Specifying and Verifying Event-based Fairness Enhanced Systems

Jun Sun[1], Yang Liu[1], Jin Song Dong[1] and Hai H. Wang[2]

[1] School of Computing,
National University of Singapore
{sunj,liuyang,dongjs}@comp.nus.edu.sg
[2] School of Electronics and Computer Science,
University of Southampton
hw@ecs.soton.ac.uk

Abstract. Liveness/Fairness plays an important role in software specification, verification and development. Existing event-based compositional models are safety-centric. In this paper, we describe a framework for systematically specifying and verifying event-based systems under fairness assumptions. We introduce different event annotations to associate fairness constraints with individual events. Fairness annotated events can be used to embed liveness/fairness assumptions in event-based models flexibly and naturally. We show that state-of-the-art verification algorithms can be extended to verify models under fairness assumptions, with little computational overhead. We further improve the algorithm by other model checking techniques like partial order reduction. A toolset named PAT has been developed to verify fairness enhanced event-based systems. Experiments show that PAT handles large systems with multiple fairness assumptions.

1 Introduction

Critical system requirements like safety, liveness and fairness play important roles in system/software specification, verification and development. Safety properties ensure that something undesirable never happens. Liveness properties state that something desirable must eventually happen. Fairness properties state that if something is enabled sufficiently often, then it must eventually happen. Often, fairness assumptions are necessary to prove liveness properties.

Over the last decades, specification and verification of safety properties have been studied extensively. There have been many languages and notations dedicated to safety-critical systems, e.g., Z, VDM, CCS and CSP. The concept of liveness itself is problematic [17]. Fairness constraints have been proved to be an effective way of expressing liveness, not mentioning that itself is important in system specification and verification. For instance, without fairness constraints, verifying of liveness properties may often produce counterexamples which are due to un-fair executions, e.g., a process or choice is infinitely ignored. State-based fairness constraints have been well studied in automata theory based on accepting states, e.g., in the setting of Büchi/ Rabin/Streett/Muller automata. It has been observed that the notion of fairness is not easily combined with the bottom-up type of compositionality (of process algebra for instance [23]), which is important for attacking the complexity of system development.

A common practice of verifying liveness relies on explicitly stating all fairness assumptions as premises of the liveness properties. This approach is not feasible if there are many fairness constraints. Note that it is relatively straightforward to verify whether the system satisfies a fairness property. It is verification under multiple fairness assumptions which may be infeasible. For instance, a method to prove that a program satisfies some property is Linear Temporal Logic (LTL) model checking. Given an LTL formula ϕ, the model checker transforms the negation of ϕ into a Büchi automaton, builds the product of the automaton and the program and then checks this product for emptiness. The size of the constructed Büchi automaton is exponential to the length of ϕ. A formula composing of many fairness premises results in a huge Büchi automaton and thus makes model checking infeasible. For example, SPIN is a rather popular LTL model checker [15]. The algorithm it uses for generating Büchi automata handles only a limited number of fairness constraints. The following table shows experiments on the time and space needed for SPIN to generate the automaton from standard notion of fairness, in particular, justice and compassion [16].

Prop.	**n**	**Time (Sec.)**	**Memory**	**#Büchi States**
$(\bigwedge_{i=1}^{n} \Box\Diamond p_i) \Rightarrow \Box\Diamond q$	1	0.08	466Kb	74
same above	3	4.44	27MB	1052
same above	5	more than 3600	more than 1Gb	–
$(\bigwedge_{i=1}^{n} (\Box\Diamond p_i \Rightarrow \Box\Diamond q_i)) \Rightarrow \Box\Diamond s$	1	0.13	487.268	134
same above	2	1.58	10123.484	1238
same above	3	30.04	55521.708	4850
same above	4	4689.24	more than 1Gb	–

The experiments are made on a 3.0GHz Pentium IV CPU and 1 GB memory executing SPIN 4.3. The results show that it takes a non-trivial amount of time to handle 5 fairness constraints. In order to overcome this problem, SPIN offers an option to handle weak fairness on the level of processes. However, it may not be always sufficient. Process-level fairness states that all enabled events from different processes must be engaged or disabled eventually, which is overwhelming sometimes. For instance, it is reasonable to require that a submitted request must eventually be served, while it is not to require that always eventually there is a request. Another approach [16] to model check under fairness assumptions is to model fairness using global accepting states (or in the form of justice/compassion condition [16]). For instance, a run accepted by a Büchi automaton must visit at least one accepting state infinitely often. This approach is not applicable to event-based compositional systems.

In [17], a language independent definition of event-based fairness has been proposed and studied. Let e be an event. Let $\Diamond$, $\Box$ be the temporal operators which informally reads as 'eventually' and 'always' respectively.

$$wf(e) \mathrel{\widehat{=}} \Diamond\Box(e \textit{ is enabled}) \Rightarrow \Box\Diamond(e \textit{ is engaged})$$
$$sf(e) \mathrel{\widehat{=}} \Box\Diamond(e \textit{ is enabled}) \Rightarrow \Box\Diamond(e \textit{ is engaged})$$

The weak fairness $wf(e)$ asserts that if an event e eventually becomes enabled forever, infinitely many occurrences of the event steps must be observed. The strong fairness

$sf(e)$ asserts that if e is infinitely often enabled (or in other words, repeatedly enabled), infinitely many occurrences of the event must be observed. Strong fairness implies weak fairness. A system satisfies a fairness constraint if and only if every run of the system satisfies the fairness constraint.

Partly inspired by the work above, we propose an alternative approach for specifying fairness constraints, with efficient verification in mind. Instead of stating the fairness assumptions as a part of the property to verify or additional global accepting states, they are embedded in the compositional models. We introduce different event annotations to associate fairness assumption with individual events. Event-based annotation allows us to model fairness naturally and flexibly. For instance, by annotating all events weak fair, we achieve process-level fairness (as offered in SPIN). Next, we investigate automated verification of models with event-based fairness. We show that existing state-of-the-art verification algorithms can be extended to handle event-based fairness with little computational overhead. An on-the-fly model checking algorithm is developed. The algorithm is further improved by techniques like partial order reduction. A toolset named PAT (stands for Process Analysis Toolset) has been developed to realize the algorithms (which also supports functionalities like standard model checking, model simulation, etc). Experiment results show that PAT handles non-trivial systems with multiple fair constraints efficiently.

Our contribution includes an approach to model fairness in event-based compositional system models and an on-the-fly model checking algorithm with partial order reduction for verifying those models. This paper is related to works on specification and verification of liveness and fairness, e.g., verification under weak fairness in SPIN and early works discussing liveness associated with events in the framework of Promela, CCS or CSP, evidenced in [15, 7, 6, 21]. One way of capturing fairness is to alter the language semantics so that all events are fairly treated (e.g., [15, 21, 3]), i.e., the semantics of parallel composition is enhanced to be fair. From our point of view, the difference between *parallel* and *sequential* processes may be irrelevant and fairness shall be independent of one particular operator. In practice, it may be that only certain events need to fulfill fairness constraints. For instance, a common requirement is that "some action must eventually occur if some other action occurs" (i.e., compassion conditions). In other works [7, 6], events or processes are annotated with special markings to capture fairness. However, previous approaches do not easily combine with the bottom-up compositionality of process algebra, i.e., the fairness constraints may be lost once the process is composed with others. In this work, annotations are used to associate different kinds of fairness assumptions with relevant events in the relevant module/process, which may yet has global effects. Moreover, we develop automated verification support for our notion of fairness. We remark that the concept of event-based fairness is not limited to process algebras. Specification of fairness in programming languages has been discussed in the line of works by Apt, Francez and Katz [2]. Event-based fairness may allow fairness constraints that may not be feasible (and thus violates one of the principles in [2]), which makes our verification techniques crucial. Our works on verification of models with embedded fairness constraints are related to previous works on verification under fairness assumptions [16, 18, 13], in which liveness/fairness constraints are either specified using temporal logic formulae or captured using *global* accepting states.

