Model Checking Conformance with Scenario-Based Specifications *

Marcelo Glusman and Shmuel Katz

Department of Computer Science
The Technion, Haifa 32000, Israel
{marce, katz}@cs.technion.ac.il

Abstract. Specifications that describe typical scenarios of operations have become common for software applications, using, for example, use-cases of UML. For a system to conform with such a specification, every execution sequence must be equivalent to one in which the specified scenarios occur sequentially, where we consider computations to be equivalent if they only differ in that independent operations may occur in a different order.
A general framework is presented to check the conformance of systems with such specifications using model checking. Given a model and additional information including a description of the scenarios and of the operations' independence, an augmented model using a transducer and temporal logic assertions for it are automatically defined and model checked. In the augmentation, a small *window* with part of the history of operations is added to the state variables. New transitions are defined that exchange the order of independent operations, and that identify and remove completed scenarios. If the model checker proves all the generated assertions, every computation is equivalent to some sequence of the specified scenarios. A new technique is presented that allows proving equivalence with a small fixed-size window in the presence of unbounded out-of-order of operations from unrelated scenarios. This key technique is based on the *prediction* of events, and the use of *anti-events* to guarantee that predicted events will actually occur. A prototype implementation based on Cadence SMV is described.

Keywords: Model checking, scenario, computation equivalence, transducer, anti-event

1 Introduction

It has become common to describe systems through a series of typical, canonic, scenarios of behavior, either through *use-cases* of UML or other formalisms. System computations in which such scenarios occur in sequence are called *convenient*. We consider two computations that differ only in the order in which independent operations are executed, as *equivalent* with respect to a scenario-based specification. A system conforms with such a specification if every computation is equivalent to a convenient one. The specifications of *serializability* of concurrency control algorithms for distributed databases and, in the area of hardware design, the *sequential consistency* of shared memory protocols have a similar structure.

* This work was partially supported by the Fund for the Support of Research at the Technion.

W.A. Hunt, Jr. and F. Somenzi (Eds.): CAV 2003, LNCS 2725, pp. 328–340, 2003.

The equivalence notion used in this paper ($\approx$) is based on a *conditional* independence relation among occurrences of operations: two operations may be independent in only some of the system states. The problem is, therefore, given a description of a fair transition system, a set of scenarios, and a conditional independence relation among operations, to prove that every computation of the system is equivalent to some concatenation of scenarios. We present a general framework for reducing this problem to one that can be solved automatically by a model checker. Based on the problem's inputs and additional heuristic information, we define an augmentation of the system – a *transducer* – and a set of LTL and CTL formulas that, if verified for the transducer, imply that the original system conforms with the scenario-based specification. As usual, this is possible when the system of interest is finite state, or can be abstracted to a finite state version in which model checking tools can be applied.

Given a fair transition system M, we build a transducer M' (See Fig.1) by composing M with a bounded *window* (a history queue) H of fixed length L, and an ω-automaton C (which we also call the *chopper*) that reads its input from H and accepts only the desired scenarios. In addition, M' has an *error* flag, initially false. When a (non-idling) transition from M is taken, if the history is not full and the error flag is false, the current state-transition pair is enqueued in H. Otherwise, the error flag is set to true, and never changed thereafter. When the error flag is true, there is no further interaction between M and H. Additional transitions of M' are *swaps* of consecutive elements of H (according to the user-defined conditional independence) and *chops* of prefixes of the window, that remove any desired scenario recognized by C. A run of M is defined as convenient if it belongs to the language of C, i.e., is composed of desired scenarios. Any infinite sequence built by concatenation of the chopped state-transition pairs is therefore a convenient run. We aim to prove that for *every* computation g of M there is a convenient

