptyset$.

A *state-transition path* of M is an infinite or finite alternating sequence of states and transitions $\sigma = s_1, t_1, s_2, t_2, \ldots$ (ends in a state if it is finite) in which $t_i(s_i) = s_{i+1}$. Let $\sigma|_K$ be the projection of σ on a set K. An infinite state-transition path is called a *run* of M if it starts from the initial state ς.

To specify properties of a system M, we use $\mathrm{LTL_x}$, which is LTL without the next–time operator. M satisfies an $\mathrm{LTL_x}$ formula ϕ, denoted by $M \models \phi$, iff for every run σ of M, the proposition sequence $L(\sigma|_S)$ satisfies ϕ, denoted by $\sigma|_S \models \phi$, — we refer the reader to the literature for the exact definition (see, for example, [15]). Checking if $M \models \phi$ is performed by a model checking procedure.

2.2 Static Partial Order Reduction

Model checking algorithms construct reachable state space of M, by starting from the initial state and by successively exploring all the transitions enabled at a state. Partial order reduction techniques calculate, for each state s, a set of transitions called $stamper(s)$, which is a subset of the enabled transitions at s. Only the transitions in $stamper(s)$ are used to generate the next states of s, rather than using all the enabled transitions, thus omitting some runs of M. Stamper[1] sets must satisfy certain conditions, so that at least one run that does not satisfy ϕ (if such a run existed in M in the first place) is not omitted.

Partial order reduction follows the observation that the correctness of ϕ on a proposition sequence $\pi = p_1, p_2, \ldots$ (where $p_i \subseteq \mathcal{P}$) does not depend on how many times one and the same subset $\mathcal{P}$ appears as adjacent elements (hence stutters) in π. An integer $i \geq 2$ is said to be a *visible index* for π if $p_i \neq p_{i-1}$. Let $i_1 < i_2 < \ldots$ be all the visible indices of π. By taking the *fluent projection* of π defined as $fluent(\pi) = p_1, p_{i_1}, p_{i_2} \ldots$, we define *stuttering equivalence* relation on two proposition sequences π and π' as, $\pi \sim_{st} \pi'$ if $fluent(\pi) = fluent(\pi')$. Two systems $M = (S, \varsigma, T, \mathcal{P}, L)$ and $M' = (S', \varsigma', T', \mathcal{P}, L')$ with common $\mathcal{P}$ are said to be stuttering equivalent, denoted by $M \sim_{st} M'$, iff for each run σ of M, there exists a run σ' of M' such that $L(\sigma|_S) \sim_{st} L'(\sigma'|_{S'})$, and vice versa.

In [10,7], the stamper set conditions were modified into the static partial order reduction (SPOR) form, making it possible to select the stamper sets prior to actual model checking, and, hence, syntactically convert M into a (reduced) system M'. The following is a SPOR variation of the main partial order reduction theorem [13].

Theorem 1. [2]. *If SPOR reduces M to M', then $M \sim_{st} M'$.*

Since any $\mathrm{LTL_x}$ formula ϕ is *stuttering-closed* [11] (i.e. if $\pi \sim_{st} \pi'$ then $\pi \models \phi$ iff $\pi' \models \phi$), the following holds too.

[1] Different names have been used in the literature for different but similar conditions [16,4,13]. The term “stamper” is introduced in [14] to unify “stubborn”, “ample”, “persistent”, etc. sets of transitions.

[2] The proof is given in detail in [17].

Corollary 1. *For any LTL$_x$ formula ϕ, $M \models \phi$ iff $M' \models \phi'$.*

The implementation of SPOR algorithm is defined for a transition system M given in the form of process control flow graphs (supplied with additional information from the source program, which characterizes the data effects of transitions). The SPOR algorithm, in this setting, returns (identifies) a subset of stamper transitions $\mathcal{A} \subseteq T$, such that if $A \subseteq \mathcal{A}$ is a subset of enabled transitions of a single process, then $A = \mathit{stamper}(s)$ is a stamper set at s.

