10-2008

Model checking CSP revisited: Introducing a process analysis toolkit

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Jin Song DONG

Citation

SUN, Jun; LIU, Yang; and DONG, Jin Song. Model checking CSP revisited: Introducing a process analysis toolkit. (2008). *Proceedings of Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15*. 307-322.
Available at: https://ink.library.smu.edu.sg/sis_research/5050

Model Checking CSP Revisited: Introducing a Process Analysis Toolkit

Jun Sun, Yang Liu, and Jin Song Dong

School of Computing,
National University of Singapore
{dongjs,liuyang,sunj}@comp.nus.edu.sg

Abstract. FDR, initially introduced decades ago, is the *de facto* analyzer for Communicating Sequential Processes (CSP). Model checking techniques have been evolved rapidly since then. This paper describes PAT, i.e., a *p*rocess *a*nalysis *t*oolkit which complements FDR in several aspects. PAT is designed to analyze event-based compositional system models specified using CSP as well as shared variables and asynchronous message passing. It supports automated refinement checking, model checking of LTL extended with events, etc. In this paper, we highlight how partial order reduction is applied to improve refinement checking in PAT. Experiment results show that PAT outperforms FDR in some cases.

1 Introduction

Hoare's classic Communicating Sequential Processes (CSP [7]) has been a rather successful event-based modeling language for decades. Theoretical development on CSP has advanced formal methods in many ways. Its distinguishable features like alphabetized parallel composition have proven to be useful in modeling a wide range of systems.

FDR (Failures-Divergence Refinement) [12] is the *de facto* analyzer for CSP, which has been successfully applied in various domains. Based on the model checking algorithm presented in [12] and later improved with other reduction techniques presented in [15], FDR is capable of handling large systems. Nonetheless, since FDR was initially introduced, model checking techniques have evolved a lot in the last two decades. Quite a number of effective reduction methods have been proposed which greatly enlarge the size the systems that can be handled. Some noticeable ones include partial order reduction, symmetry reduction, predicate abstraction, etc. Moreover, verification based on temporal logic properties has gathered much attention. In this work, we present a process analysis toolkit named PAT[1], which is designed to incorporate advanced model checking techniques to analyze event-based compositional system models. PAT complements FDR in a number of ways. The following is a list of PAT's main functionalities.

– refinement checking. Refinement checking in FDR has been proved useful [16,14]. Given a process representing the implementation and another representing the specification, PAT (like FDR) automatically verifies whether there is a refinement relationship between them. Refinement checking in FDR replies on normalizing the

[1] Available at http://www.comp.nus.edu.sg/~liuyang/pat

specification before hand, which has proven to be very effective for some systems [15]. Nonetheless, normalization is computational expensive in general. In PAT, an alternative approach which brings normalization on-the-fly is adopted.

– temporal logic based model checking. An LTL model checker is embedded in PAT. Users are allowed to specify properties using standard LTL (extended with events, refer to Section 4.4). An on-the-fly explicit model checking algorithm is then used to produce counterexamples (if there is) or to conclude true.
– simulation. PAT supports various ways of system simulation, e.g., random simulation, user-guided step-by-step simulation, system graph generation, etc.

Besides, dedicated algorithms have been developed to analyze specialized properties, e.g., deadlock-freeness, divergence-freeness, invariants, LTL properties under weak or strong fairness assumptions, etc. In order to handle systems with large number of states, partial order reduction has been realized in PAT to enhance LTL model checking, refinement checking as well as other dedicated verification algorithms. Previous works on partial order reduction have only been applied to refinement checking in limited ways (refer to *diamond reduction* presented in [15]). Based on previous theoretical works [18,19], novel reduction techniques have been developed in PAT to enhance refinement checking. In this paper, the algorithms for partial order reduction as well as its soundness are discussed in detail. Other features of PAT are briefly introduced.

The remainder of the paper is organized as follows. Section 2 reviews PAT's input language. We briefly introduce FDR and its refinement checking in Section 3. Section 4 presents FDR's refinement checking algorithm and the refined one with partial order reduction. Section 5 concludes the paper and reviews related works and future works.

2 Communicating Sequential Processes with Extensions

In this section, we introduce our extended CSP language, i.e., its operational semantics as well as the definitions of the trace refinement, stable failure refinement and failure/divergence refinement.

Definition 1 (Process). *A process is defined as follows,*

$$\begin{aligned} P ::= {} & Stop \mid Skip \mid e \rightarrow P \mid c!exp \rightarrow P \mid c?x \rightarrow P \\ \mid {} & P;\ Q \mid P \,\Box\, Q \mid P \sqcap Q \mid P \lhd b \rhd Q \mid [b] \bullet P \mid P \bigtriangleup Q \mid P \setminus X \\ \mid {} & P_1 \parallel P_2 \parallel \cdots \parallel P_n \mid P_1 \mid\mid\mid P_2 \mid\mid\mid \cdots \mid\mid\mid P_n \end{aligned}$$

where c is a channel with bounded buffer size, b is a Boolean expression, X is a set of events, exp is an expression and e is an event. Event e may be a simple abstract event or a compound one (e.g., $e.x.y$) with an optional sequence of assignments.

We support most of the CSP language constructs. The two most noticeable extensions are shared variables and asynchronous message passing. It has long been known (see [7] and [13], for example) that one can model a variable as a process parallel to the one that uses it. The user processes then read from, or write to, the variable by CSP communication. Similarly, one can model a (bounded) message channel as a process. Nonetheless, these 'syntactic sugars' are mostly welcomed. Supporting them explicitly allows us to avoid generating multiple parallel processes and hence verify more efficiently in some cases[2]. Most of the operators (as explained in [7]) are well-understood. We briefly review the extended ones as well as ones whose semantics needs clarification. The operational semantics is presented in Appendix A.