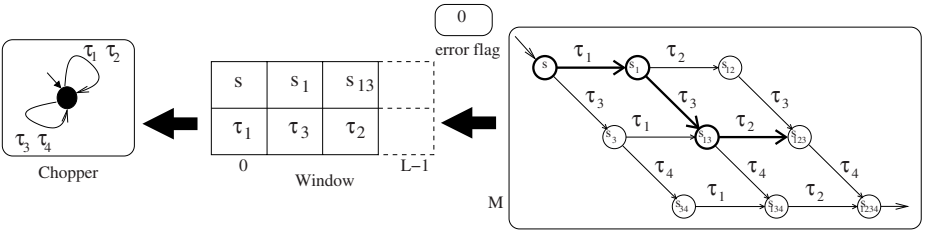

Fig. 1. Example of a transducer M' with two scenarios: $\tau_1\tau_2$ and $\tau_3\tau_4$. After swapping (s_1, τ_3) with (s_{13}, τ_2), scenario $\tau_1\tau_2$ can be chopped from the window.

computation c such that $c \approx g$. The transducer augments M without interfering with its operation. Therefore, for every computation g of M, there is some computation g' of M' such that g is the projection of g' to M's variables and transitions. If there is such a g' in which M's transitions are interleaved with swaps in H and transitions of C (that dequeue elements from H), so that H is never full when M makes a transition, then the error flag will never be raised in g'.

A bounded transducer cannot deal directly with events that may be delayed without bound, e.g., if they can be preceded by any number of independent events ("unbounded out-of-order"). Such a situation may arise in many concurrent systems, where the fairness constraints on the computations do not set any bound on how long a transition can be delayed before it is taken. If the window were *not* bounded, the transducer could wait until the needed event occurs, then swap it "back in time" until it reaches its place in a convenient prefix. A bounded window will necessarily become full in some computations, and the $error$ flag will be raised before the needed event occurs.

To overcome unbounded out-of-order, when part of a scenario has been identified a *prediction* can be made that a currently enabled event will eventually occur. In our novel prediction mechanism, the transducer then creates and inserts an event/*anti-event* pair after the existing part of a scenario being built in the window. After the predicted event is chopped together with the rest of the scenario, the anti-event remains in the window to keep track of the prediction. The anti-event can be swapped with the following events in the window, only if its corresponding event could have been swapped in the opposite direction. When the anti-event later meets the actual occurrence of its predicted event, they cancel out and disappear (see Fig.2). [1] For this method to work, we need to make sure

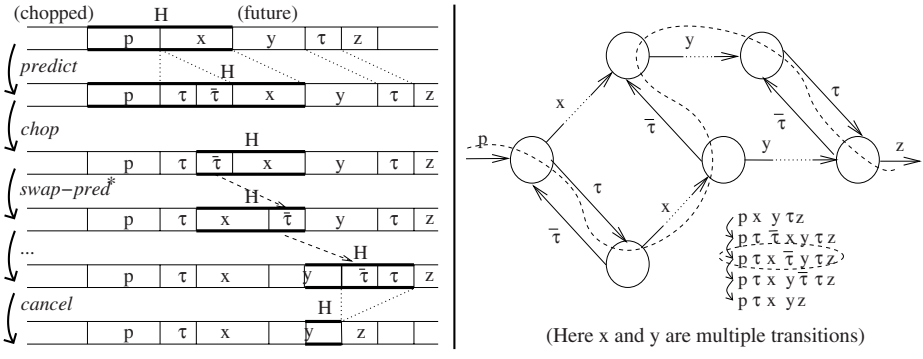

Fig. 2. Predicting τ to prove $pxy\tau z \equiv_{sw} p\tau xyz$.

every event/anti-event pair is inserted after any previously predicted event and before its corresponding anti-event. This method is analogous to the use of prophecy variables, but is easier to implement and understand.

The generated LTL and CTL proof obligations over the extended model verify that for every computation g of M there is a computation c of M' that "simulates" g, while avoiding raising the $error$ flag, fulfilling all the predictions, and in which no event remains forever in the window. They also require that it is always possible to postpone the initiation of new scenarios until the pending ones are completed. Together with a

[1] A nice analogy can be drawn between the behavior of our anti-events (and events) and that of particles (resp., antiparticles) in modern physics, where a positron can be seen as an electron travelling back in time.

restriction on the independence relation, needed to preserve fairness, the correctness of all these formulas for M' guarantees that M conforms with the scenario-based specification given by C.