We use a different way of specifying fairness constraints and hence our model checking algorithm is different from theirs. We also extend our algorithm with techniques like partial order reduction to handle large systems. This work is remotely related to our previous works on verification of event-based specifications [10, 9, 19].

The remainder of the paper is organized as follows. Section 2 reviews the input language of PAT and its semantics. Section 3 introduces our event annotations. Section 4 presents and evaluates the on-the-fly model checking algorithm and partial order reduction. Section 5 concludes the paper.

2 Background

Without loss of generality, we present our ideas using a simple compositional language which supports concurrency, multi-threaded synchronization, shared variables and assignments. In the next section, we will extend this language with fairness annotations.

A model is composed of a set of global variables and a set of process definitions. One of the processes is identified by the starting process (as the main method in Java), which captures the system behaviors after initialization. A process is defined as using the following constructs. Most of the compositional operators are borrowed from the classic CSP [14].

$$\begin{aligned} P \mathrel{\widehat{=}} & \mathit{Stop} \mid \mathit{Skip} \mid e \rightarrow P_1 \mid P_1;\ P_2 \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \\ & \mid P_1 \lhd b \rhd P_2 \mid [b] \bullet P \mid P_1 \bigtriangleup P_2 \mid P_1 \,|[\, X \,]|\, P_2 \end{aligned}$$

where b is a Boolean expression, X is a set of events and e is an event. Note that e could be an abstract event (single or compound) or an assignment (e.g., $x := x + 1$). Process Stop does nothing but deadlocks. Process Skip terminates successfully. Event prefixing $e \rightarrow P$ is initially willing to engage in event e and behaves as P afterward. $\mathit{Skip} = \checkmark \rightarrow \mathit{Stop}$ where $\checkmark$ is the termination event. The sequential composition, $P_1;\ P_2$, behaves as P_1 until its termination and then behaves as P_2. A choice between two processes is denoted as $P_1 \Box P_2$ (for external choice) or $P_1 \sqcap P_2$ (for nondeterminism). 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. The state guard $[b] \bullet P$ is blocked until b is true and then proceeds as P, i.e., a guarded command. $P_1 \bigtriangleup P_2$ behaves as P_1 until the first event of P_2 is engaged, then P_1 is interrupted and P_2 takes control.

One reason for using a CSP-based language is to study fairness assumptions in a setting with multi-threaded lock-step synchronization. Let Σ_P be the alphabet of P which excludes τ (internal action) and $\checkmark$. Note that alphabets can be manually set or by default be the set of events constituting the process expression. Parallel composition of processes is written as $P \,|[\, X \,]|\, Q$. Events in X must be synchronized by both processes. If X is empty, P and Q run in parallel independently. X is skipped if it is exactly $\Sigma_P \cap \Sigma_Q$. Multiple processes may run in parallel, written as $P_1 \parallel P_2 \parallel \cdots \parallel P_n$. Shared events must be synchronized by all processes whose alphabet contains the event. Recursion is allowed by process referencing. The semantics of recursion is defined as Tarski's weakest fixed-point. The valuation of the variables is a set of pairs which map a variable to its current value. A system state is a pair (P, V) where P is a process expression and V is the valuation of the global variables.

Example 1. The classic dining philosophers example [14] is used as a running example.

$$\begin{aligned}
Phil(i) &= get.i.(i+1)\%N \rightarrow get.i.i \rightarrow eat.i \rightarrow put.i.(i+1)\%N \rightarrow \\
&\qquad put.i.i \rightarrow Phil(i) \\
Fork(i) &= get.i.i \rightarrow put.i.i \rightarrow Fork(i) \,\square \\
&\qquad get.(i-1)\%N.i \rightarrow put.(i-1)\%N.i \rightarrow Fork(i) \\
College(N) &= \Vert_{i=0}^{N-1} (Phil(i) \parallel Fork(i))
\end{aligned}$$

where N is 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. Process $Phil(i)$ models the behaviors of the i-th philosopher. Process $Fork(i)$ models the behaviors of the i-th fork. Process $College(N)$ is the indexed parallel composition of multiple philosophers and forks. It is known that $College(N)$ deadlocks when each dining philosopher picks up one fork. By asking one of the philosophers to pick up the forks in a different order, the system becomes deadlock-free. □

We focus on the operational semantics in this paper. The operational semantics for CSP presented in [4] has been extended with shared variables (presented in [28]). The sets of behaviors of processes can equally and equivalently be extracted from the operational semantics, thanks to congruence theorems. Fairness properties state that an event which is either repeatedly or continuously enabled must eventually occur. They therefore affect only the infinite, and not finite, traces of a process. We present an infinite trace semantics, inspired by the infinite trace semantics for CSP [25]. Note that finite traces are extended to infinite ones in a standard way, i.e., by attaching infinite number of idling events to the rear. Let Σ^* and Σ^ω be the set of finite and infinite sequences of events respectively.

Definition 1 (Infinite Traces). *Let P be a process and V be a valuation of the data variables. The set of infinite traces is written as $inftr(P, V)$. An infinite trace $\tilde{tr} : \Sigma^\omega$ is in $inftr(P, V)$ if and only if there exists an infinite sequence of $\tilde{P}$ and $\tilde{V}$ such that*

- *$\tilde{P}(0) = P$ and $\tilde{V}(0) = V$,*
- *for all i, $(\tilde{P}(i), \tilde{V}(i)) \overset{\tilde{tr}(i)}{\Rightarrow} (\tilde{P}(i+1), \tilde{V}(i+1))$*

where $\Rightarrow$ is the smallest transition relation defined by the operational semantics [28].

An infinite run of a process P with variable valuation V is an alternating infinite sequence of states and events $(\tilde{P}(0), \tilde{V}(0)), \tilde{a}(0), \cdots (\tilde{P}(i), \tilde{V}(i)), \tilde{a}(i), \cdots$ which conforms to the operational semantics. An infinite sequence of events is a trace if and only if there is a run with the exact same sequence of events. A state (P', V') is reachable from (P, V) if and only if there is a finite run from (P, V) to (P', V').

Definition 2 (Enabledness). *Let P be a process. Let V be a valuation of the data variables. $enabled(P, V) = \{e : \Sigma \mid \exists\, P', V' \bullet (P, V) \overset{e}{\Rightarrow} (P', V')\}$.*

Given P and V, an event is enabled if and only if it is in $enabled(P, V)$. It is disabled if it is not enabled. Note that given a parallel composition $P \,|[\, X \,]|\, Q$, an event in X is enabled in the composition if and only if it is enabled in both P and Q.