3 Compressing Links into Jump Transitions

Since the correctness of an LTL$_\mathrm{x}$ property actually depends on $fluent(L(\sigma|_S))$, the stuttering states, and the transitions outgoing from these states are not needed at all. We take this view, and try to identify such stuttering states, and the outgoing transitions from these states. We then combine a chain of stuttering transitions into a single *jump* transition, so that the intermediate stuttering states disappear. As an introductory example, assume that we have the following transitions $t_1 = \{(s_1, s_2)\}$ and $t_2 = \{(s_2, s_3)\}$. If s_1 is reachable, then so is s_2 and s_3, and a model checker has to generate all these states. If $L(s_2) = L(s_3)$, then t_1 and t_2 can be safely replaced with the transition $j_{(t_1,t_2)} = \{(s_1, s_3)\}$ in M, s_1 and s_3 will still be reachable, but s_2 is no longer reachable. This is the basic idea, however there are certain conditions under which such a transition compression can be done.

Below, we define the rules for compressing a transition system M. Also we claim the correctness of the rules by a number of lemmas. The proofs of the lemmas are given in [8] and omitted here due to the space limitations. We assume that a transition system M has been analyzed for static partial order reduction (hence, the stamper transitions are known), but has not been reduced yet. Applying compression rules to M may modify it into a different transition system M', which we can show is then stuttering-equivalent to M.

Definition 1. *Given two transitions $a, b \in T$ of a system $M = (S, \varsigma, T, \mathcal{P}, L)$, the pair (a, b) is called a* link *in M if the following conditions hold:*
(1) $final(a) = start(b)$
(2) $\{b\}$ is a stamper set at all the states in $start(b)$
(3) For any run σ, $\sigma|_{\{a,b\}}$ is either
(i) a finite sequence of the form $(ab)^\star$ or $(ab)^\star a$; or
(ii) an infinite sequence of the form $(ab)^\omega$
Given a link (a, b), we define the jump *transition relation as*
$$j_{(a,b)} = \{(s_a, s_b) | s_a \in start(a) \wedge s_b = b(a(s_a))\}$$

Intuitively, conditions 1 and 3 guarantee that we only produce a jump transition for a pair of transitions that follow each other, in the sense that b cannot get enabled before a on any run of M. They further impose that a is the only enabler for b. Removing the transitions a and b, and inserting the new transition $j_{(a,b)}$, will end up in missing some reachable states. Condition 2 guarantees that such states are either at the start or in the middle of some stuttering sequence of states, and thus safe to remove. The conditions of Definition 1 can further be relaxed, but this formulation is relatively easier to explain and implement.

Lemma 1. *Given a transition system $M = (S, \varsigma, T, \mathcal{P}, L)$ and a link (a, b) in M, let $M' = (S, \varsigma, T \cup \{j_{(a,b)}\} \setminus \{a, b\}, \mathcal{P}, L)$. Then, $M \sim_{st} M'$.*

Lemma 2. *Given a transition system $M = (S, \varsigma, T, \mathcal{P}, L)$, and a link (a, b) in M, let $M' = (S, \varsigma, T \cup \{j_{(a,b)}\} \setminus \{a, b\}, \mathcal{P}, L)$. Then if $\{a\}$ is stamper in M and $start(a) \cap final(b) = \emptyset$, then $\{j_{(a,b)}\}$ is stamper in M'.*

Lemma 3. *Given a transition system $M = (S, \varsigma, T, \mathcal{P}, L)$, and two transitions $t_1, t_2 \in T$, such that $start(t_1) \cap start(t_2) = \emptyset$, let $t_0 \notin T$ be defined as $t_0 = t_1 \cup t_2$. Then for $M' = (S, \varsigma, T \cup \{t_0\} \setminus \{t_1, t_2\}, \mathcal{P}, L)$, $M \sim_{st} M'$.*

Lemma 4. *Given a transition system $M = (S, \varsigma, T, \mathcal{P}, L)$, and a transition $t_0 \in T$, let $t_1, t_2 \notin T$ be two transitions such that t_1 and t_2 are a partitioning for t_0. Then for $M' = (S, \varsigma, T \cup \{t_1, t_2\} \setminus \{t_0\}, \mathcal{P}, L)$, $M \sim_{st} M'$.*

4 Compressing Transitions in a Multi-process System

A transition system $M = (S, \varsigma, T, \mathcal{P}, L)$ is a model for a multi-process program with an inter-process communication through message exchange, such as an SDL program [6]. The Algorithm 2 given below compresses local, input and output transitions with other local, input and output transitions, which all belong to one and the same process. The Algorithm 1 attempts to convert (to some extent) a pair of corresponding output and input transitions into a rendezvous transition. Thus, the interim and final transition systems which the algorithm produces may be associated with a multi-process program having both message exchange and two-process rendezvous synchronization. Below we explain specific features of such transition systems.