To demonstrate the feasibility of our approach, we built a prototype tool (CNV) that implements our method. A description of the problem in CNV's language is used to generate automatically the transducer and the related formulas, both in the language of the model checker Cadence SMV [15], which is then used to verify them. In CNV, additional facilities (beyond prediction) are provided to reduce the size of the window, such as a means for defining a small abstraction of the state information that suffices to support the transducer's operation.

Related work: The Convenient Computations proof method [4, 5, 11] is based on showing that every computation of a system can be reduced or is equivalent to one in a specified subset called the *convenient* computations. These computations are easier to verify for properties that should hold for the entire system. In a proof, one must define which computations are convenient, prove that they satisfy the property of interest, and then extend this result by showing that a property-preserving reduction exists from every computation to a convenient one. This general approach is useful both for (potentially infinite state) systems with scenario-based specifications, as is done here, and for dividing inductive property verification efforts into several subproblems. Manual proofs following this approach appeared before in [9, 11]. In [4], a proof environment based on a general-purpose theorem prover was presented. As usual in a deductive theorem-proving tool, the proof environment is wide in its range of applicability but shallow in the degree of automation it provides. However, it formalizes the proof method's definitions, and encapsulates key lemmas needed in a theorem proving setting.

The work presented here also differs from partial order reductions in model checking [2, 7, 17, 18], that exploit similar notions of independence to reduce the size of the model being checked. In those works there is no separate description of the desired convenient computations. Moreover, only restricted forms of independence can be used, and many computations that are not convenient are left in the reduced version, because the reduction is conservative, and is done on-the-fly based on little context information. Thus those works use some related ideas to solve a completely different problem.

Transduction methods have been used in a theoretical deductive setting for the verification of partial-order refinements, and in particular sequential consistency of lazy caching, in [8]. There, the transducer holds a pomset of unbounded size, and the independence relation among operations is fixed. Finite-state serializers [3, 6] have been used to verify Sequential Consistency for certain classes of shared memory protocols by model checking. These can be seen as special cases of the transduction idea, optimized and tailored to support verification of that specific property.

This paper is organized as follows: In Section 2 we provide basic definitions about the reductions and equivalence relations we want to prove. In Section 3 we show how to augment a given transition system to enable model checking equivalence of every computation to a convenient one. In Section 4 we describe a prototype tool that implements the techniques described in the previous sections, and apply it to a simple yet paradigmatic example. Section 5 concludes with a discussion.

2 Theoretical Background

2.1 Computation Model, Conditional Independence

The following definition of Fair Transition Systems (FTS) is based on [13].
Definition: $M = \langle V, \Theta, \mathcal{T}, \mathcal{J} \rangle$ is an FTS iff: (i) V is a finite set of *system variables*. Let Σ denote the set of assignments to variables in V, also called $states$.
(ii) Θ is an assertion on variables from V (characterizing the *initial* states).
(iii) $\mathcal{T}$ is a finite set of *transitions*. Each $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \to 2^{\Sigma}$. We say that τ is *enabled* on the state s if $\tau(s) \neq \emptyset$, which we denote $en(s, \tau)$. We require that $\mathcal{T}$ includes an *idling* transition ι, such that for every state s, $\iota(s) = \{s\}$.
(iv) $\mathcal{J} \subseteq \mathcal{T} \setminus \{\iota\}$ is a set of transitions called *just* or *weakly fair*.

A *run* (or *execution sequence*) σ of M is an infinite sequence of state-transition pairs: $\sigma \in (\Sigma \times \mathcal{T})^{\omega}$, i.e., $\sigma : (s_0, \tau_0), (s_1, \tau_1), \ldots$, such that s_0 satisfies Θ ("initiality"), and $\forall i \in \mathbb{N} : s_{i+1} \in \tau_i(s_i)$ ("consecution"). A transition τ is enabled *at position* i in σ if it is enabled on s_i, and it is *taken* at that position if $\tau = \tau_i$. A *computation* is a run such that for each $\tau \in \mathcal{J}$ it is not the case that τ is continually enabled beyond some position j in σ but not taken beyond j ("justice"). We say that a state s is *quiescent* iff every transition $\tau \in \mathcal{J}$ is disabled in s. A computation σ is quiescent at position i iff s_i is quiescent, and $\tau_i = \iota$.