It is known that CSP (as well as CCS) lacks the notion of liveness or fairness [14, 6]. An event can be enabled *forever* but never be engaged, or an event may be enabled *infinitely often* but never been engaged. In the paper, we assume properties are stated in the form of LTL formulae. Different from stand LTL, we adopt the work presented in [5] so that events may be used to form LTL formulae. A desirable property for process $College(5)$ is $\Box\Diamond eat.0$, i.e., always eventually the 0-th philosopher eats and thus never starves. Note that this property is not true, i.e., $College(5) \not\models \Box\Diamond eat.0$. The following traces may be returned as counterexamples.

$$\begin{array}{ll}
\langle get.0.1, get.1.2, get.2.3, get.3.4, get.4.0\rangle & \text{– T0} \\
\langle get.3.4, get.3.3, eat.3, put.3.4, put.3.3\rangle^{\omega} & \text{– T1} \\
\langle get.1.2, get.1.1, eat.1, put.1.2, put.1.1\rangle^{\omega} & \text{– T2}
\end{array}$$

Assume that given a trace tr, tr^{ω} repeats tr infinitely. T0 is a trace which leads to the deadlock situation, in which case all philosophers starve. T1 and T2 are returned because the model lacks of both weak and strong fairness. In T1's scenario, the 3-rd philosopher greedily gets the forks and eats forever. This counterexample is due to lack of weak fairness, i.e., the event $get.0.1$ is always enabled but never engaged. In T2's scenario, the 1-st philosopher greedily gets the forks and eats forever. This counterexample is due to lack of strong fairness, i.e., the event $get.0.1$ is repeatedly enabled (after event $put.1.1$ and disabled after event $get.1.1$) but never engaged.

In order to verify that the system does satisfy the property under the assumption that the system is (strongly) fair and the deadlock situation never occurs, we may verify the following property,

$$\begin{array}{ll}
(\bigwedge_{i=0}^{N-1} \Box\Diamond get.i.(i+1)\%N) & \text{– C1} \\
\wedge\ \Box\Diamond put.1.2 & \text{– C2} \\
\Rightarrow\ \Box\Diamond eat.0 &
\end{array}$$

where C1 and C2 are stated as premises of the property. C1 states that each philosopher must always eventually get his first fork. C2 states that one of the philosopher (in this case, the 1-st) must eventually put down a fork. It is used to avoid the deadlock situation. Though this property is true, automata-based verification is deficient because of its size.

3 Event Annotations

In this section, we introduce a way of modeling event-based fairness, i.e., by annotating an event with fairness assumptions. Given an event e, four different annotations can be used to associate different fairness assumptions with e. The annotations are summarized in Table 1. In the following, we discuss them one by one.

A *weak* fair event is written as $wf(e)$. Event $wf(e)$ plays the same role as e except that it carries a weak fairness constraint. That is, if a weak fair event is always enabled, it must be eventually engaged. In other words, the system must move beyond a state where

Annotation	Name	Semantics
$wf(e)$	weak fair event	$\Diamond\Box e$ is enabled $\Rightarrow$ $\Box\Diamond e$ is engaged
$sf(e)$	strong fair event	$\Box\Diamond e$ is enabled $\Rightarrow$ $\Box\Diamond e$ is engaged
$wl(e)$	weak live event	$\Diamond\Box e$ is ready $\Rightarrow$ $\Box\Diamond e$ is engaged
$sl(e)$	strong live event	$\Box\Diamond e$ is ready $\Rightarrow$ $\Box\Diamond e$ is engaged

Table 1. Event-based Fairness Annotations

there is a weak fair event enabled. Weak fair events allow us to express weak fairness constraints naturally. It can be shown that both weak and strong fairness are expressible using weak fair events (as strong fairness can be transformed to weak fairness by paying the price of one variable [16]). However, strong fairness constraints may require more than what fair events can offer in a natural way. Therefore, we introduce the notion of *strong* fair events to capture strong fairness elegantly. A *strong* fair event, written as $sf(e)$, must be engaged if it is repeatedly enabled.

Example 2. The following demonstrates how we may achieve process level weak fairness (as the option offered in SPIN).

$$\begin{aligned}
fPhil(i) &= wf(get.i.(i+1)\%N) \rightarrow wf(get.i.i) \rightarrow wf(eat.i) \\
&\quad \rightarrow wf(put.i.(i+1)\%N) \rightarrow wf(put.i.i) \rightarrow fPhil(i) \\
fFork(i) &= wf(get.i.i) \rightarrow wf(put.i.i) \rightarrow fFork(i) \,\Box \\
&\quad\; wf(get.(i-1)\%N.i) \rightarrow wf(put.(i-1)\%N.i) \rightarrow fFork(i) \\
fCollege(N) &= \|_{i=1}^{N-1} (fPhil(i) \parallel fFork(i))
\end{aligned}$$

The idea is to annotate all events in a process weak fair so that an enabled event of the process is not ignored forever. Model checking $\Box\Diamond eat.0$ against $fCollege(5)$ may return T0 and T2 as counterexamples but not T1. □

Example 3. The following specifies the Peterson's algorithm for mutual exclusion. Without fairness assumptions, the algorithm allows unbounded overtaking, i.e., a process which intends to enter the critical section may be overtaken by other processes infinitely (refer to [1] for a concrete example).

$$\begin{aligned}
P(i,j) &= (sf(pos[i] := j) \rightarrow wf(step[j] := i) \rightarrow \\
&\qquad\quad [step[j] \neq i \vee \forall\, k \mid k \neq i \bullet pos[k] < j] \bullet P(i, j+1)) \\
&\quad \lhd\, j < N \,\rhd\, (wf(cs.i) \rightarrow wf(pos[i] := 0) \rightarrow P(i,1)) \\
Peterson(N) &= \|_{i=1}^{N} P(i,1)
\end{aligned}$$

where N is the number of processes and pos, $step$ are two lists of integers (with initial value 0) of size $N-1$ and N respectively. Infinite overtaking is evidenced by showing that $Peterson(N) \not\models \Box(pos[i] > 0 \Rightarrow \Diamond cs.i)$. Once a process has indicated that its intention to enter the critical section (by setting $pos[i]$ and $step[j]$), the assignment $pos[i] := j$ may be enabled only repeatedly. This is because it depends on the condition $[step[j] \neq i \vee \forall\, k \mid k \neq i \bullet pos[k] < j]$. Because the assignment $step[j] :=$

i is not synchronized or guarded, weak fairness is sufficient to guarantee it will be engaged once enabled. The weak fairness associated with $cs.i$ and $pos[i] := 0$ prevents the system from idling forever. Notice that this is not necessary if we assume that the system shall never idle forever unless it is deadlocked. The above model guarantees that $Peterson(N) \vDash \Box(pos[i] > 0 \Rightarrow \Diamond cs.i)$. □

In order to guarantee a system is completely strongly fair, communicating events or events guarded by conditions must be annotated with strong fairness, whereas weak fairness is sufficient for local actions which are not guarded. Weak/strong fairness annotation allows us to model event-based fairness flexibly. In practice, even stronger fairness may be necessary. One example of a fairness constraint which is very strong is the notion of accepting states in Büchi automata, i.e., the system must keep moving until entering at least one accepting state (and do that infinitely often). Other examples of stronger fairness include the compassion conditions [16]. In order to capture these fairness constraints, we introduce two additional fairness annotations, which have the capability of driving the system to reach certain point. The additional annotations relies on the concept of "readiness" so that system behaviors may be restricted by fairness assumptions which are associated with events that are not even enabled.