Let Σ denote the set of all visible events and τ denote an invisible action. Let Σ^* be the set of finite traces. Let Σ_τ be $\Sigma \cup \{\tau\}$. Event prefixing $e \rightarrow P$ performs e and behaves as P afterward. If e is attached with assignments, the valuation of the global variables is updated accordingly. For simplicity, assignments are restricted to update only global variables.$Skip = \checkmark \rightarrow Stop$ where $\checkmark$ is the termination event. Sequential composition, $P_1;\ P_2$, behaves as P_1 until its termination and then behaves as P_2. An external choice is solved only by the engagement of an visible event. A choice depending on the truth value of a Boolean expression is written as $P_1 \lhd b \rhd P_2$. If b is true, the process behaves as P_1, otherwise P_2. State guard $[b] \bullet P$ is blocked until b is true and then proceeds as P. $P_1 \bigtriangleup P_2$ behaves as P_1 until the first visible event of P_2 is engaged, then P_1 is interrupted and P_2 takes control. Process $P \setminus X$ hides all occurrences of events in X. One of the key features of CSP is the alphabetized multi-threaded parallel composition. Let αP be the alphabet of P which excludes τ and $\checkmark$. In PAT, alphabets can be manually set or derived from the events constituting the process expression. Parallel composition of processes is written as $P_1 \parallel P_2 \parallel \cdots \parallel P_n$, where shared events must be synchronized by all processes whose alphabet contains the event. The indexed interleaving is written as $P_1 \mid\mid\mid P_2 \mid\mid\mid \cdots \mid\mid\mid P_n$, in which all processes run independently except communication through shared variables and message channel (and synchronization upon termination, i.e., rule ter). Recursion is allowed by process referencing. The semantics of recursion is defined as Tarski's weakest fixed-point. Processes may be parameterized (see examples later).

For simplicity, we focus on the operational semantics in this work, i.e., the semantics of a model is associated with a labeled transition system. Due to the global variables and channels, configuration of a given system is composed of three parts (P, V, C) where P is the current process expression, V is the current valuation of the global variables which is a set of mappings from a name to a value, and C is the current status of the channels which is a set of mappings from a channel name to a sequence of items in the channel. A transition is of the form $(P, V, C) \xrightarrow{e} (P', V', C')$, which means (P, V, C) evolves to (P', V', C') by performing event e.

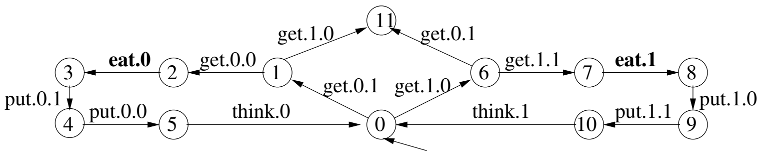
Example 1. The following models the classic dining philosophers [7],

$$\begin{aligned}
Phil(i) &= get.i.(i+1)\%N \rightarrow get.i.i \rightarrow eat.i \rightarrow \\
&\qquad \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \rightarrow think.i \rightarrow Phil(i) \\
Fork(i) &= get.i.i \rightarrow put.i.i \rightarrow Fork(i) \;\Box \\
&\qquad get.(i-1)\%N.i \rightarrow put.(i-1)\%N.i \rightarrow Fork(i) \\
Pair(i) &= (Phil(i) \parallel Fork(i)) \setminus \{get.i.i, put.i.i, think.i\} \\
College &= (\big\|_{i=0}^{N-1} Pair(i)) \setminus \bigcup_{i=0}^{N-1} \{get.i.(i+1)\%N, put.i.(i+1)\%N\}
\end{aligned}$$

where N is a global constant (i.e., the number of philosophers), $get.i.j$ ($put.i.j$) is the action of the i-th philosopher picking up (putting down) the j-th fork and fc is a global

[2] Refer to [13] for cases in which this is not true in the world of FDR.

variable recording the amount of food that has been consumed. The system is composed of N philosopher-fork-pairs running in parallel. The following is the transition system of $College$ with $N = 2$. All events except the bolded ones are invisible. □



3 FDR and Refinement Checking

Failures-Divergence Refinement (FDR [12]) is a well-established model checker for CSP. Different from temporal logic based model checking, using FDR, safety, liveness and combination properties can be verified by showing a refinement relation from the CSP model of the system to a CSP process capturing the properties. In addition, FDR verifies whether a process is deadlock-free or not. In the following, we review the notion of different refinement/equivalence relationship in terms of labeled transition systems.

Definition 2 (Labeled Transition System). *An LTS is 3-tuple $L = (S, init, T)$ where S is a set of states, $init \in S$ is the initial state and $T : S \times \Sigma_\tau \times S$ is a labeled transition relation. Let s, s' be members of S.*

- $s \overset{e_1, e_2, \cdots, e_n}{\rightarrow} s'$ *if and only if there exists $s_0, \cdots, s_n \in S$ such that for all $0 \le i \le n$ such that $s_i \overset{e_i}{\rightarrow} s_{i+1}$ and $s_0 = s \wedge s_n = s'$.*
- *Let $tr : \Sigma^*$ be a sequence of visible events. $s \overset{tr}{\Rightarrow} s'$ if and only if there exists $e_1, e_2, \cdots, e_n$ such that $s \overset{e_1, e_2, \cdots, e_n}{\rightarrow} s'$ and $tr = \langle e_1, e_2, \cdots, e_n \rangle \upharpoonright \{\tau\}$ is the trace with invisible actions filtered.*
- $s \rightarrow^* s'$ *if and only if there exists $e_1, \cdots, e_n$ such that $s \overset{e_1, e_2, \cdots, e_n}{\rightarrow} s'$. In particular, $s \rightarrow^* s$.*
- $enabled(s) = \{e : \Sigma_\tau \mid \exists s' \bullet s \overset{e}{\rightarrow} s'\}$. *A state is stable if and only if $\tau \notin enabled(S)$.*
- $mrefusal(S) = \Sigma \setminus enabled(S)$ *is the maximum refusal set, i.e., the maximum set of events which can be refused.*
- $s \overset{\tau*}{\rightarrow} s'$ *if and only if $s \overset{\tau, \cdots, \tau}{\rightarrow} s'$. $\tau^*(s) = \{s' : S \mid s \overset{\tau*}{\rightarrow} s'\}$ is the set of stable states reachable from s by performing zero or more τ transitions.*
- *A state is a divergence state $div(s)$ if and only if $\tau \in enabled(s) \wedge s \in \tau^*(s)$.*