4.1 A Multi-process Transition System

In a multi–process transition system M, processes P are subsets of transitions: $P \subseteq 2^T$. Transitions T are partitioned into two sets, T^1 and T^2, that contain, respectively, "private" and "rendezvous" transitions. Each private transition belongs to only one process $p \in P$, which performs this transition (yet, a private transition may access variables of other processes too: consider, for example, an output action that updates a buffer variable of a receiving process). A rendezvous transition belongs to exactly two different processes $p, q \in P$. For each process p, we may produce (by translation from the source program) a graph G^p, called the *control flow graph* of p. The graph's nodes N are called *locations*. Multiple edges are allowed between two locations. However, there exist two mappings from the graph edges E into the locations N that give for each edge $e \in E$ its starting and final locations, $sloc(e)$ and $floc(e)$[3]. The edges in G^p are associated with the transitions p, i.e. there exists a one-to-one mapping $p \mapsto E$. We denote $\boldsymbol{t}$ the edge associated with transition t. If t is a rendezvous transition shared by processes p and q then it is associated with edge $\boldsymbol{t}^p$ in graph G^p and edge $\boldsymbol{t}^q$ in graph G^q. We will also apply edge specific notations (fanin, fanout, starting and final nodes, etc.) to transitions. In p, the following conditions are assumed to hold:

– All transitions in p which are enabled at the initial state ς have the common starting location, let loc^p_ς. No other transition may leave this location.

– For every run σ of M, $\sigma|_p$ corresponds to a (infinite or finite) path in G^p.

Lemma 5. *If a is the only fanin transition of a location and b is the only fanout transition of the same location then for a and b the condition 3 of Definition 1 is true.*

As explained above, the SPOR algorithm analyzes the process control flow graphs of system M (enriched with the data effects of transitions) and returns the set of stamper transitions $\mathcal{A} \subseteq T$. Thus, a *multi-process (transition) system M* can be represented by a tuple $(S, \varsigma, T = T^1 \cup T^2, \mathcal{P}, L, P, (G^p)_{p \in P}, \mathcal{A})$.

We also introduce a multi–process system with message exchange. In such systems, the set of private actions T^1 is further partitioned into subsets of local, output and input transitions, denoted respectively by, T^l, T^o and T^i. We assume that the output transitions $p^o \subseteq p$ of process p, are partitioned into subsets, called *output signals*. Respectively, the input transitions $q^i \subseteq q$ of process q, are partitioned into subsets called *input signals*. There exists a matching mapping from the set of (all) output signals to the set of (all) input signals such that at most one output signal of process p matches a given input signal of process q. This gives us a relation $\mathcal{C} \subseteq T^o \times T^i$ that connects every output transition of process p with one or more input transitions which all belong to process q. Relation $\mathcal{C}$ reflects, for example, a communication topology of SDL programs. Below, we refer to $\mathcal{C}$ as to *the signal connection* of system M.

4.2 Syntactic Manipulations with Transitions

In the implementation of the compression algorithms, we have to deal with syntactic representation of transitions of a system M. An example of syntax that fits well our purpose is Dijkstra guarded command $g(X) \hookrightarrow U := e(X)$, where guard $g(X)$ is a boolean expression, and $U := e(X)$ is a parallel assignment of values produced by expressions $e(X)$ to variables U. In the context of this syntax, states of the system M are given as valuations of program's variables X, among which we assume to have not only data variables D of the source program, but also the program counters of the processes, i.e. control variables, which model the control flow graphs of M. For example, if a is a local (input or output) transition of process p that starts in location c_1 and finishes in location c_2 then it can be

[3] Through introducing interim nodes, graph G^p can be explained as a traditional graph, where edges are pairs of nodes.

expressed by the command $p.c = c_1 \wedge g(D) \hookrightarrow V, p.c := e(D), c_2$, where $p.c$ is the program counter of process p and V is a list of (different) data variables. In a case of a local transition, all data variables used in this command must be local variables of process p. Additionally, in a case of an input or output action, one shared data variable is involved, which models the input buffer of the receiving process. A rendezvous transition we will deal with is represented by the following command:

$$p.c = c_1^p \wedge q.c = c_1^q \wedge g(D) \hookrightarrow V, p.c, q.c := e(D), c_2^p, c_2^q$$

where c_1^p, c_2^p and c_1^q, c_2^q are, respectively, locations in processes p and q.