Given a set $S \subseteq \Sigma$ and a transition τ, let $\tau(S) = \bigcup_{s \in S} \tau(s)$.
Definition: Two transitions τ_1, τ_2 are *conditionally independent* in state s – denoted by $CondIndep(s, \tau_1, \tau_2)$ – iff $\tau_1 \neq \tau_2 \wedge \tau_2(\tau_1(s)) = \tau_1(\tau_2(s))$

In the rest of this paper we assume that all transitions of the system being analyzed are deterministic (i.e., their range has only empty or singleton sets), and if $\tau(s) = \{s'\}$ we refer to s' as $\tau(s)$. Under this assumption, our definition of conditional independence coincides with the functional independence defined in [4], and with the restriction imposed on acceptable *collapses* in [10], for pairs of transitions that actually occur in consecution (which are candidates to be swapped).

2.2 Swap Equivalence

In the sequel we consider a user-defined conditional independence relation $I \subseteq (\Sigma \times \mathcal{T} \times \mathcal{T})$ that is a subset of the full conditional independence $CondIndep$ [2], and such that $\forall s \in \Sigma \ \ \forall \tau \in \mathcal{T} : \ I(s, \tau, \iota) \wedge I(s, \iota, \tau)$. We define swap equivalence, following [4]. Note that for runs that share a suffix, the justice requirements defined by $\mathcal{J}$ hold equally. Let $\sigma = (s_0, \tau_0), (s_1, \tau_1), \ldots$ be a computation of M,such that for some $i \in \mathbb{N}$, $I(s_i, \tau_i, \tau_{i+1})$ holds, and thus $CondIndep(s_i, \tau_i, \tau_{i+1})$. Then, the sequence $\sigma' = (s_0, \tau_0), \ldots, (s_i, \tau_{i+1}), (\tau_{i+1}(s_i), \tau_i), (s_{i+2}, \tau_{i+2}), \ldots$ is also a legal computation of M, and we say that σ and σ' are *one-swap-equivalent* ($\sigma \equiv_{1sw} \sigma'$). The *swap-equivalence* ($\equiv_{sw}$) relation is the reflexive-transitive closure of one-swap-equivalence. The definitions of $\equiv_{1sw}$ and $\equiv_{sw}$ can also be applied to finite prefixes.

Swap-equivalence is the "conditional trace equivalence" (for finite traces) of [10]. If the independence relation I is fixed (i.e., $\forall s, t \in \Sigma, \forall \tau_1, \tau_2 \in \mathcal{T} : \ I(s, \tau_1, \tau_2) = I(t, \tau_1, \tau_2)$), swap-equivalence coincides with trace equivalence [14].

[2] we refer to sets and their characteristic predicates interchangeably

2.3 Conditional Trace Equivalence

Let $\sigma^{\restriction l} \in (\Sigma \times \mathcal{T})^l$ denote σ's prefix of length l. For infinite runs g, c:
Definition: $c \sqsubseteq g$ iff $\forall m \in \mathbb{N}\ \exists h \in (\Sigma \times \mathcal{T})^\omega : (c^{\restriction m} = h^{\restriction m}) \wedge h \equiv_{sw} g$

The relation $\sqsubseteq$ also appeared in [11, 12], but exploiting only fixed independence among transitions. If $c \sqsubseteq g$, there are finite execution paths from every state in c to one in g (See Fig.3) but c and g may not converge into a shared state.

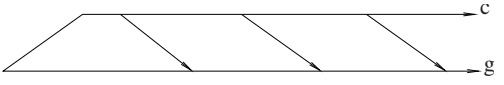

Fig. 3. $c \sqsubseteq g$. (Converging paths represent swap-equivalent runs)

Definition: Two infinite runs g and c are *conditional trace equivalent* ($g \approx c$) iff $c \sqsubseteq g$ and $g \sqsubseteq c$.