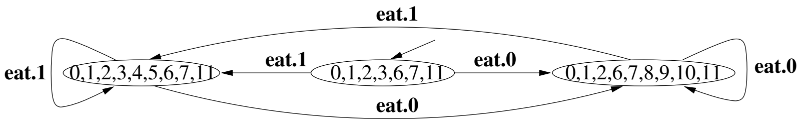
The set of traces of L is $traces(L) = \{tr : \Sigma^* \mid \exists s' : S \bullet init \overset{tr}{\Rightarrow} s'\}$. The set of divergence traces of L, written as $divergence(L)$, is $\{tr : \Sigma^* \mid \exists tr' \bullet tr'$ is a prefix of $tr \wedge \exists s : S \bullet init \overset{tr'}{\rightarrow} s \wedge div(s)\}$. Note that if some prefix of a given trace is a divergence trace, the given trace is too. The set of failures of L, written as $failures(L)$, is $\{(tr, X) : \Sigma^* \times 2^\Sigma \mid \exists s : S \bullet init \overset{tr}{\rightarrow} s \wedge X \subseteq \Sigma \setminus enabled(s)\} \cup \{(tr, \Sigma) : \Sigma^* \times 2^\Sigma \mid tr \in divergence(L)\}$. Note that the system state reached by a divergence state may refuse all events. Given a model composed of a process P and a valuation V and a set of channels C, we may construct an LTS $(S, init, T)$ where $S = \{s \mid (P, V, C) \rightarrow^* s\}$, $init = (P, V, C)$ and $T = \{(s_1, e, s_2) : S \times \Sigma_\tau \times S \mid s_1 \xrightarrow{e} s_2\}$ using the operational semantics. However, S can be infinite due to several reasons. The first one is that the variables may have infinite domains or the channels may have infinite buffer size. We require (syntactically) that the sizes of the domains and buffers are bounded. The second is that P may allow unbounded recursion or replication, e.g., $P = (a \rightarrow P;\ c \rightarrow Skip) \ \Box\ b \rightarrow Skip$ or $P = a \rightarrow P \ ||| \ P$. In this paper, we focus on LTSs with finite number of states for practical reasons. The following defines refinement and equivalence.

Definition 3 (Refinement and Equivalence). *Let $L_{im} = (S_{im}, init_{im}, T_{im})$ be an LTS representing an implementation. Let $L_{sp} = (S_{sp}, init_{sp}, T_{sp})$ be an LTS representing a specification. L_{im} refines L_{sp} in the trace semantics, written as $L_{im} \sqsupseteq_T L_{sp}$, if and only if $traces(L_{im}) \subseteq traces(L_{sp})$. L_{im} refines L_{sp} in the stable failures semantics, written as $L_{im} \sqsupseteq_F L_{sp}$, if and only if $failures(L_{im}) \subseteq failures(L_{sp})$. L_{im} refines L_{sp} in the failures/divergence semantics, written as $L_{im} \sqsupseteq_D L_{sp}$, if and only if $failures(L_{im}) \subseteq failures(L_{sp})$ and $divergence(L_{im}) \subseteq divergence(L_{sp})$. L_{im} equals L_{sp} in the trace (stable failures/failures divergence) semantics if and only if they refine each other in the respective semantics.*

Different refinement relationship can be used to establish different properties. Safety can be verified by showing a trace refinement relationship. Combination of safety and liveness is verified by showing a stable failures refinement relationship if the system is divergence-free or otherwise by showing a failures/divergences refinement relationship. The readers shall refer to [14] for a discussion on the expressiveness of CSP refinement. In the following, we write $Im \sqsupseteq Sp$ to mean $L_{Im} \sqsupseteq L_{Sp}$ whenever it will not cause confusion. Internally, equivalence relationships may be used to simplify process expressions, e.g., $P \ \Box\ P$ is replaced by P for simplicity.

Example 2. Assume that the following process is used to capture the property for the dining philosophers: $Prop \ \widehat{=} \|_{i=0}^{N-1} \ Eat(i)$ where $Eat(i) = eat.i \rightarrow Eat(i)$. It can be shown that $College$ trace-refines $Prop$ (given a particular N). Informally speaking, that means it is possible for each philosopher to eat, i.e., $\{eat.0, \cdots, eat.(N-1)\}^*$ are traces of $College$. In order to show that it is *always* possible for him/her to eat, we need to establish $College \sqsupseteq_F Prop$, which is not true, i.e., assume $N = 2$, $(\langle get.0.1, get.1.0\rangle, \{eat.0, eat.1\})$ is in $failures(College)$ but not $failures(Prop)$. $\Box$

In order to check refinement, every state of the implementation reachable from the initial state via some trace must be compared with every state of the specification reachable via the same trace. There may be many such states in the specification due to nondeterminism. In FDR, the specification is firstly normalized so that there is exactly one state corresponding to each possible trace. A state in the normalized LTS is a set of states which can be reached by the same trace from the initial state. For instance, The following shows the normalized LTS of the one presented in Example 1.



Definition 4 (Normalized LTS). *Let $(S, init, T)$ be an LTS. The normalized LTS is $(NS, Ninit, NT)$ where NS are subsets of S, $Ninit = \tau^*(init)$ and $NT = \{(P, e, Q) \mid Q = \{v : S \mid \exists\, v_1 : P \bullet v_1 \xrightarrow{e} v_2 \wedge v \in \tau^*(v_2)\}\}$. Given a normalized state $s \in NS$,*

- *$enabled(s)$ is $\bigcup_{x \in s} enabled(x)$,*
- *$mrefusal(s)$ is $\{mrefusal(x) \mid x \in s\}$, which is a set of maximum refusal sets,*
- *$div(s)$ is true if and only if there exists $x \in s$ such that $div(x)$ is true.*

Given an LTS constructed from a process, the normalized LTS corresponds the normalized process. A state in the normalized LTS groups a set of states in the original LTS which are all connected by τ-transitions. Given a trace, exactly one state in the normalized LTS is reached. FDR then traverses through every reachable states of the implementation and compare them with the corresponding normalized states in the specification (refer to the algorithm presented in [12]).