When representing a transition a by a guarded command $g(X) \hookrightarrow U := e(X)$, $g(X)$ defines the set of states where a is enabled ($\mathit{start}(a)$). If $g(X) = g_1(X) \vee g_2(X)$ and $g_1(X) \wedge g_2(X) = \mathit{false}$, then the command can immediately be split into two commands as, $g_1(X) \hookrightarrow U := e(X)$ and $g_2(X) \hookrightarrow U := e(X)$, cf. Lemma 4. Merging two transitions, cf. Lemma 3, whose guards are disjoint is a bit more complicated, since, a variable, say, x may be updated in the both commands, say, by expression $f_1(X)$ in one and expression $f_2(X)$ in the other one. Yet, this is still simple: in the merged command, x will be updated by the conditional expression if $g_1(X)$ then $f_1(X)$ else $f_2(X)$ fi.

Merging commands (which represent transitions) as explained above can be utilized in an implementation of the algorithms given below. However, we need a more complicated splitting method. Consider two consecutive output transitions a, b of process p that send messages to two different processes q_1 and q_2. In general, we cannot make a jump transition $j_{(a,b)}$ that implements the both outputs in one step, i.e. simultaneously sends messages to q_1 and q_2, because the output b may be disabled (for example, the input buffer of process q_2 is full) when output a is enabled. However, if we split a into two outputs a_1 and a_2 such that a_1 is enabled iff output b is enabled, then we can make the jump transition $j_{(a_1,b)}$ (note that a copy of transition b must be preserved to work as a partner of transition a_2). Thus, we need a method to split transition a in such a way that $s \in \mathit{start}(a_1)$ iff $s \in \mathit{start}(a)$ and $a(s) \in \mathit{start}(b)$. Since, $\mathit{start}(a)$ is expressed by the guard of the command for transition a — let it be $g^a(X) \hookrightarrow U^a := e^a(X)$, the problem is only to express the predicate $a(s) \in \mathit{start}(b)$.

Fortunately, this problem can be approached through a well studied technique of manipulation with program actions and predicates. Namely, we refer to the *weakest liberal precondition* predicate transformer [3] $\mathit{wlp}(\alpha, \pi)$, where α is a program action and π is a predicate. In our context, α is the update function of transition a, i.e. the parallel assignment $U^a := e^a(X)$, and π stands for $\mathit{start}(b)$, which is already expressed by the guard $g^b(X)$ of the transition b command. Thus, the predicate $a(s) \in \mathit{start}(b)$ can be syntactically expressed as $\mathit{wlp}(U^a := e^a(X), g^b(X))$. This wlp expression can then be unfolded into an ordinary boolean expression through substitution of the expression $e^a(X)$ for occurrences of variables U^a in the guard $g^b(X)$.

A harder task would be to express the set of final states of transition a. However, in order, to obtain good compression it may suffice to under-approximate this set with a big enough set $F \subseteq \mathit{final}(a)$, which is expressible by a boolean expression in a chosen syntax. The first approximation set F can always be given by the expression $p.c = floc(a)$ provided that a belongs to process p.

4.3 Compression Algorithms

Next are two compression algorithms, presented in a pseudo-algorithmic form. These algorithms can be implemented, by embedding the steps into the traversal of the control flow graph. The algorithms perform syntactic modification of control flow graphs (and related components) of a multi-process system M.

Algorithm 1 deals with output/input pairs of (connected) transitions. It splits output o and input i in such a way that reveals the rendezvous sub-case of their interaction: namely, an output sub-transition o_1 makes a link with input sub-transition i_1, whereas the remaining private sub-transitions o_2 and i_2 take care of the asynchronous sub-case of the original output/input interaction.