3 Proving Equivalence by Model Checking

We define the transducer described in Section 1 as $M' = (V', \Theta', \mathcal{T}', \mathcal{J}')$, where
(i) V' includes the variables in V, the *error* flag, a queue that can hold at most L state-transition pairs (the window), the state variables V_c of the chopper, and an index into H. Let Σ_c denote the possible states of C. The set of anti-transitions is:
$\overline{\mathcal{T}} = \{\overline{\tau} : \Sigma \to 2^\Sigma \mid \exists \tau \in \mathcal{T} \cdot \forall s_1, s_2 \in \Sigma,\ s_1 \in \overline{\tau}(s_2)$ iff $s_2 = \tau(s_1)\}$. The set of transitions that can be recorded in the window is $\mathcal{T}_h = \mathcal{T} \cup \overline{\mathcal{T}} \setminus \{\iota, \overline{\iota}\}$. Let $A^{0..L} = \cup_{i=0}^{L} A^i$. The set of states of M' (valuations of V') is $\Sigma' = \Sigma \times \{0,1\} \times (\Sigma \times \mathcal{T}_h)^{0..L} \times \Sigma_c \times 0..L$. M' is in state $s' \in \Sigma' = (s, b, h, s_c, np)$ iff its M component is in the state s, the error flag has the boolean value b, the window's contents is h, the chopper is in the state s_c and the earliest point in the window where a prediction can be introduced is np. Let the predicate $ep(h, np)$ hold iff np points to the place in the window right after the last predicted event.
(ii) $\Theta' = \{(s_0, false, \epsilon, s_{c_0}, 0) \mid s_0 \models \Theta \wedge s_{c_0} \models \Theta_c\}$, where Θ_c characterizes the initial states of the chopper.
(iii)$\mathcal{J}' = \{\tau' | \tau \in \mathcal{J}\}$
(iv)$\mathcal{T}'$ has the following transitions:

- τ': Simulating transitions of M. For every $\tau \in \mathcal{T}$:
  $\tau'((s, b, h, s_c, np)) =$(**if** $\tau = \iota$ **then** $\{(s, b, h, s_c, np)\}$ **else if** $(b = true \vee |h| = L)$ **then** $\{(\tau(s), true, h, s_c, np)\}$ **else** $\{(\tau(s), false, h \cdot (s, \tau)\ , s_c, np)\}$).
- $swap$: Swapping consecutive events in the window.
  $swap((s, b, h_1, s_c, np)) = \{(s, b, h_2, s_c, np') \mid h_2 \equiv_{1sw} h_1 \wedge ep(h_2, np')\}$.
- $chop$: Removing convenient prefixes from the window. $chop((s, b, h_1, s_{c_1}), np) = \{(s, b, h_2, s_{c_2}, np') \mid \exists p \in (\Sigma \times \mathcal{T}_h)^{0..L}.\ h_1 = p \cdot h_2 \wedge s_{c_1} \xrightarrow{p} s_{c_2} \wedge ep(h_2, np')\}$.[3]

[3] $s_{c_1} \xrightarrow{p} s_{c_2}$ means that C can move from s_{c_1} to s_{c_2} by reading p.

- *predict*: Inserting an event/anti-event pair.
  $predict((s,b,h,s_c,np)) = \{(s,b,p\cdot(s_1,\tau)\cdot(\tau(s_1),\overline{\tau})\cdot q, s_c, np') \mid h = p\cdot q \wedge$
  $(q=\epsilon \rightarrow s_1 = s) \ \wedge \ (q\neq\epsilon \rightarrow (s_1$ *is the first state of* $q)) \ \wedge$
  $p\in(\Sigma\times\mathcal{T})^* \ \wedge \ |p|\geq np \ \wedge \ np' = |p|+1\}$
- *swap-pred*: Swapping an anti-transition with an unrelated following transition.
  $((s,b,h,s_c,np)) = \{(s,b,p\cdot(\tau(s_1),\alpha)\cdot(\alpha(\tau(s_1)),\overline{\tau})\cdot q, s_c, np) \mid h = p\cdot(\tau(s_1),\overline{\tau})\cdot$
  $(s_1,\alpha)\cdot q \wedge I(s_1,\alpha,\tau)\}$
- *cancel*: removing an anti-event/event pair from the window.
  $cancel((s,b,h,s_c,np)) = \{(s,b,p\cdot q, s_c, np) \mid h = p\cdot(s_1,\overline{\tau})\cdot(s_2,\tau)\cdot q\}$

We now describe a set of properties to be checked on the transducer M'. Our implementation of this technique generates automatically CTL and LTL formulas that express these properties. For clarity reasons, the properties are described verbally here, omitting most of the actual formulas generated by the tool.