4 Verification

This section is devoted to algorithms for refinement checking. We start with reviewing a slightly modified on-the-fly checking algorithm based on the one implemented in FDR and then improve it with partial order reduction. Lastly, we review an alternative approach for verification that has been implemented in PAT, i.e., LTL-based verification.

4.1 On-the-Fly Refinement Checking Algorithm

Let $Spec$ be the specification and $Impl$ be the implementation. In FDR, $Spec$ is firstly normalized. Refinement checking is then reduced to reachability analysis of the product of the $Impl$ and the normalized $Spec$. It has been shown that such an approach works well for certain models [15]. Nonetheless, because normalization in general is computational expensive, it may not be always desirable. Thus, we adopted an alternative approach. Figure 1 presents the on-the-fly refinement checking algorithm which is modified and implemented in PAT. The algorithm similarly performs a reachability analysis in the product of the implementation and the normalized specification. The different is that normalization is brought on-the-fly as well.

Details of the following procedures are skipped for brevity. Procedure $tau(S)$ explores all outgoing transition of S and returns the set of states reachable from S via a τ-transition. We remark that this procedure will be refined later. Procedure $tauclosure(S)$ implements $\tau^*(S)$ using a depth-first-search procedure. The set of states reachable from S via only τ transitions is returned. For instance, given the LTS in Example 1, $tauclosure(0)$ returns $\{0, 1, 2, 6, 7, 11\}$. The procedure $tau(S)$ is applied repeatedly until all τ-reachable states are identified. Procedure $existSuperSet(x, Y)$ where x is a set and Y is a set of sets returns true if and only if there exists y in Y such that $x \subseteq y$.

Depending on the type of refinement relationship, the algorithm performs a depth-first search for a pair (Im, NSp) where Im is a state of the implementation and NSp is a state of the normalized specification such that, the enabled events of Im is not a subset of those of NSp (`C1`), or Im is stable and there does not exist a state in NSp which refuse all events which are refused by Im (`C2`), or Im diverges but not NSp (`C3`). The

```
procedure $refines(Impl, Spec)$
0. $checked := \varnothing$
1. $pending.push((Impl, tauclosure(Spec)))$;
2. while $pending \neq \varnothing$
3.       $(Im, NSp) := pending.pop()$;
4.       $checked := checked \cup \{(Im, NSp)\}$;
5.       if $\neg(enabled(Im) \setminus \{\tau\} \subseteq enabled(NSp))$                      – C1
6.          $\vee\ (\tau \notin Im \wedge \neg\ existSuperSet(mrefusal(Im), mrefusal(NSp)))$   – C2
7.          $\vee\ (\neg\ div(NSp) \wedge div(Im))$                                             – C3
8.             return $false$;
9.       else
10.            foreach $(Im', NSp') \in \underline{next(Im, NSp)}$
11.                 if $(Im', NSp') \notin checked$
12.                      $pending := pending \cup \{(Im', NSp')\}$
13.                 endif
14.            endfor
15.       endif
16. endwhile
17. return $true$;
```

Fig. 1. Algorithm: $refines(Impl, Spec)$

algorithm returns true if no such pair is found. Note that if C1 is satisfied, a counterexample is found for any refinement checking; if C2 is satisfied, a counterexample is found for stable failures refinement checking or fairlure/divergence refinement checking; if C3 is satisfied, a counterexample is found for fairlure/divergence refinement checking only. The procedure for producing a counterexample is skipped for simplicity. Producing the shortest counterexample requires a breath-first-search after identifying the faulty state. Line 10 to 14 of algorithm $refines$ explores new states of the product and pushes them into the stack $pending$. The procedure $next$ is presented in Figure 2. Given a pair (Im, NSp), it returns a set of pairs of the form (Im', NSp') for each enabled event in Im. If the event is visible, NSp' is a successor of NSp via the event and Im' is the successor of Im via the same event. Otherwise, Im' is a successor of Im via a τ-transition and NSp' is Sp. Because normalization is brought on-the-fly, it is sometimes possible to find a counterexample before the specification is completely normalized. The soundness of the algorithm follows the soundness discussion in [12].

4.2 Partial Order Reduction

As any model checking algorithm, refinement checking suffers from state space explosion. A number of attempts have been applied to reduce the search space [15]. This section describes the one implemented in PAT based on partial order reduction. Our reduction realizes and extends the early works on partial order reduction for process algebras and refinement checking presented in [18] and [19]. The inspiration of the reduction is that events may be independent, e.g., $think.i$ is mutually independent of each other. Given $P = P_1 \parallel \cdots \parallel P_n$ and two enabled events e_1 and e_2, e_1 is dependent of e_2, written as $dep(e_1, e_2)$, and vice versa only if one of the following is true,

```
procedure $next(Im, NSp)$
0. $toReturn := \varnothing$
1. foreach $e \in enabled(Im)$
2.     if $e = \tau$
3.         foreach $Im' \in tau(Im)$
4.             $toReturn := toReturn \cup \{(Im', NSp)\};$
5.         endfor
6.     else
7.         $NSp' := \{s \mid \exists x : NSp \bullet x \xrightarrow{e} x' \land s \in tauclosure(x')\};$
8.         foreach $Im'$ such that $Im \xrightarrow{e} Im'$
9.             $toReturn := toReturn \cup \{(Im', NSp')\};$
10.        endfor
11.    endif
12. endfor
13. return $toReturn;$
```

Fig. 2. Algorithm: $next(Im, NSp)$

- e_1 and e_2 are from the same process P_i.
- $e_1 = e_2$ so that they may be synchronized, e.g., $get.i.i$ of process $Phil(i)$ and $get.i.i$ of process $Fork(i)$.
- e_1 updates a variable which e_2 depends on or vice versa, e.g., because $eat.i$ updates a global variable, all $eat.i$ are inter-dependent.

Two events are independent if they are not dependent. Because the ordering of independent events may be irrelevant to a given property, we may deliberately ignore some of the ordering so as to reduce the search space. Partial order reduction may be applied to a number of places in algorithm $refines$, namely, the procedure $tau(S)$ (and therefore $tauclosure$) and $next$. Since indexed parallel composition (and indexed interleaving) is the main source of state space explosion, we assume that Im is of the form $((P_1 \parallel P_2 \parallel \cdots \parallel P_n) \setminus X, V, C)$ in the following and show how it is possible to only explore a subset of the enabled transitions and yet preserve the soundness.