Algorithm 1. Revealing a rendezvous component of an output/input pair.
Input: Multi–process system M augmented with a signal connection $\mathcal{C}$.
Action: Modify M as follows. In each pair of processes (p, q), $p, q \in P$, such that an output signal of p matches an input signal of q, for each pair of transitions $(o, i) \in \mathcal{C}$ such that i is a stamper transition, do the following:

1. Choose an approximation set $F \subseteq \mathit{final}(o)$, using available heuristics (for example, the set expressed by condition $p.c = floc(o)$).
2. Split input transition i into i_1 and i_2, and output transition o into o_1 and o_2 in such a way that $s \in \mathit{start}(i_1)$ iff $s \in \mathit{start}(i) \cap F$ and $s \in \mathit{start}(o_1)$ iff $o(s) \in \mathit{start}(i_1)$.
3. (Fact: the pair (o_1, i_1) is a link.) Make a jump transition $j = j_{(o_1,i_1)}$. (Note that j is a rendezvous transition shared by processes p and q.)
4. In process p, replace o by j, o_2. In process q, replace i by j, i_2.
5. In graph G^p, replace edge $\boldsymbol{o}$ by two new edges $\boldsymbol{o_2}, \boldsymbol{j^p}$ which both have the same starting and final locations as $\boldsymbol{o}$. In graph G^q, replace edge $\boldsymbol{i}$ by two new edges $\boldsymbol{i_2}, \boldsymbol{j^q}$ which both have the same starting and final locations as $\boldsymbol{i}$.
6. In the set of stamper transitions $\mathcal{A}$, replace i by i_2 and, if transition $o \in \mathcal{A}$ then also replace o by o_2, j. In relation $\mathcal{C}$ replace pair (o, i) by (o_2, i_2).

Return the modified system M' and the signal connection $\mathcal{C}'$. The sets T^l, T^o, T^i, T^2 are changed according to the changes in processes P and their control flow graphs $(G^p)_{p\in P}$.

A *hammock* is a set of $n > 1$ edges (or associated transitions) such that they all start at one location, let, s and finish at one location, let, f, not necessarily different from s. We say that an edge (a transition) b *follows* a if $sloc(b) = floc(a)$. A *chain* is a set of $n > 1$ edges (or associated transitions) such that (i) they form an acyclic path, i.e. may be ordered into a sequence $e_1, \dots e_n$ where e_{i+1} follows e_i and e_1 does not follow e_n, and (ii) each of them has either one fanin or one fanout edge. A hammock (chain) is maximal if it would not be a hammock (respectively, chain) any more if any other edge (transition) is included into it. Below, whenever a hammock (chain) of edges $\{e_1, \ldots e_n\}$ is said to be replaced by a new edge e, it is meant that $sloc(e) = sloc(e_1)$ and $floc(e) = floc(e_n)$ (if the edges form a chain, e_1 and e_n are, respectively, the first and last elements).

Algorithm 2 compresses chains and hammocks in each process of system M. For simplicity, it only deals with private transitions and ignores rendezvous transitions. Algorithm 2 utilizes Procedure 1 that basically (though not exclusively) targets output transitions, for which making a link with a preceeding transition, say, a, is a problem: the set of final states of transition a does not necessarily (in a fact, almost never) coincides with the set of starting states of the following output transition. As an example, consider two consequtive outputs.

Algorithm 2. Compressing hammocks and chains.
Input: A multi-process system M.
Variables: C, H and R will keep temporary sets of (structures of) transitions of a current process p. C is designated for chains, H for hammocks, R contains those remaining transitions of process p that the algorithm has not rejected to deal with yet. Some pairs of transitions in chains of C can be marked as *bad*.

Action: Modify each process $p \in P$ and graph G^p, and the $\mathcal{A}$ as follows.