Definition 3 (Readiness). *Let P be a process. Let V be a valuation of the variables.*

$$\begin{array}{lll}
ready(Stop, V) & = ready(Skip, V) = \varnothing & \\
ready(e \rightarrow P, V) & = \{e\} & \\
ready(Skip;\ Q, V) & = ready(Q, V) & \\
ready(P;\ Q, V) & = ready(P, V) & \text{– if } P \neq Skip. \\
ready(P \Box Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \sqcap Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \bigtriangleup Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \lhd b \rhd Q, V) & = ready(P, V) & \text{– if } V \vDash b. \\
ready(P \lhd b \rhd Q, V) & = ready(Q, V) & \text{– if } V \vDash \neg b. \\
ready([b] \bullet P, V) & = ready(P, V) & \text{– if } V \vDash b. \\
ready([b] \bullet P, V) & = \varnothing & \text{– if } V \nvDash b. \\
ready(P \,|[\, X \,]|\, Q, V) & = ready(P, V) \cup ready(Q, V) &
\end{array}$$

Event e is *ready* given process P and valuation V if and only if $e \in ready(P, V)$. Note that *enabledness* and *readiness* are similarly defined for all process expressions except parallel composition. The difference is captured by the last line of the above definition. Given process $P \,|[\, X \,]|\, Q$, an event in X is enabled if and only if it is enabled in both P and Q, whereas it is ready if it is ready in either P or Q. Intuitively, an event is *ready* if and only if one thread of control is ready to engage in it. An *enabled* event must be *ready*. A *weak* live event, written as $wl(e)$, must be engaged if it is always ready (not necessarily enabled). Similarly, a *strong* live event, written as $sl(e)$, must be engaged if it is repeated ready[3]. Because whether an event is ready or not depends on only one process (in a parallel composition), live events may be used to design a controller which drives the execution of a given system.

[3] A similar modeling concept is hot locations in Live Sequence Charts [8], which force the system to move beyond.

Example 4. Let $\mathit{LiftSystem}$ be the modeling of a multi-lift system, which contains two events $\mathit{turn\_on\_light}$ and $\mathit{turn\_off\_light}$. In order to model that the light is always eventually turned off, the $\mathit{LiftSystem}$ may be replaced by $\mathit{LightSystem} \parallel \mathit{LightCon}$ where $\mathit{LightCon} = \mathit{turn\_on\_light} \rightarrow wl(\mathit{turn\_off\_light}) \rightarrow \mathit{LightCon}$. Because both events must be synchronized, whenever event $\mathit{turn\_on\_light}$ is engaged, event $\mathit{turn\_off\_light}$ becomes ready. In this case, it remains ready until it is engaged. Thus, by definition, the light must eventually be turned off. □

Example 5. With live events, the dining philosophers may be modified as follows,

$$\begin{aligned}
\mathit{lPhil}(i) &= wl(get.i.(i+1)\%N) \rightarrow get.i.i \rightarrow eat.i \\
&\quad \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \rightarrow \mathit{lPhil}(i) \\
\mathit{lFork}(i) &= get.i.i \rightarrow wl(put.i.i) \rightarrow \mathit{lFork}(i) \;\Box \\
&\quad get.(i-1)\%N.i \rightarrow wl(put.(i-1)\%N.i) \rightarrow \mathit{lFork}(i) \\
\mathit{lCollege}(N) &= \Big\|_{i=1}^{N-1} (\mathit{lPhil}(i) \parallel \mathit{lFork}(i))
\end{aligned}$$

Model checking $\Box\Diamond eat.0$ against $\mathit{lCollege}(5)$ returns true. Initially, $wl(get.i.(i+1)\%N)$ is ready and therefore by definition, it must be engaged (since it is not possible to make it not ready). Once $get.i.(i{+}1)\%N$ is engaged, $wl(put.(i{-}1)\%N.i)$ becomes ready and thus the system is forced to execute until it is engaged. For the same reason, $wl(put.i.i)$ must be engaged afterwards. Once $put.i.i$ is engaged, $wl(get.i.(i{+}1)\%N)$ becomes ready again. Therefore, the system is forced to execute infinitely and fairly. The traces which lead to the deadlock state is not returned as a counterexample. This is because event $wl(put.(i-1)\%N.i)$ is ready in the deadlock state. Hence the trace is considered invalid because it does not satisfied the fairness assumption, i.e., the event $wl(put.(i{-}1)\%N.i)$ is always ready but never engaged. Refer to Example 6 for further explanation. □

The fairness annotations restrict the possible behaviors of the system. It thus results in a smaller set of traces. Note that fairness constraints cannot be captured using structural operational semantics. Therefore, a two-levels semantics is used to prune un-fair traces from infinite traces. Let Σ_{wf}, Σ_{sf}, Σ_{wl} and Σ_{sl} be the set of all weak fair, strong fair, weak live and strong live events respectively.

Definition 4 (Fair Traces). *Let P be a process. Let V be a valuation of the data variables. The set of fair traces is written as $\mathit{ftraces}(P, V)$. An infinite sequence of events $\tilde{tr} : \Sigma^\omega$ is in $\mathit{ftraces}(P, V)$ if and only if there exists an infinite sequence of $\tilde{P}$ and $\tilde{V}$ such that*

- *$\tilde{tr}$ is in $\mathit{inftr}(P, V)$,*
- *for all i, if there exists $e : \Sigma_{wf}$ such that e is* enabled *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or e is not* enabled *at state $(\tilde{P}(j), \tilde{V}(j))$.*
- *for all i, if there exists $e : \Sigma_{sf}$ such that e is* enabled *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or for all k such that $k \geq j$, e is not* enabled *at state $(\tilde{P}(k), \tilde{V}(k))$.*
- *for all i, if there exists $e : \Sigma_{wl}$ such that e is* ready *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or e is not* ready *at state $(\tilde{P}(j), \tilde{V}(j))$.*

– *for all i, if there exists $e : \Sigma_{sl}$ such that e is* ready *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or for all k such that $k \geq j$, e is not* ready *at state $(\tilde{P}(k), \tilde{V}(k))$.*

Compared to Definition 1, the additional constraint states that an infinite trace must be *fair*. That is, whenever a weak fair (live) event is enabled (ready), later it must be either engaged or become not enabled (ready); whenever a strong fair (live) event is enabled (ready), either it becomes not enabled (ready) forever after some execution or it is eventually engaged. By definition, all traces in $\mathit{ftrace}(P, V)$ satisfy the fairness constraints regarding the annotated events (see proof in [28]). Compared with previous proposals [6, 7, 3], our notion of fair events is more flexible and natural. For example, in [3] fairness constraints only concern parallel composition, whereas in our setting fairness concerns individual events and thus not only parallel composition but also choice and others. For instance, the process $P = sl(a) \rightarrow b \rightarrow P \,\Box\, sl(b) \rightarrow a \rightarrow P$ requires that the choice must not be completely biased, i.e., both choices must eventually be taken.

CSP algebraic laws [14] are largely preserved in our extended semantics, e.g., the symmetry and associativity laws of parallel composition. Nevertheless, a few do not apply any more because of the weak/strong live events, e.g., a new expansion laws for parallel composition is needed (refer to [28]). Moreover, the fairness might be overwhelming so that the specification may become infeasible. For instance, given $P = wl(e) \rightarrow P$, $\mathit{ftraces}(P \parallel (e \rightarrow \mathit{College}(5)), \varnothing) = \varnothing$. This specification is not feasible because the fairness constraint can never be satisfied, i.e., event e can be engaged only once. This boils down to the question on how to effectively verify a model under the embedded fairness.

4 Verification

In this section, we show that existing state-of-the-art model checking algorithms may be extended to handle our notion of event-based fairness with little computational overhead. We define the notion of feasibility and then present an algorithm for feasibility checking. A specification is feasible if it allows at least one infinite trace. Given a process and a valuation of the data variables, a feasibility checking checks whether there exists an infinite trace which satisfies the fairness constraints. The same algorithm is used for LTL model checking. The product of the model and the Büchi automaton generated from negation of the property is feasible if only and if the property is not true. For simplicity, we assume the number of system states is always finite, i.e., the domains of the variables are finite and the process specifies regular languages.