We start with applying partial order reduction to the procedure tau. Note that tau is applied to the specification or implementation independently. Thus, as long as we guarantee that the reduced state space (of either $Impl$ or $Spec$) is failures/divergence equivalent to the full state space, we prove that there is a refinement relationship in the reduced state space if and only if there is one in the full state space. Figure 3 show our algorithm for selecting a subset of the τ-transitions. The soundness proof is presented in Appendix B. In the algorithm tau', we try to identify one set of τ-transitions which are independent of the rest. If such a subset is found (i.e., the algorithm $stubborn\_tau$ returns a non-empty set of successors), only the subset is explored further. Otherwise (i.e., $stubborn\_tau$ returns an empty set), all possible τ-transitions are explored. In $stubborn\_tau$, the idea is to identify one process P_i such that all τ-transitions from P_i are independent of those from other processes. Note that this approach is most effective with τ-transition generated from one process only. It is possible to handle τ-transition generated from multiple processes with a slightly more complicated procedure (which we skip for brevity). The details of the following simple

```
procedure $tau'(Im)$
0. $nextmoves := stubborn\_tau(Im)$;
1. if $(nextmoves \neq \varnothing)$ then return $nextmoves$;  else return $tau(Im)$;

procedure $stubborn\_tau(Im)$
0. foreach $P_i$
1.       $por := enabled_{P_i}(Im) \subseteq \{\tau\} \cup X \wedge enabled_{P_i}(Im) = current(P_i)$
2.       foreach $e \in enabled_{P_i}(Im)$
3.             $por := por \wedge \neg\, loop(e) \wedge \forall\, e' : \Sigma_j \bullet j \neq i \Rightarrow \neg\, dep(e, e')$
4.       endfor
5.       if $por$ then
6              return $\{(((\cdots \parallel P_i' \parallel \cdots) \setminus X), V, C') \mid (P_i, V, C) \xrightarrow{e} (P_i', V, C')\}$;
7.       endif
8. endfor
9. return $\varnothing$;
```

Fig. 3. Algorithm: $tau'(Im)$ and $stubborn\_tau(Im)$

procedures have been skipped. Given $Im = (P, V, C)$, $enabled_{P_i}(Im)$ is the set of enabled event from component P_i, i.e., $enabled(Im) \cap enabled((P_i, V, C))$. For instance, given $College$ with $N = 2$, $enabled(Pair(0))$ is $\{get.0.1\}$. $current(P_i)$ is the set of events that could be enabled in process P_i given the most cooperative environment. For instance, $current(Phil(i)) = \{get.i.(i + 1)\%N\}$ despite whether the fork is available or not. $loop(e)$ is true if and only if performing this event results in a state on the search stack, i.e., forming a cycle.

A process P_i is considered a candidate only if all enabled events from P_i result in τ-transitions (i.e., $enabled_{P_i}(Im) \subseteq \{\tau\} \cup X$) and no other transition could be possibly enabled given a different environment (i.e., $enabled_{P_i}(Im) = current(P_i)$). The former is required because we are only interested in τ-transitions. The latter (partly) ensures that no disabled event from P_i is enabled before executing an event from P_i. Furthermore, all enabled events from P_i must not form a cycle (so that an enabled event is not skipped for ever) or dependent on an enabled event from some other component. For detailed discussion on the intuition behind these conditions, refer to [4].

Example 3. Assume that $N = 2$ and the following is the current process expression,

$$((think.0 \rightarrow Phil(0) \parallel put.1.0 \rightarrow Fork(0)) \setminus \{get.0.0, put.0.0, think.0\}) \parallel$$
$$(((get1.1 \rightarrow eat.1 \rightarrow put.1.0 \rightarrow put.1.1 \rightarrow think.1 \rightarrow Phil(1)) \parallel Fork(1))$$
$$\setminus\{get.1.1, put.1.1, think.1\}) \setminus \{get.0.1, get.1.0, put.0.1, put.1.0\}$$

where the first philosopher has just put down both forks while the second one has just picked up his first fork. Two τ-transitions are enabled, i.e., one due to $think.0$ and the other due to $get.1.1$. The algorithm tau' would return only the successor state after performing $get.1.1$ (assuming it is not on the stack). This is the only event enabled for the second component of the outer parallel composition is the τ transition due to $get.1.1$ (and thus the condition at line 1 of $stubborn\_tau$ is satisfied). Because $get.1.1$ is local to the component, por is true after the loop from line 2 to line 4. □

```
procedure $next'(Im, Sp)$
0. if $\tau \in enabled(Im)$
1.     $nextmoves := stubborn\_tau(Im)$;
2.     if $(nextmoves \neq \varnothing)$ then return $nextmoves$;
3. else
4.     foreach $e \in enabled(Im)$
5.         $por := stubborn\_visible(Im, e)$;
6.         foreach $S \in Sp$
7.             $por := por \wedge stubborn\_visible(S, e)$;
8.         endeach
9.         if $por$ then return $\{(Im', tauclosure(Sp')) \mid Im \xrightarrow{e} Im' \wedge Sp \xrightarrow{e} Sp'\}$
10.    endeach
11. return $next(Im, Sp)$;

procedure $stubborn\_visible(Im, e)$
0. $por := \neg\, loop(e) \wedge \forall\, e' : \Sigma_j \bullet e' \neq e \Rightarrow \neg\, dep(e, e')$;
1. foreach $P_i \in processes(e)$
2.     $por := por \wedge enabled_{P_i}(Im) = current(P_i) = \{e\}$;
3. return $por$;
```