1. Initialize: $C :=$ the maximal chains of $p \setminus T^2$, $H :=$ the maximal hammocks of $p \setminus T^2$, $R := p \setminus (T^2 \cup \bigcup C \cup \bigcup H)$.
2. If no chain in C contains transitions a, b s.t. b follows a and (a, b) is not *bad* then do the following. If $H = \emptyset$ then select the next process in P and go to step 1. Otherwise, go to step 6.
3. Otherwise, pick up such a pair (a, b) in a chain of C, which is not marked *bad* and b follows a. If (a, b) is not a link then perform Procedure 1 given below and then go to step 2.
4. ((a, b) is a link.) In process p replace a and b by a jump transition $j = j_{(a,b)}$. In graph G^p replace the chain $\{\boldsymbol{a}, \boldsymbol{b}\}$ by a new edge $\boldsymbol{j}$. Remove b from $\mathcal{A}$. If $a \in \mathcal{A}$ then replace a by j. Remove a and b from (a chain in) C.
5. If any fanin or fanout transition of j belongs to a chain in C then include j into this chain. Otherwise, include j into a hammock $h \in H$ such that $\{j\} \cup h$ is a hammock. If no such hammock is found, then, check if j makes a hammock with a transition $r \in R$. If it does, include the hammock $\{j, r\}$ into H and remove r from R. Otherwise, include j into R. Go to step 2.
6. If $H = \emptyset$ then go to step 2. Otherwise, choose a hammock, let, $\{a_1, \ldots a_n\} \in H$ and remove it from H. If the hammock cannot be merged into a single transition a (cf. Lemma 4), then repeat this step 6.
7. In process p replace $a_1, \ldots a_n$ by a. In graph G^p replace the hammock $\{\boldsymbol{a_1}, \ldots \boldsymbol{a_n}\}$ by a new edge $\boldsymbol{a}$. If all $a_1, \ldots a_n \in \mathcal{A}$ then replace all of them by a; otherwice, $\mathcal{A} := \mathcal{A} \setminus \{a_1, \ldots a_n\}$.
8. If a makes a chain with transitions in the set $\{a\} \cup R \cup \bigcup C$ then form the maximal chain (in this set) that contains a and include it into C. Chains

from C are included into this new chain with their *bad* markings (if any). Remove from R the transitions which participate in this new chain. Go to step 6.

9. If a makes a hammock $\{a, r\}$ with transition $r \in R$ then include it into H and remove r from R. Otherwise, include a into R. Go to step 6.

Return the modified system M'

In Algorithm 2, each process p is modified within two loops that perform in one or more turns, one after another. The first loop (steps from 2 through 5 and Procedure 1) attempts to compress chains in C as much as possible. When nothing remains to compress in C, the second loop (steps from 6 through 9) attempts to merge hammocks in H as much as possible. Each of the loops may generate a new hammock for H. The first loop may also include a new transition into an existing hammock in H, whereas the second loop may generate a new chain for C with a transition in R. When nothing is left to compress (merge) in both C and H, modification of process p finishes and the next process is selected.

Procedure 1. (Performed for pair (a, b), if it is not a link.)

1. If $b \notin \mathcal{A}$ or a or b makes a link with a transition in the same chain of C that a, b belong to then go to step 2. Otherwise, go to step 3.
2. In C, mark the pair (a, b) as *bad*. If a follows c and (c, a) is *bad*, then remove a from C. If d follows b and (b, d) is *bad*, then remove b from C. Return to continue step 3 of Algorithm 2.
3. Split transition a into a_1, a_2 s.t. state $s \in \mathit{start}(a_1)$ iff $s \in \mathit{start}(a)$ and $a(s) \in \mathit{start}(b)$. In process p, replace a by a_1, a_2. In graph G^p, let $\mathit{sloc}(\boldsymbol{a}) = \lambda$ and $\mathit{floc}(\boldsymbol{a}) = \lambda_1$; then, make a new location λ_2 and replace the edge $\boldsymbol{a}$ by two new edges $\boldsymbol{a_1}, \boldsymbol{a_2}$ s.t. the both have λ as the starting location, whereas $\mathit{floc}(\boldsymbol{a_1}) = \lambda_1$ and $\mathit{floc}(\boldsymbol{a_2}) = \lambda_2$. Associate the edges $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ with transitions a_1 and a_2, respectively. (Fact: (a_1, b) is a link. Note that in G^p edge $\boldsymbol{a_2}$ is not connected yet to edge $\boldsymbol{b}$.)
4. Make the jump transition $j = j_{(a_1,b)}$. In process p replace a_1 by j. In graph G^p replace chain $\{\boldsymbol{a_1}, \boldsymbol{b}\}$ by a new edge $\boldsymbol{j}$, and associate transition b with a new edge $\boldsymbol{b}'$ s.t. $\mathit{sloc}(\boldsymbol{b}') = \lambda_2$ and $\mathit{floc}(\boldsymbol{b}') = \mathit{floc}(\boldsymbol{b})$. (Thus, from now on, the new location λ_2 effectively replaces the old λ_1 — so that the new edge $\boldsymbol{b}'$ follows $\boldsymbol{a_2}$). If $a \in \mathcal{A}$ then replace it by j, a_2. Remove a, b from (a chain in) C. Return to continue step 3 of Algorithm 2.