Definition 5 (State Graph). *Let P_0 be a process. Let V_0 be the valuation of the variables. A state graph $G_{(P_0, V_0)}$ is (S, s_0, E) where S is a set of system states of the form (e, P, V); s_0 is the initial state $(\mathit{init}, P_0, V_0)$ where init is the event of system initialization; and E is a set of edges such that $((e, P, V), (e', P', V')) \in E \Leftrightarrow (P, V) \overset{e'}{\Rightarrow} (P', V')$.*

Without loss of generality, the just-engaged events are stored as part of the state information instead of transition labels, which turns a labeled transition system to a directed

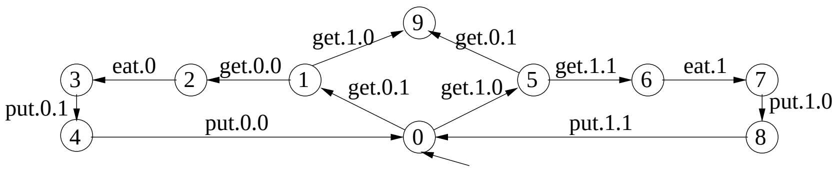

Fig. 1. LTS for 2 Dining Philosophers

graph. A run of $G_{(P,V)}$ is an infinite sequence of vertices following the edges. It is straightforward to show that for all tr such that $tr \in \mathit{inftr}(P, V)$ if and only if there is a corresponding run in $G_{(P,V)}$. There is a loop in $G_{(P,V)}$ if and only if a vertex is reachable from the initial state and itself.

Definition 6 (Fair Loop). *Let P_0 be a process. Let V_0 be a valuation of the data variables. Let $\langle (a_i, P_i, V_i), (a_{i+1}, P_{i+1}, V_{i+1}), \cdots, (a_j, P_j, V_j), (a_i, P_i, V_i) \rangle$ where $j \geq i$ be a loop in $G_{(P_0,V_0)}$. Let $\mathit{Engaged} = \{a_k \mid i \leq k \leq j\}$ be the set of engaged events during the loop. The loop is* fair *if and only if the following are satisfied,*

– $\bigcap_{k=i}^{j}(\mathit{enabled}(P_k, V_k) \cap \Sigma_{wf}) \subseteq \mathit{Engaged}$
– $\bigcup_{k=i}^{j}(\mathit{enabled}(P_k, V_k) \cap \Sigma_{sf}) \subseteq \mathit{Engaged}$
– $\bigcap_{k=i}^{j}(\mathit{ready}(P_k, V_k) \cap \Sigma_{wl}) \subseteq \mathit{Engaged}$
– $\bigcup_{k=i}^{j}(\mathit{ready}(P_k, V_k) \cap \Sigma_{sl}) \subseteq \mathit{Engaged}$

The set $\bigcap_{k=i}^{j}(\mathit{enabled}(P_k, V_k) \cap \Sigma_{wf})$ contains the weak fair events which are always enabled during the loop. Similarly, $\bigcap_{k=i}^{j}(\mathit{ready}(P_k, V_k) \cap \Sigma_{wf})$ is the set of weak live events which are always ready during the loop. The set $\bigcup_{k=i}^{j}(\mathit{enabled}(P_k, V_k) \cap \Sigma_{sf})$ contains the strong fair events which are enabled once during the loop. Similarly, $\bigcup_{k=i}^{j}(\mathit{ready}(P_k, V_k) \cap \Sigma_{sl})$ is the set of strong live events which are ready once during the loop. A loop is fair if and only if,

– all always-enabled weak fair events are engaged,
– all once-enabled strong fair events are engaged,
– all always-ready weak live events are engaged,
– all once-enabled strong live events are engaged.

A loop may contain the same state more than once, e.g., states 0,1,2,3,4,0,5,6,7,8,0 in Figure 1 forms a loop. It is straightforward to prove that a specification is feasible if and only if the state graph contains a fair loop. Feasibility checking is thus reduced to find a fair loop if possible. Equivalently, we can show that a specification is feasible if and only if the graph contains a fair strongly connected component (SCC) [16]. A strongly connected subgraph is fair if and only if the loop which visits every vertex in the subgraph is fair.

Example 6. Figure 1 shows the labeled transition system generated from $\mathit{lCollege}(2)$ (presented in Example 5). The loop containing state 0,1,2,3,4 is not fair because event

$get.1.0$, which is annotated weak live, is always ready during the loop (though not always enabled, Definition 3). Similarly, the loop containing states 0,5,6,7,8 is not fair neither because $get.0.1$ is always ready. Note that the deadlock state 9 is considered as a trivial loop. It is not fair because both $put.0.1$ and $put.1.0$, which are annotated weak live, are ready (and therefore trivially always ready during the loop). The loop containing 0,1,2,3,4,5,6,7,8 (which constitute an SCC) is fair. Note that this loop satisfies the property $\Box\Diamond eat.0$. $\Box$

4.1 On-the-fly Verification

In literature, there are two sets of algorithms for identifying a loop or equivalently checking the emptiness of Büchi automata, namely, ones based on nested depth-first-search and ones based on SCC (refer to the survey in [26]). We present here an SCC-based algorithm which extends the one presented in [12].

The problem of LTL model checking with event-based fairness is to verify whether every fair trace of the model satisfies a given LTL formula ϕ. Or equivalently, let $B_{\neg\phi}$ be a Büchi automaton which is equivalent to the negation of ϕ, the model violates ϕ if and only if there is a *fair* SCC in the synchronized product of $G_{(P,V)}$ and $B_{\neg\phi}$ which contains at least one accepting state from the Büchi automaton. In [16], a backward searching algorithm is used to identify all fair maximum SCCs if there is any. Because the query language of PAT is based on LTL (extended with events), we developed an on-the-fly approach based on Tarjan's algorithm for finding one maximum SCC. The idea is to search for maximum SCCs while building the state graph. If the found one is not fair, a set of 'bad' states are pruned and then the SCC is decomposed into smaller SCCs. Whenever a fair SCC is found, we proceed to produce a counterexample.

Figure 2 shows the detailed algorithm. The inputs are a process P_0, a valuation of the data variables V_0 and an initial state of the Büchi automaton S_0. Note that S_0 is skipped for feasibility checking. The main loop from line 3 to 25 is based on an iterative improved version of Tarjan's algorithm (refer to [20, 12] for details). Stack $working$ holds all states that are yet to explored and $stack$ holds states which may be part of an SCC. At line 6 (and 11), a subset of the enabled actions (i.e., **ample**(P, V, S) which will be explained in detail in Section 4.2) is expanded. In order to conclude as soon as possible, a simple procedure is added at line 8 to check whether the found loop is fair. Experiences show that although this procedure becomes overhead for true properties, it may produce a counterexample early if there is any. This is particularly true for models which are strongly connected. A maximum SCC is discovered once the condition at line 17 is satisfied. Line 18 to 20 collect states of the SCC from *stack*. If the found SCC contains a Büchi accepting state (as a component of one state in scc), i.e., satisfying the condition at line 21, and if scc is fair and nontrivial (i.e., with at least one transition), the algorithm returns false after producing a counterexample (refer to algorithms presented in [16] on how to produce counterexamples). If the SCC is not fair, a set of bad states is removed from scc (line 23). A bad state carries a fairness constraint which can not be fulfilled by any loop formed by states in the SCC. A node (e, P, V) in scc is bad, i.e., $(e, P, V) \in bad(scc)$, if and only if one of the following conditions is satisfied,

```
1. $preorder, lowlink, found := \varnothing;\ stack;\ done := 1;\ i := 0;$
2. $working := \langle (Init, P_0, V_0, S_0) \rangle;$
3. while $working \neq \langle\rangle$
4.     $v = (a, P, V, S) := working.peek();$
5.     if $preorder(v) = null$ then $preorder[v] := i{+}{+};$
6.     foreach $v' \in$ ample$(P, V, S)$
7.         if $preorder(v') = null$ then $working.push(v');\ done := 0;$ break;
8.         else early-fair-loop-detection
9.     if $done = 1$
10.        $lowlink[v] := preorder[v];$
11.        foreach $w \in$ ample$(P, V, S)$
12.            if $w \notin found$
13.                if $preorder[w] > preorder[v]$
14.                    $lowlink[v] := min(lowlink[v], lowlink[w]);$
15.                else $lowlink[v] := min(lowlink[v], preorder[w]);$
16.            $working.pop();$
17.            if $lowlink[v] = preorder[v]$
18.                $found.add(v);\ scc := \{v\};$
19.                while $stack \neq \langle\rangle \wedge preorder[stack.peek()] > preorder[v]$
20.                    $k := stack.pop();\ found.add(k);\ scc.add(k);$
21.                if $scc$ is Büchi-fair
22.                    if $scc$ is fair and nontrivial then return $false;$
23.                    if not $OntheflyMC2(scc \setminus bad(scc))$ then return $false;$
24.            else $stack.push(v);$
25. return $true;$
```