1. From every state in which the error flag is false, a state can be reached in which the window is not full: $AG(\neg error \rightarrow EF(\neg error \wedge |h| < L))$
2. In every computation where the error flag remains false, every anti-transition in the history can be swapped with any "normal" transition following it in the window (so it can reach the end of the history queue).
3. (For every τ that can be predicted): If $\overline{\tau}$ is at the end of the window then τ is enabled in the current state of M.
4. (For every τ that can be predicted): If $\overline{\tau}$ is not followed by τ somewhere behind it in the window, then τ will eventually be taken.
5. (For every τ): If the computation does not remain forever quiescent, then if τ is in the window, it will eventually be chopped or will be cancelled with $\overline{\tau}$.
6. From every quiescent state with the error flag false, it is possible to remain quiescent (while reordering and chopping the window) until the window is emptied.
7. Along every path it is infinitely often possible to stop executing the system's transitions (and perform history-related operations like predictions, swaps, chops), until the history is emptied or contains only anti-transitions: $AG\ AF(\neg error \rightarrow E($*no-*$\tau$*-taken U no-*$\tau$*-in-*$h))$.
8. In every state $s\in\Sigma$, and for every $\tau_1,\tau_2\in\mathcal{T}$ such that $I(s,\tau_1,\tau_2)$, we have (i) $CondIndep(s,\tau_1,\tau_2)$, and (ii) if a third transition $\tau_j\in\mathcal{J}$ is enabled in s and remains enabled after executing τ_1 and then τ_2, then it is also enabled after executing τ_2 from s: $en(s,\tau_j)\wedge en(\tau_1(s),\tau_j)\wedge en(\tau_2(\tau_1(s)),\tau_j)\rightarrow en(\tau_2(s),\tau_j)$.

If branching-time property 1 holds, then the transducer can "simulate" every computation g of M, while the concatenation of the events chopped from the window forms a convenient run c such that $c\sqsubseteq g$. Linear-time properties 2,3 and 4 are needed to justify the predictions. For the chopped sequence of events to be equivalent to the original computation of M, they must have the same events. Properties 5 and 6 are needed to rule out cases when an event remains in the window forever. Branching-time property 6 deals with computations that become and remain quiescent. Branching-time property 7 is used to extend the reduction ($c\sqsubseteq g$) into full equivalence ($c\approx g$). Property 8 is an invariant that can also be checked on the original system M (without a transducer). It checks the requirement that the user-defined independence relation implies conditional independence, and a restriction needed for the equivalence to preserve justice.

Theorem 1 *When verified on the transducer M', properties 1-8 imply that every computation of M is equivalent to some convenient one.*

The full proof can be found in [1]. Here we present only a detailed outline of the proof. First, we define the main concept used to link the definition of the transducer with the equivalence we need to prove.

Definition: Given an infinite computation g, and an infinite sequence $R = \{r_i\}_{i=0}^{\infty}$ of infinite computations, we say that R *is a reduction sequence from* g iff $g = r_0$ and $\forall i \geq 0 : r_i \equiv_{sw} r_{i+1}$.

We say that a computation g' of M' *follows* a computation g of M if g is the projection of g' to the transitions and variables of M, and the $error$ flag remains false along g'. Let us consider a computation g' that follows g. At any given time i along the execution of g', a computation r_i can be built as the concatenation of the portion already chopped, with the contents of the window, followed by the part of g that did not execute yet (e.g., as depicted in Figure 2). If we do not consider predictions, it is easy to see that the sequence $\{r_i\}_{i=0}^{\infty}$ is a reduction sequence: $r_0 = g$, and the only change from some r_i to r_{i+1} can be a swap of consecutive independent events. In general, if a reduction sequence from g converges, and its limit is c, then it can be proved that $c \sqsubseteq g$.