Theorem 2. *Given a multi-process transition system M (with signal connection $\mathcal{C}$), Algorithm 2 (respectively Algorithm 1) returns a transition system M' such that $M \sim_{st} M'$.*

The proof of this theorem follows from the two facts: (i) each complete sequence of steps of the algorithm (Algorithm 2 or Algorithm 1, respectively) produces an intermediate system $\widetilde{M}$ such that $\widetilde{M} \sim_{st} M$, and (ii) the algorithm terminates. In Algorithm 1, a complete sequence of steps include all steps that deal with one output/input pair (o, i). In Algorithm 2, there are two types of complete step sequences: one deals with a pair of transitions (a, b) taken from a chain in C and the other one deals with a hammock taken from H.

5 Implementation and Experimental Results

Experimental results are presented in this section. Currently, revealing a rendezvous component of output/input transition pairs is not implemented. The table below summarizes our results.

Example	Experiments	
Leader(6)	Explicit, SPOR 1548 sec, 380M	Explicit, SPOR, Compression 1080sec, 204M
Sort(12)	Symbolic, SPOR 87 sec, 20M	Symbolic, SPOR, Compression 54 sec, 9.4M
HW/SW Elevator	–	Symbolic+SPOR+Compression 14403 sec, 415M
Triple Ring	Symbolic, SPOR 21 sec, 28M	Symbolic, SPOR, Compression 18 sec, 24M

A leader election protocol with 6 processes could only be verified using an explicit search, since the symbolic verification ran out of memory. The compressed system could be verified in a shorter time and using less memory than the uncompressed system. The sort example (a chain of SDL processes that runs as a parallel sorting algorithm), on the contrary, could not be verified (even with 7 processes) using explicit search after waiting 24 hours. However, the compression technique in this case also reduced the amount of time and memory required in symbolic search. As a HW/SW co-design example, we tried a simple elevator system as given in [10]. This example could only be verified using symbolic search, SPOR and compression all at the same time. It ran out of memory when compression was not used.

Fourth example is a ring consisting of three processes P, Q and R such that P sends messages to Q, Q to R and R to P. The property we checked is invisible in processes P and Q. Using SDLCheck with SPOR and no compression, we generated S/R code for this program. Then we manually modified this S/R code in such a way that one of the output/input pairs, namely, between P and Q, is split into the rendezvous action and non-rendezvous output and input (respectively, in P and Q.) The result of this small optimization, which is additional to SPOR, is not just 15% improvement. The more important is what it indeed demonstrates. Since, the cycle of inter-process communication [7] has been deliberately broken in process R (in SDLCheck, this is controllable), all local actions in P and Q appear stamper.

As a result, they are enforced to execute prior to any input and output actions. Therefore, neither of the two non-rendezvous actions (the output from P to Q and the corresponding input in Q), which remain after extraction of the rendezvous component, ever becomes enabled. In other words, communication between P and Q could be implemented as fully rendezvous, for this particular property. Remarkably, COSPAN recognizes and utilizes this situation. Before actual model checking, COSPAN applies the localization reduction, which eliminates state variables that fall out of the influence cone of the property. So, we can observe that this reduction indeed completely eliminates the buffer of the process Q. We emphasize that no such effect can be observed using only (any) one or even (any) two of the three reductions we applied in the sequence: compression of the output/input pair (based on SPOR), SPOR itself and (finally) localization reduction.

As the experimental results suggest, the transition compression may help reducing the required time and memory further, in addition to other relief techniques such as SPOR, localization reduction, symbolic verification, etc.

6 Conclusion and Related Work

A formal basis for the compression of transitions in a transition system is presented. The formalization of the method at the transition system level provides a more general framework than previous work. Recently, a similar approach was outlined in [5] and then elaborated in [18] that compresses transitions of one process. However, even within a single process, that approach does not compress consecutive output transitions. We show that consecutive transitions may be compressed, provided that the conditions of Definition 1 are satisfied. The other method does not suggest any technique for extracting a rendezvous component from asynchronous communication, as we do. Although the previous work states that their method is closely related to partial order reduction, the exact relationship is not made explicit. We show this relationship explicitly.