Fig. 4. Algorithm: $next'(Im, Sp)$ and $stubborn\_visible(Im, e)$

The above algorithms apply partial order reduction to τ-transitions only. $tauclosure$ is refined as well since it is based on tau'. Unlike FDR, PAT is capable of applying partial order reduction to visible events. Because both $Impl$ and $Spec$ must make corresponding transitions for a visible event, reduction for visible events is complicated. A conservative approach has been implemented in PAT. Figure 4 present the algorithm, i.e., the refined $next$. If Im is not stable, we apply the algorithm $stubborn\_tau$ to identify a subset of τ-transitions (line 1). If no such subset exists, the pair (Im, Sp) is fully expanded (line 11). An algorithm $stubborn\_visible$ similar to $stubborn\_tau$ is used to check if a given visible event e is a candidate for partial order reduction. Function $processes(e)$ returns all process components (of the parallel composition) whose alphabet contains e. Firstly, we choose a possible candidate from Im using the algorithm $stubborn\_visible$. Event e is chosen if and only if, for each process in $processes(e)$, e is the only event from the process which can be enabled and all other enabled events are independent of e and performing e does not result in a state on the stack. Next, we check if e satisfies the same set of conditions for each state in the normalized state of the specification. If it does, e is used to expand the search tree at line 9 (and all other enabled events are ignored). In order to find such e efficiently, the candidate events are selected in a pre-defined order, i.e., events which have the least number of associated processes are chosen first. The soundness of the algorithm is presented in Appendix B.

Example 4. Let $P(i) = a.i \rightarrow b.i \rightarrow P(i)$. Assume the specification and implementation is defined as: $Spec = \|_{i=0}^{2} P(i)$ and $Impl = \|_{i=0}^{1} P(i)$. Assume we need to show that $Impl$ trace-refines $Spec$. Initially, two events are enabled in $Impl$, i.e., $a.0$ and $a.1$. Assume that $a.0$ is selected first, because $loop(a.0)$ is false and $a.0$ is independent of all other enabled events (i.e., $a.1$), the condition at line 0 of algorithm $stubborn\_visible$

is satisfied. Because $a.0$ is the only event that would possibly be enabled from $P(0)$, the condition at line 2 is satisfied too. Thus, $a.0$ is a possible candidate for partial order reduction for *Impl*. Similarly, it is also a candidate for *Spec* (which is the only state in the normalized initial state). Therefore, we only need to explore $a.0$ initially. □

4.3 Refinement Checking Experiments

We compare PAT with FDR using benchmark models for refinement checking. For the sake of a fair comparison, all models use only standard CSP features which are supported by both. The following table shows the experiment results for three models, obtained on a 2.0 GHz Intel Core Duo CPU and 1 GB memory.

model	N	property	result	PAT	FDR
Dining Philosophers	5	P [T= S	true	0.28125	0.067
Dining Philosophers	6	P [T= S	true	0.8593	0.069
Dining Philosophers	8	P [T= S	true	13.78	0.076
Dining Philosophers	10	P [T= S	true	430.28	0.107
Dining Philosophers	12	P [T= S	true	-	0.319
Reader/Writers	12	P [T= S	true	< 1	0.812
Reader/Writers	14	P [T= S	true	< 1	6.906
Reader/Writers	16	P [T= S	true	< 1	81.247
Reader/Writers	18	P [T= S	true	< 1	-
Reader/Writers	50	P [T= S	true	1.097	-
Reader/Writers	100	P [T= S	true	9	-
Reader/Writers	200	P [T= S	true	77.515	-
Milner's Cyclic Scheduler	11	P [T= S	true	< 1	19.011
Milner's Cyclic Scheduler	13	P [T= S	true	< 1	419.021
Milner's Cyclic Scheduler	14	P [T= S	true	< 1	-
Milner's Cyclic Scheduler	50	P [T= S	true	2.406	-
Milner's Cyclic Scheduler	100	P [T= S	true	9.765	-
Milner's Cyclic Scheduler	200	P [T= S	true	60.453	-

The first example is the classic dining philosopher problem, where N is the number of philosophers and forks. Because of the modeling, partial order reduction is not effective for this example. As a result, PAT handles about 10^7 states (about 11 philosophers and forks) in a reasonable amount of time. FDR performs extremely well for this example because of the strategy discussed in [15]. Namely, it builds up a system gradually, at each stage compressing the subsystems to find an equivalent process with (for this particular example) many less states. Notice that with manual hiding (to localize some events), PAT performs much better. The second example is the classic readers/writers problem, in which the readers and writers coordinate to ensure correct read/write ordering. N is the number of readers/writers. Reduction in PAT is very effective for this example. As a result, PAT handles a few hundreds readers/writers efficiently, whereas FDR suffers from state space explosion quickly (for $N = 18$). The third example is the Milner's cyclic scheduling algorithm, in which multiple processes are scheduled in a cyclic fashion. Partial order reduction is extremely effective for this model. As a result, PAT handles hundreds of processes, whereas FDR handles less than 14 processes. The experiment results show our best effort by far on automated model checking of an extended version of CSP. It by no means suggests the limit of our tool. We believe that by incorporating more reduction techniques (e.g., symmetry reduction) as well as fine-tuning the implementation, the performance of PAT can be improved significantly.

4.4 Temporal Logic Based Verification

Verification of CSP models has been traditionally based on refinement checking. CSP refinement is expressive enough to cover a large class of properties [14]. Nonetheless, temporal logic formulae have been proved effective as well as intuitive. Verification based on temporal logic has gathered much, evidenced by the rich set of theories and tools developed for CTL/LTL based verification [3,8]. In this section, we briefly discuss the LTL model checker embedded in PAT. We adopt an automata-based approach for explicit LTL model checking as Spin [8]. Because we are dealing with an event-based formalism, we extend standard Linear Temporal Logic (LTL) with events so that properties concerning both states and events can be stated and verified. For instance, the following specifies a desirable property of process $College$: $\Box\Diamond eat_0 \land \Box\Diamond eat_1 \cdots \Box\Diamond eat_{N-1}$ where $\Box$ reads as "always" and $\Diamond$ reads as "eventually". The property states that every philosopher will always eventually eat, i.e., no one starves.