Property 1 implies that for every computation g there is a g' that follows it, so for every g there is a convenient run c such that $c \sqsubseteq g$, which is one half of what we need to prove. The predictions, as we saw, are just another way to prove swap equivalence, by moving anti-events forward in time instead of real events backwards. The net effect of a predict-swap-cancel sequence is a simple swap equivalence, so a reduction sequence from g can still be built from a computation g' that follows g. Of course, this requires properties 2, 3, and 4 to make sure the predictions are correct.

To prove full equivalence ($c \approx g$), we need to prove $g \sqsubseteq c$, which means that infinitely often along g, there is a continuation that is swap-equivalent to c. We now explain why this is precisely what Property 7 implies. Consider point g_0 in Figure 4. By Property 7,

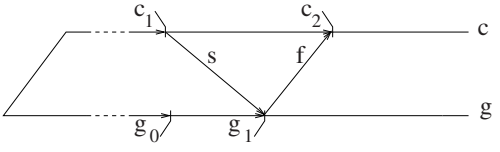

Fig. 4. Proving full equivalence.

there is a later point g_1 from which the transducer can stop executing transitions until (i) the window is empty, or (ii) it contains only anti-transitions. If the window is empty, then g and c have actually converged. If it contains only anti-transitions, this means that all the pending scenarios have been completed by predictions, and chopped out of the window. In the figure, the transitions in the segment marked with s are the ones starting new scenarios, those in the segment marked with f are the ones finishing pending scenarios. The contents of the window are the anti-transitions of those in the f segment. The actual occurrences of the events from f along g will appear after g_1, and cancel with their

respective predictions. Thus, a segment like f (connecting g to c) *exists*, for every point g_0 arbitrarily chosen along g, and this is what we need to prove the full equivalence.

The limit of a reduction sequence may not be a fair run. If we proved that the limit is an equivalent run, then the set of transitions that are taken infinitely often is the same in c as in g. The limit will be non-fair if some just transition which is never taken becomes enabled forever. This is avoided by the additional constraints on the user-defined independence relation introduced by Property 8. It is then easy to show by induction on the chain of swaps, that if τ_j is enabled from some point onwards, then the same happens in any swap equivalent run. This can be used in turn to show that if τ_j is enabled in c from some point onwards, then it must also be so in g. The proof is based on the fact that infinitely many paths connect c and g (in both directions), and converging paths are swap-equivalent.

4 Prototype Implementation - CNV

CNV (for CoNVenient), a prototype implementation of the described techniques, is based on a simple language for the description of fair transition systems, their convenient computations, the independence relation I, and additional configuration information such as the window size L, and predictions. The program `cnv2smv` translates a `.cnv` file into a description of the transducer and its proof obligations in the language of Cadence SMV, in which the verification and analysis of counterexamples is done. CNV includes optimizations meant to make the window smaller and use it efficiently:
(i) If the definition of the scenarios and of I depends only on an abstraction of the state information (e.g., on part of the state variables), and it is possible to update that abstract state with the effect of every transition, then it suffices to keep only the abstract state information in the window. An example is a queue, where the independence of *sends* and *receives* is affected only by the length of the queue, and not by its contents. A common example is a system where $I(s, \tau_1, \tau_2)$ does not depend on s. It is then possible to have only transitions in the window, without any state information. CNV's input language supports the manual definition of an abstract state to be stored in the window.
(ii) When a predicted transition completes a scenario so it can be chopped, instead of inserting the prediction and then chopping the whole scenario, the partial scenario is automatically replaced by the corresponding anti-transition. Similarly, if a transition is taken when its anti-transition appears at the end of the window, then instead of enqueueing the transition and then cancelling it with the anti-transition, CNV immediately removes the anti-transition from the window.
(iii) To keep the window as small as possible, the transitions of the transducer generated by CNV are prioritized. Enabled transitions are executed in the following descending order of priority: chops, cancelling of predictions, swapping anti-transitions towards the end of the history, making new predictions, and finally (with the same priority) making swaps and executing transitions from M.