The compression method presented, may also make some of the variables in the system automatically removed, as in the case of the triple ring example. Live variable analysis, given in [1] can be considered as a similar approach. However, live variable analysis technique identifies the unused assignments to variables, and then removes these assignments from the model. Note that, in the triple ring example, the buffer of the process is actually used effectively in the original system. It becomes unused only after compressing the output and input transitions into a jump transition. Therefore, a live variable analysis on the original system would not recognize the process' buffer as a dead variable.

An observation that partial order reduction may be improved by jumping via a sequence of stamper (ample) transitions, was reported in [2], which suggests a different method (called a *leap*) that simultaneously executes transitions from all (current) stamper sets, rather than from one such set. This means merging independent concurrent transitions from different processes, hence, contrasts to compressing consequtive transitions from one or several processes. The leap and compression methods can be combined together, and SDLCheck implements this. However, our experiments do not show that the leap method consistently improves performance. For example, even after waiting eight hours, we could not verify the sorting algorithm with 12 processes when we used a compressed S/R model that realizes the leap technique. Apparently, such performance decline is due to the intrinsic complexity of the leap mechanism, which must deal (during the model checking, not statically) with the entire set of all current stamper sets.

Our method can benefit directly from any progress in the field of partial order reduction theory. Relaxing visibility conditions of transitions [14], for example, can directly be incorporated into our tool.

References

1. Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. State space reduction based on live variables analysis. In *Static Analysis Symposium*, pages 164–178, Venezia, Italy, 1999. 580
2. H. Van der Schoot and H. Ural. An improvement on partial order model checking with ample sets. Technical Report TR-96-11, Univ. of Ottawa, Canada, 1996. 580
3. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of ACM*, 18(8):453–457, August 1975. 575
4. P. Godefroid. *Partial–Order Methods for the Verification of Concurrent Systems: An Approach to the State–Explosion Problem*. PhD thesis, University of Liège, Liège, Belgium, November 1994. 569, 571
5. G. J. Holzmann. The engineering of a model checker: The gnu i-protocol case study revised. In *6th Workshop on SPIN*, LNCS 1680, 1999. 580
6. ITU–T, Geneva. *Functional Specification and Description Language (SDL), Recommendation Z.100*, March 1993. 570, 573
7. R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Combining hardware and software verification tecniques. *Formal Methods in System Design*. accepted for publication. 570, 571, 579
8. R. P. Kurshan, V. Levin, and H. Yenigün. Compressing transitions for model checking. Technical report, Bell Labs, Lucent Technologies, 2002. 572
9. R. P. Kurshan. *Computer–Aided Verification of Coordinating Processes: The Automata–Theoretic Approach*. Princeton University Press, 1994. 570
10. R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In 4^{th} *International Conference Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 345–357, Portugal, 1998. 569, 570, 571, 579
11. L. Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Information Processing*, pages 657–668, Paris, September 1983. Elsevier Science Publishers. 571
12. V. Levin and H. Yenigun. SDLCheck : A model checking tool. In *13th CAV*, France, 2001. 570
13. D. Peled. All from one, one for all – on model checking using representatives. In *5th CAV*, LNCS 697, pages 409–423, Crete, June 1993. Springer–Verlag. 569, 571
14. D. Peled, A. Valmari, and I. Kokkarinen. Relaxed visibility enhances partial order reduction. *Formal Methods in System Design*, 19(3), November 2001. 571, 581
15. A. Pnueli. The temporal logics of programs. In *18th Annual IEEE–CS Symposium on Foundations of Computer Science*, pages 46–57, Cambridge, 1977. 570, 571
16. A. Valmari. A stubborn attack on state explosion. In *2nd CAV*, LNCS 531, pages 156–165, Rutgers, June 1990. Springer–Verlag. 569, 571
17. H. Yenigün. *Static Partial Order Reduction and Model Checking of HW/SW Co–Design Systems*. PhD thesis, Middle East Technical University, Turkey, 2000. 571

18. K. Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion – Israel Institute of Technology, Haifa, Israel, 2000. 580