Definition 5. *Let Pr be a set of propositions. An extended LTL formula is*[3]*,*

$$\phi ::= p \mid a \mid \neg\,\phi \mid \phi \land \psi \mid \Box\phi \mid \Diamond\phi \mid \phi U \psi$$

where p ranges over Pr and a ranges over Σ. Let $\pi = \langle P_0, x_0, P_1, x_1, \cdots\rangle$ be an infinite sequence of events. Let π^i be the suffix of π starting from P_i.

$$\begin{array}{ll}
\pi^i \models p & \Leftrightarrow P_i \models p \\
\pi^i \models a & \Leftrightarrow x_{i-1} = a \\
\pi^i \models \neg\,\phi & \Leftrightarrow \neg(\pi^i \models \phi) \\
\pi^i \models \phi \land \psi & \Leftrightarrow \pi^i \models \phi \land \pi^i \models \psi \\
\pi^i \models \Box\phi & \Leftrightarrow \forall\, j \geq i \bullet \pi^j \models \phi \\
\pi^i \models \Diamond\phi & \Leftrightarrow \exists\, j \geq i \bullet \pi^j \models \phi \\
\pi^i \models \phi U \psi & \Leftrightarrow \exists\, j \geq i \bullet \pi^j \models \psi \land \forall\, k \mid i \leq k \leq j-1 \bullet \pi^j \models \phi
\end{array}$$

The simplicity of writing formulae concerning events as in the above example is not purely a matter of aesthetics. It may yield gains in time and space (refer to examples in [2]). Given an extended LTL formula, PAT internally constructs a trace equivalent Büchi automaton using the state-of-the-art conversion proposed in [6]. For efficient reasons, the Büchi automata are transition-labeled (instead of state-labeled). Let $B^{\neg\,\phi}$ be the Büchi automaton constructed from property $\neg\,\phi$. The product of $B^{\neg\,\phi}$ and the model is generated. Two different algorithms (e.g., nested depth first search and strongly

[3] The next operator is not supported purposely because of partial order reduction.

connected component search based on Tarjan's algorithm) are then used to determine the emptiness of the product, i.e., explore on-the-fly whether the product contains a loop which is composed of at least one accepting state. Finite traces are extended to infinite ones in a standard way. In the presence of a counterexample, on-the-fly model checking usually produces a trace leading to a bad state or a loop quickly (refer to Section 4.3). Partial order reduction (similar to the one implemented in Spin) is applied for LTL verification. One unique feature of LTL verification in PAT is that we allow fairness assumptions to be associated with individual events and then verify the system under the fairness assumptions. For details, refer to [9].

5 Conclusion and Future Works

We present PAT, a process analysis toolkit, designed to apply model checking techniques to verify event-based compositional models. A number of verification algorithms have been implemented. This paper is related to works on developing tool support for CSP and process algebras, works on heuristics for partial order reduction and works on model checking in general. ARC (Adelaide Refinement Checker [11]) is a refinement checker based on ordered binary decision diagrams (BDD). It has been shown that ARC outperforms FDR in a few cases [11]. PAT adapts an explicit approach for model checking. It has long been known there are pros and cons choosing an explicit approach or a BDD approach (refer to comparisons between SPIN and SMV). Nonetheless, in the future, we may incorporate partial order reduction and BDD to achieve better performance. ProBE [1] is a simulator developed by Formal Method Europe to interactively explore traces of a given process. The simulator embedded in PAT has the full functionality of Probe. Though we have shown cases where PAT outperforms FDR, we believe that a full comparison is yet to be carried out with more experiments. A number of algorithms have been previously proposed for partial order reduction which is trace/failures/divergence preserving, e.g., [18,19]. The algorithms presented in the paper may be considered as lightweight realization and extension of those presented in [18,19]. In addition, this work is related to the huge amount of works dedicated to theories and tools development for model checking.

We are actively developing PAT. There are a number of directions to pursue in the future. Firstly, based our framework, more languages features will be incorporated, e.g., higher-order processes, broadcasting communication, integrated data operations as in integrated languages [10,5,17], etc. Secondly, more advanced reduction techniques will be incorporated. Lastly, a broad range of experiments and case studies must be performed to not only fully compare PAT with FDR but also to make PAT as a reliable and extensible framework for developing model checking techniques.

Acknowledgement

This work is partially supported by the research project "Sensor Networks Specification and Validation" (R-252-000-320-112) funded by Ministry of Education, Singapore.

References

1. Formal Systems (Europe) Ltd. Process Behaviour Explorer (2003), http://www.fsel.com/probe_download.html
2. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
3. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
5. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed Patterns: TCOZ to Timed Automata. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 483–498. Springer, Heidelberg (2004)
6. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
7. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1985), www.usingcsp.com/cspbook.pdf
8. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. on Soft. Eng. 23(5), 279–295 (1997)
9. Sun, J.S.D.J., Liu, Y., Wang, H.H.: Specifying and Verifying Event-based Fairness Enhanced Systems. In: Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM 2008) (accepted, 2008)
10. Mahony, B., Dong, J.S.: Timed Communicating Object Z. IEEE Transactions on Software Engineering 26(2) (February 2000)
11. Parashkevov, A., Yantchev, J.: ARC - a Tool for Efficient Refinement and Equivalence Checking for CSP. In: Proceedings of the IEEE International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 1996), pp. 68–75 (1996)
12. Roscoe, A.W.: Model-checking CSP, pp. 353–378 (1994)
13. Roscoe, A.W.: Compiling Shared Variable Programs into CSP. In: Proceedings of PROGRESS workshop 2001 (2001)
14. Roscoe, A.W.: On the expressive power of CSP refinement. Formal Aspects of Computing 17(2), 93–112 (2005)
15. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)
16. Roscoe, A.W., Wu, Z.Z.: Verifying Statemate Statecharts Using CSP and FDR. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 324–341. Springer, Heidelberg (2006)
17. Sun, J., Dong, J.S.: Design Synthesis from Interaction and State-Based Specifications. IEEE Transactions on Software Engineering 32(6), 349–364 (2006)
18. Valmari, A.: Stubborn Set Methods for Process Algebras. In: Proceedings of the Workshop on Parital Order Methods in Verification (PMIV 1996). DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, pp. 213–231 (1996)
19. Wehrheim, H.: Partial order reductions for failures refinement. Electronic Notes in Theoretical Computer Science 27 (1999)

Appendix A: Operational Semantics

The following Structural Operational Semantics (SOS) rules. We remark that $\Box$, $\sqcap$, $\|$ and $|||$ are symmetric and associative. $eval(V, exp)$ evaluates the value of the exp given valuation V. Notice that for any P, $\checkmark \in \Sigma_P$.