Fig. 2. On-the-fly Model Checking Algorithm: $OnTheFlyMC1$

- there exists $x \in \Sigma_{wf}$ such that $x \in enabled(P, V)$ and there does *not* exist a state (e', P', V') in scc such that $e' = x$ or $x \notin enabled(P', V')$. That is, x is *always* enabled but never engaged.
- there exists $x \in \Sigma_{sf}$ such that $x \in enabled(P, V)$ and there does *not* exist a state (x, P', V') in scc. That is, x is enabled but never engaged.
- there exists $x \in \Sigma_{wl}$ such that $x \in ready(P, V)$ and there does *not* exist a state (e', P', V') in scc such that $e' = x$ or $x \notin ready(P', V')$. That is, x is *always* ready but never engaged.
- there exists $x \in \Sigma_{sl}$ such that $x \in ready(P, V)$ and there does *not* exist a state (x, P', V') in scc. That is, x is ready in SCC but never engaged.

Then algorithm $OnTheFlyMC2$ is invoked at line 23. The logic of $OnTheFlyMC2$ is the same as $OnTheFlyMC1$ except that it only searches for maximum SCCs within the given states (and transitions which have been stored externally during $OnTheFlyMC1$). Refer to [28] for the details of $OnTheFlyMC2$. Whenever $OnTheFlyMC2$ returns false (i.e., a nontrivial fair SCC is found), we conclude a counterexample is found. If

$OnTheFlyMC2$ returns true (i.e., no fair SCC is found) or there is a weak fair/live event which is always ready/enabled but never engaged or a Büchi fairness condition is not fulfilled by scc, scc is abandoned and then we proceed to search for the next maximum SCC. The soundness of Algorithm 2 is presented in Appendix.

Example 7. Applying feasibility checking to $lCollege(2)$ (shown in Figure 1) would return two maximum SCCs, i.e., one containing state 9 only and one containing the rest. By definition, the one containing 9 is not fair (as discussed) and state 9 is a bad state. Removing state 9 from the SCC results in an empty set and thus it is abandoned. The other SCC is fair and therefore the model is feasible. □

4.2 Partial Order Reduction

In the worst case, where the whole system is one SCC or the property is true, Algorithm 2 constructs the complete state graph and suffers from state space explosion. We thus apply partial order reduction to solve the problem. The idea is to only construct a *reduced* graph (in contrast to the complete graph in Definition 5) which is equivalent to the complete one with respect to the given property. This is realized by exploring only a subset of the enabled transitions at line 6 of algorithm 2.

Partial order reduction has been explored for almost two decades now. There have been theoretical works on partial order reduction under fairness constraints [22] in a different setting. In our setting, not only the reduction shall respect the property but also the fairness constraints. That is, a fair loop must be present in the reduced graph if there is one in the complete graph. In the *reduced* graph, for every node, only a subset of the enabled synchronized (outgoing) transition of the model and the Büchi automaton is explored. In particular, given a state (a, P, V, S) where a is the just engaged event, P is the current process expression, V is the current valuation and S is the current state of the Büchi automaton, a successor node (a', P', V', S') is explored, written as $(a', P', V', S') \in$ **ample**(P, V, S), if and only if the following conditions are satisfied,

- (a', P', V', S') is a successor in the complete graph, i.e., $(P, V) \overset{a'}{\Rightarrow} (P', V')$ and the transition is allowed by the Büchi automaton, i.e., (S, a', S') is a transition in $B_{\neg\phi}$ and the condition which guards the transition is true. Note that the Büchi automata are transition-labeled for efficient reasons.
- the successors must satisfy a set of additional conditions for property-preserving partial order reduction, which is denoted as $(P', V') \in ample(P, V)$.

The algorithm presented in Figure 3 has been implemented in PAT to produce small but sound $ample(P, V)$, which extends the one proposed in [11] to handle event annotations.

If P is not an indexed parallel composition (or indexed interleaving), the node is fully expanded. Otherwise, we identify one process which satisfies a variety of conditions and expand the node with only enabled events from that process. Notice that $enabled_{P_i}(P, V)$ is $enabled(P, V) \cap enabled(P_i, V)$, e.g., the set of globally enabled events which P_i participates in. $current(P_i)$ is the set of actions that could be enabled given P_i and a cooperative environment. For instance, a guarded event is in the set even

```
1. **if** $P$ is of the form $P_1 \parallel P_2 \parallel \cdots \parallel P_n$
1.     **foreach** $i$ such that $1 \leq i \leq n$
2.         **if** $enabled_{P_i}(P, V) = current(P_i)$
3.             $ample := true;$
4.             $ampleset := \varnothing;$
5.             **foreach** $e, P', V'$ s.t. $e \in enabled_{P_i}(P, V)$ and $(P, V) \overset{e}{\Rightarrow} (P', V')$
6.                 **if** $\underline{visible(e)} \vee on\_stack(P', V') \vee \exists\, e' : \Sigma_{P_j} \mid i \neq j \bullet dep(e, e')$
7.                     $ample := false;$ **break**;
8.                 **else**
9.                     $ampleset := ampleset \cup \{(e, P', V')\};$
10.                **endif**
11.            **endfor**
12.            **if** $ample$ **then return** $ampleset;$
13.        **endif**
14.    **endfor**
15. **endif**
16. **return** $\{(e, P', V') \mid e \in enabled(P, V)$ and $(P, V) \overset{e}{\Rightarrow} (P', V')\};$
```

Fig. 3. Partial Order Reduction

if the guard condition is false. Two events are dependent, written as $dep(e, e')$ if they synchronize or write/read a shared variable (with at least one writing). Note that both $current(P_i)$ and $dep(e, e')$ are based on static information, which can be collected during compilation. A component P_i is chosen if and only if the following conditions are satisfied,

- $enabled_{P_i}(P, V) = current(P_i)$. No other events could be enabled given P_i. Refer to [11] for intuitions behind this condition.
- The condition $on\_stack(P', V')$ is true if and only if the state (P', V') is on the search stack (refer to [11]). Performing any event in $enabled_{P_i}(P, V)$ must not result in a state on the search stack. This is to prevent enabled actions from being ignored forever. Note that this condition can be removed for checking deadlock-freeness.
- The actions in $enabled_{P_i}(P, V)$ must be independent from transitions of other components, i.e., $\nexists e' : \Sigma_{P_j} \mid i \neq j \bullet dep(e, e')$. Refer to [11] for intuitions behind this condition.
- Different from the one in [11], an event is visible, i.e., written as $visible(e)$, if it is visible to a given property or the fairness annotations. Event e is visible to a property if and only if e constitutes the property, e.g., $eat.0$ is visible given property $\Box\Diamond eat.0$, or e updates a variable which constitutes the property. Event e is visible to the fairness annotations if and only if performing e may change the set of annotated events which are enabled or ready.