$$\frac{}{(Skip, V, C) \xrightarrow{\checkmark} (Stop, V, C)} \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \neq \checkmark}{(P \mid\mid\mid Q, V, C) \xrightarrow{e} (P' \mid\mid\mid Q, V', C')} [\ int_1\ ]$$

$$\frac{}{(e\{x = exp\} \rightarrow P, V, C) \xrightarrow{e} (P, V[x/eval(V, exp)], C)} [\ ass\ ]$$

$$\frac{\neg\, full(C[c])}{(c!exp \rightarrow Q, V, C) \xrightarrow{c!eval(V,exp)} (Q, V, C[c/C[c] \frown \langle c!eval(V, exp)\rangle])} [\ output\ ]$$

$$\frac{\neg\, empty(C[c])}{(c?x \rightarrow Q, V, C) \xrightarrow{c?C[c].head} (Q, V, C[c/C[c].tail])} [\ input\ ]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C')}{(P;\ Q, V, C) \xrightarrow{e} (P';\ Q, V', C')} \qquad \frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C')}{(P;\ Q, V, C) \xrightarrow{\tau} (Q', V', C')} [\ seq_2\ ]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \neq \tau}{(P \Box Q, V, C) \xrightarrow{e} (P', V', C')} [\ ex_1\ ] \qquad \frac{(P, V, C) \xrightarrow{\tau} (P', V', C')}{(P \Box Q, V, C) \xrightarrow{\tau} (P' \Box Q, V', C)}$$

$$\frac{}{(P \sqcap Q, V, C) \xrightarrow{\tau} (P, V, C)} \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C'), V \models b}{(P \lhd b \rhd Q, V, C) \xrightarrow{e} (P', V', C')} [\ con_1\ ]$$

$$\frac{(Q, V, C) \xrightarrow{e} (Q', V', C'), V \not\models b}{(P \lhd b \rhd Q, V, C) \xrightarrow{e} (Q', V', C')} \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C'), V \models b}{([b] \bullet P, V, C) \xrightarrow{e} (P', V', C')} [\ grd\ ]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C')}{(P \bigtriangleup Q, V, C) \xrightarrow{e} (P' \bigtriangleup Q, V', C')} \qquad \frac{(Q, V, C) \xrightarrow{e} (Q', V', C'), e \neq \tau}{(P \bigtriangleup Q, V, C) \xrightarrow{e} (Q', V', C')} [\ int_2\ ]$$

$$\frac{(Q, V, C) \xrightarrow{\tau} (Q', V', C')}{(P \bigtriangleup Q, V, C) \xrightarrow{\tau} (P \bigtriangleup Q', V', C')} \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \notin X}{(P \setminus X, V, C) \xrightarrow{e} (P' \setminus X, V', C')}$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \in X}{(P \setminus X, V, C) \xrightarrow{\tau} (P' \setminus X, V', C')} \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C') \wedge e \notin \Sigma_Q}{(P \parallel Q, V, C) \xrightarrow{e} (P' \parallel Q, V', C')}$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V, C) \wedge (Q, V, C) \xrightarrow{e} (Q', V, C)}{(P \parallel Q, V, C) \xrightarrow{e} (P' \parallel Q', V, C)} [\ syn\ ]$$

Appendix B: Soundness of the Partial Order Reduction

We prove the soundness in two steps. Firstly, because the algorithm tau' applies to one model only (whereas $next'$ must coordinate both the implementation and the specification), it is sufficient to show that the reduction regarding τ-transitions (i.e., the algorithm $stubborn\_tau$) preserves failures/divergence equivalence. Secondly, we show that the reduction regarding visible events (i.e., the algorithm $next'$) is sound.

In [18], a set of sufficient conditions has been proved to preserve CSP failures divergence equivalence. It is thus sufficient to prove that the reduction regarding τ-transitions satisfies the sufficient conditions. In the following, let E be the reduced set of successors (i.e., the stubborn set as in [18]) and F be the full set. Notice that the result returned by algorithm $stubborn\_tau$ is returned by algorithm tau' or $next'$ if and only if it is not empty (line 1 of tau' and line 2 of $next'$). Thus, as long as F is not empty, E is not empty. By line 3 of algorithm $stubborn\_tau$, transitions other than those selected in E are all independent of those in E. By line 1 of $stubborn\_tau$, because the set of possibly enabled events must be the same of the set enabled event from the component, a transition from the component must remain disabled unless a transition from the components has been taken. By theorem 3.2 of [18], $\ddot{A}_0$, $\ddot{A}_1$, $\ddot{A}_2$, $\ddot{A}_3$ hold. Because only τ-transitions are reduced in tau', condition $\ddot{A}_4$ is trivial. By the condition $\neg\, loop(e)$ at line 3 of $stubborn\_tau$, no action will be ignored forever, and thus $\ddot{A}_5$ holds. $\ddot{A}_6$ is trivial for the same reason as for $\ddot{A}_4$. By theorem 4.2 and 5.3 in [18], the reduction regarding τ-transitions preserves trace/failures/divergence equivalence and thus is sound.

In order to prove that algorithm $next'$ is sound, we need to prove (in addition to the above) that the reduction regarding visible events are sound as well. We reuse the results which have been proved in [19] and show that the sufficient conditions proposed in [19] have been full-filled. Firstly, C1 and C3 in [19] are trivial true. Because of line 0 and 2 of $stubborn\_visible$, an action dependent (say e) on an action selected can only be executed after some action selected has been executed. There are two cases in which this might be violated. In both of these cases, some transition (say a) independent of e are executed, eventually enabled a transition that is dependent on e. In the first case, if a belongs to some other components. A necessary condition for this to happen is that a is dependent on e. This is prevented by line 1. In the other case, a belongs to the same component of e, which is not possible because we require that $current(P_i) = \{e\}$. The same argument applies to line 6 to 8 which guarantees that no action dependent on e is executed before e is executed (and there C1in [19] is proved). C2 in [19] is guaranteed by the condition $\neg\, loop(e)$. Therefore, we conclude the reduction is sound.