The soundness of the partial order reduction with respect to next-free LTL and event-based fairness is proved in Appendix. Note that the above realizes only one of the possi-

Model	Property	without fairness			with event-based fairness		
		result.	w/o red.	with red.	result.	w/o red.	with red.
$College(7)$	$\Box\Diamond eat0$	No	< 1	< 1	Yes	10.3	11.3
$College(9)$	$\Box\Diamond eat0$	No	< 1	< 1	Yes	469.1	504.4
$College(11)$	$\Box\Diamond eat0$	No	4.2	< 1	Yes	–	–
$College(13)$	$\Box\Diamond eat0$	No	25.6	< 1	Yes	–	–
$College(15)$	$\Box\Diamond eat0$	No	–	< 1	Yes	–	–
$Milner\_Cyclic(10)$	$\Box\Diamond work0$	Yes	17.8	< 1	Yes	17.7	< 1
$Milner\_Cyclic(12)$	$\Box\Diamond work0$	Yes	322.9	< 1	Yes	283.3	< 1
$Milner\_Cyclic(100)$	$\Box\Diamond work0$	Yes	–	3.3	Yes	–	3.4
$Milner\_Cyclic(200)$	$\Box\Diamond work0$	Yes	–	17.6	Yes	–	18.1
$Milner\_Cyclic(400)$	$\Box\Diamond work0$	Yes	–	118.4	Yes	–	119.2
$ReadersWriters(100)$	$\Box !error$	Yes	–	4.3	Yes	–	3.9
$ReadersWriters(200)$	$\Box !error$	Yes	–	37.3	Yes	–	29.1
$ReadersWriters(400)$	$\Box !error$	Yes	–	251.3	Yes	–	257.1

Table 2. Experiment Results

ble heuristics for partial order reduction, which we believe is cost-effective. Achieving the maximum reduction is in general computational expensive.

Besides partial order reduction, we have also implemented optimizations based on CSP's algebraic laws. For instance, in order to handle systems with large number of identical or similar processes, efficient procedures are applied to sort the processes of a parallel or interleaving composition. The soundness is proved by the symmetry and associativity of indexed interleaving and parallel composition.

4.3 Experiments

In this part, we evaluate the algorithm and the effectiveness of the reductions using benchmarks systems. Table 2 presents a part of the experimental results. The experiments are conducted on Windows XP with a 2.0 GHz Intel CPU and 1 GB memory.

The first model is the dining philosophers. The property is $\Box\Diamond eat.0$. Without fairness assumptions, this property is false. A counterexample is quickly produced in most of the cases. Nonetheless, it may take considerably long if the trace leading to a counterexample is explored very late (e.g., for $College(15)$ without reduction). Partial order reduction significantly reduces the time to discover a counterexample for this example. This model is then annotated with fairness, as shown in Example 5. The last three columns show verification results of $lCollege(N)$. The property becomes true and therefore a complete search is necessary. Note that partial order reduction gains little. The reason is that the model is highly coupled and heavy in communication. Manually hiding local communicating could reduce the verification time, as shown in [24].

The second model is Milner's cyclic scheduler. Milner's cyclic scheduler describes a scheduler for N concurrent processes. The processes are scheduled in cyclic fashion so that the first process is reactivated after the N-th process has been activated. The fairness assumptions state that a process must eventually finish its local task and then activate the next process. The property to verify is that a process must eventually be scheduled, which is true with/without the fairness assumptions. This model demonstrates the effectiveness of the partial order reduction. Without the reduction, the size of the search graph grows exponentially and thus verification soon becomes infeasible (e.g., for 15 processes). With partial order reduction, we are able to verify 400 processes reasonably fast (using less than 2 minutes). Notice that the computational overhead for handling fairness annotations are negligible, e.g., same amount of time is taken to verify the model with/without fairness. The third models the classic readers/writers problem. The readers/writers model describes a protocol for coordination of N identical readers and N identical writers accessing a shared resource. The property to verify is reachability of an erroneous situation (i.e., wrong readers/writers coordination). This mode is then annotated with fairness assumptions to state that each reader/writer must eventually finish reading/writing. This model demonstrates the effectiveness of the reduction for handling identical/similar processes.

Details of the models and more experiments are available online [28]. Compared to existing tools, PAT complements the CSP model checker FDR in serval aspects. Namely, PAT supports verification under fairness assumptions and temporal logic based verification. For common features like deadlock-freeness checking, PAT outperforms FDR sometimes because we use a completely on-the-fly checking strategy (refer to the results at our web site). Compared to SPIN, PAT is not yet as efficient for systems with no event-based fairness and small LTL properties. PAT offers a more flexible way of modeling fairness and verifying under fairness assumptions (than SPIN's option for process-level weak fairness). PAT differs from SPIN in two aspects. Firstly, because we are dealing with an event-based formalism, we extend LTL with events so that properties concerning both states and events can be stated and verified. Secondly, because fairness constraints have been embedded in the specification, the size of the property is reduced and thus model checking under fairness is carried out efficiently.

5 Conclusion and Future Works

In this work, we presented an approach to systematically model a variety of fairness in event-based compositional systems. We also developed algorithms to efficiently verify systems under fairness assumptions. A toolset named PAT has been developed for specification and verification of event-based fairness enhanced systems. Our experiments show clear advantage over the common practise of assuming a fair scheduler and then proving liveness properties over safety-centric specifications.

As for future works, there are a number directions to go in terms of tool development. We are currently adding more language features, e.g., arrays, broadcasting messages, etc. We are exploring how to extend event-based fairness and its verification to languages like C# or Java. PAT is not yet as efficient for systems with no event-based fairness and small LTL properties. Optimization techniques like symmetry reduction need to be studied under fairness and incorporated. One future work of theoretical interests is to study the notion of process refinement/equivalence for fairness enhanced processes. Because of fairness constraints, trace refinement becomes a stronger notion. Namely, a fair branching may not be removed in a refined process without introducing new traces. In this work, we choose not to prevent inputs from being marked *fair*. Marking inputs from an open channel *fair* restricts the behaviors of the environment, which could be largely undesirable. Nevertheless, assuming fair/live environments would help effective model checking of open systems and synthesis (e.g., [27]). One future work is to investigate verification/synthesis of open systems under fairness assumptions.

Acknowledgement

Jun Pang, Yuxin Deng and anonymous referees provided helpful comments on early drafts of this paper. This work is partially supported by the research grant titled "Sensor Networks Specification and Validation" (R-252-000-320-112) funded by Ministry of Education, Singapore.

References

1. K. Alagarsamy. Some Myths About Famous Mutual Exclusion Algorithms. *SIGACT News*, 34(3):94–103, 2003.
2. K. R. Apt, N. Francez, and S. Katz. Appraising Fairness in Languages for Distributed Programming. *Distributed Computing*, 2:226–241, 1988.
3. S. D. Brookes. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In *Proc. of the 13th Inte. Conf. on ConcCTheory (CONCUR 2002)*, pages 466–482, 2002.
4. S. D. Brookes, A. W. Roscoe, and D. J. Walker. An Operational Semantics for CSP. Technical report, 1986.
5. S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *Proc. of the 4th Inter. Conf. on Integrated Formal Methods (IFM 2004)*, pages 128–147, 2004.
6. G. Costa and C. Stirling. Weak and Strong Fairness in CCS. In *Mathematical Foundations of Computer Science (MFCS 1984)*, pages 245–254, 1984.
7. J. F. Costa and A. Sernadas. Progress Assumption in Concurrent Systems. *Formal Aspects of Computing*, 7(1):18–36, 1995.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
9. J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *Proceeding od the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 342–359. Springer, 2006.
10. J. Song Dong, P. Hao, S. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *Lecture Notes in Computer Science*, pages 483–498. Springer, 2004.
11. O. Grumberg E. M. Clarke and D. A. Peled. *Model Checking*. The MIT Press, 2000.
12. J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoritical Computer Science*, 345(1):60–82, 2005.
13. M. R. Henzinger and J. A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *Proc. of the 5th Scandinavian Workshop on Algorithm Theory (SWAT '96)*, pages 16–27, 1996.
14. C. A. R. Hoare. *Communicating Sequential Processes*. Inte. Series in Computer Science. Prentice-Hall, 1985.

15. G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engeering*, 23(5):279–295, 1997.
16. Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model Checking with Strong Fairness. *Formal Methods and System Design*, 28(1):57–84, 2006.
17. L. Lamport. Fairness and Hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.
18. T. Latvala and K. Heljanko. Coping with Strong Fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.
19. Y. Liu, J. Sun, and J. S. Dong. An Analyzer for Extended Compositional Process Algebras. In *30th International Conference on Software Engineering (ICSE 2008) Companion Volume*, pages 919–920. ACM, 2008.
20. E. Nuutila and E. Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters*, 49(1):9–14, 1994.
21. S. Older. Strong Fairness and Full Abstraction for Communicating Processes. *Information and Computation*, 163(2):471–509, 2000.
22. D. Peled. Ten Years of Partial Order Reduction. In *Proc. of the 10th Inte. Conf. on Computer Aided Verification (CAV'98)*, pages 17–28, 1998.
23. A. Puhakka and A. Valmari. Liveness and Fairness in Process-Algebraic Verification. In *Proc. of the 12th Inte. Conf. on Concurrency Theory (CONCUR 2001)*, pages 202–217, 2001.
24. A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Proc. of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 95)*, pages 133–152, 1995.
25. S. Schneider. *Concurrent and Real-time Systems: the CSP Approach*. John Wiley and Sons, LTD, 2000.
26. S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *Proc. of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 174–190, 2005.
27. J. Sun and J. S. Dong. Design Synthesis from Interaction and State-Based Specifications. *IEEE Transactions on Software Engineering*, 32(6):349–364, 2006.
28. J. Sun, Y. Liu, J. S. Dong, and H. Wang. The Process Analysis Toolset PAT. Technical report. http://www.comp.nus.edu.sg/~sunj/pat.pdf.

Appendix: Soundness Proofs

Theorem 1. *Let P be a process and V be a valuation of the variables. Let ϕ be a next-free LTL formula (with events). $(P,\, V) \vDash \phi$ if and only if Algorithm 2 returns true.*

Proof: By a standard proof we can show that $(P,\, V) \vDash \phi$ if and only if there does not exist an infinite path of $G_{(P,V)} \times B_{\neg\,\phi}$ which is fair with respect to $G_{(P,V)}$ and is accepting to $B_{\neg\,\phi}$. Equivalently, $(P,\, V) \nvDash \phi$ if and only if there is a fair loop (since we assume $G_{(P,V)}$ is finite) in $G_{(P,V)} \times B_{\neg\,\phi}$ which is also accepting. Equivalently, $(P,\, V) \nvDash \phi$ if and only if there is a fair SCC which is also accepting to $B_{\neg\,\phi}$. To prove the above theorem, we thus need to show that if the algorithm returns false if and only if there is such an SCC. If there exists such an SCC (say scc), there must be one maximum SCC (say SCC) which contains scc. By the soundness of Tarjan's algorithm, SCC must be discovered by line 21. If scc is SCC, the algorithm returns false as we shall prove. Otherwise, because scc contains no bad states (by Definition 6 and the definition of bad

states on page 13), all states of scc are not pruned. By induction we conclude either a fair and accepting SCC which contains all states of scc is found or scc is found. In both cases, the algorithm returns false. Thus, the algorithm returns false if there is a fair SCC which is also accepting. Ergo, it returns false if $(P,\, V) \not\vDash \phi$. It is straightforward to prove that the algorithm returns false, either at line 8 (a fair loop which is also accepting is found) or line 22 (the maximum SCC found is fair and accepting) or line 23 (a sub-SCC is), only if such an SCC is found. The algorithm is terminating because the number of $States$ is finite (by assumption) and i is monotonically increasing (i.e., the number of visited states). The 'recursive' call at line 23 is terminating because the number of states in scc is monotonically decreasing (i.e., $bad(scc)$ is not empty by definition). Therefore, we conclude the algorithm returns true only if $(P,\, V) \vDash \phi$. □

Theorem 2. *Let P be a process. Let V be the valuation of the global variables. Let $R_{(P,V)}$ be the reduced graph constructed by expanding each node with only events returned by the Algorithm (shown in Figure 3). Then, for every next-free LTL formula ϕ, $G_{(P,V)}, s_0 \vDash \phi$ if and only if $R_{(P,V)}, s_0 \vDash \phi$.*

Proof: For simplicity, we only prove the case for weak live events, i.e., given the weak live annotations, there is a fair loop in the reduced graph if and only if there is one in the complete graph. Strong live events can be transformed to weak live events at the cost of auxiliary variables as shown in [16]. Weak/strong fair events can be proved similarly.

For each weak live event $wl(a)$, we introduce an auxiliary variable x_a. P is modified to be P' in which x_a is set to 0 if it is not ready or just engaged or otherwise set to 1. ϕ is then modified to be ϕ' which is of the form $(\wedge_a\ \Box\Diamond x_a = 0) \Rightarrow \phi$ for each auxiliary variable x_a. We show that $(P,\, V) \vDash \phi$ if and only if $(P',\, V) \vDash \phi'$. For every loop in the given model, if it is fair with respect to the fairness constraints, then for every auxiliary variable x_a, the loop satisfies that $\Box\Diamond x_a = 0$ because $wl(a)$ can not be always ready during the loop and never engaged by Definition 6. Thus, if the fair loop satisfies ϕ, it satisfies ϕ'. The reverse is proved trivially. Thus, $(P,\, V) \vDash \phi$ if and only if $(P',\, V) \vDash \phi'$.

Next, we apply Theorem 12 in Chapter 10 of [11] to show that $G_{(P',V)}, s_0 \vDash \phi'$ if and only if $R_{(P',V)}, s_0 \vDash \phi'$. By apply similar arguments, we can show that the Algorithm satisfies the following conditions,

C0 $ample(P,\, V)$ is empty if and only if $enabled(P,\, V)$ is empty. This is trivial.
C1 Along every path in the full state graph that starts at s, a transition that is dependent on a transition in $ample(P,\, V)$ cannot be executed without a transition in $ample(s)$ occurring first. This is proved by the same argument in Section 10.5.2 in [11].
C2 If a node is not fully expanded, then every event in $ample(P,\, V)$ is invisible. This is guaranteed by line 6 and 7 in the algorithm presented in Figure 3, i.e., the condition $visible$, so that only events which preserves the valuation of propositions in ϕ and the auxiliary variables are presented in the ample set .
C3 There must be at least one node which is fully expanded along a cycle. This is guaranteed by condition $on\_stack(P',\, V')$ at line 6.

Thus, we conclude that $G_{(P',V)}, s_0 \vDash \phi'$ if and only if $R_{(P',V)}, s_0 \vDash \phi'$. By transitivity, we conclude that the theorem holds. □