Apart from proving that every computation is equivalent to a convenient one (by answering "true"), CNV can in some cases provide a detailed reduction sequence from a specific computation g. If it is possible to describe a *single* computation g by an LTL formula, then CNV can be asked to check that there is *no* reduction from g to a convenient

computation. If this is not the case, the counterexample generated shows the reduction steps in detail.

The language of CNV allows the definition of scalar variables and a finite set of transitions, where each transition may be marked as just. For every transition, an enabling predicate and an assignment to every affected variable are defined. It is possible to define abstract state variables to be kept in the history, and to specify how they depend on the system state variables. The choppers currently supported by CNV are defined by giving a prioritized list of "convenient prefixes"(finite sequences of transition names) representing scenarios. The predictions are also given in a prioritized list. Each entry actually describes a sequence of predictions. For example: PRED $Y\ Z$ AFTER $W\ X$ will trigger the prediction of Y when the history has a prefix WX, and will then predict Z after the prefix $W\ X\ Y$, chopping the whole scenario immediately. The result of these two predictions on $W\ X\ Q$ is $\overline{Z}\ \overline{Y}\ Q$.

Application Example: We use a schematic example to describe various parts of a methodology for which our technique can be useful. We consider a setting as the one shown in Fig.5, where two *producers* create complex values (maybe from an unbounded domain) and send them through a FIFO buffer, to a single *consumer*. The insertion of the value into the queue is not atomic, therefore the producers run Peterson's mutual exclusion protocol [16] to avoid enqueuing at the same time. The convenient sequences are naturally defined as those in which every produced value is immediately enqueued, dequeued and consumed. The relation I we will define does not depend on the actual values being produced and transmitted. Therefore, equivalence based on it can be model checked by CNV, by ignoring those values altogether.

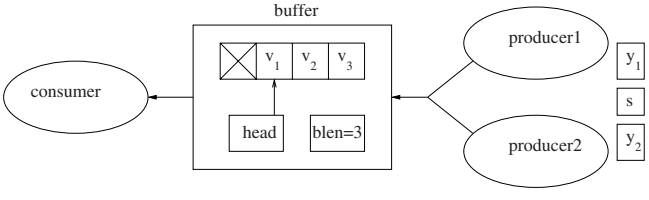

Fig. 5. Two mutually exclusive producers and one consumer with a FIFO buffer in between.

A valuable feature of CNV is its capability to verify Property 8 by generating and model checking an invariant of the original system (without any transducer). To prove commutativity, we cannot completely ignore the fact that different processes may overwrite each other's values in the buffer. However, when proving the equivalence itself with the transducer, the actual values produced and sent through the buffer are not relevant. Therefore, we can remove the produced data bits from the transducer we use to check the equivalence itself (Properties 1-7).

The convenient computations of the system in Figure 6, as described before, should be those in which transition l_2 is immediately followed by l_3, l_4, c_1, c_2 (and similarly for m_2). Proving the reduction in a single step means that the history queue we may need for the proof could be quite long, since there must be enough room in the history

```
Initially: head=blen=y1=y2=0, s=1
Producer 1:                  ||Producer 2:                  ||Consumer:
loop forever do              ||loop forever do              ||loop forever do
 l1:<PRODUCE v1;y1:=1;s:=1> || m1:<PRODUCE v2;y2:=1;s:=2> || c1:await blen>0
 l2: await (y2=0\/s!=1) /\  || m2: await (y1=0\/s!=2) /\  || c2:<CONSUME
b[head];
           (blen<N)          ||           (blen<N)          ||  head:=(head+1) mod
N;
 l3: STORE(v1,b[head+blen]) || m3: STORE(v2,b[head+blen]) ||  blen:=blen-1>
 l4: <blen:=blen+1;y1:=0>   || m4: <blen:=blen+1;y2:=0>   ||end loop
end loop                    ||end loop                    ||
```

Fig. 6. Pseudo code of the system

for the convenient prefixes and for any anti-transitions that may be generated before a convenient prefix is created. The computational complexity of the verification depends heavily on the length of the history, so we prefer to perform the reduction in stages.

In a first stage, we prove that every computation is equivalent to one in which the sequences (l_2, l_3, l_4), (m_2, m_3, m_4), and (c_1, c_2) are executed atomically. This basically proves that there is mutual exclusion between the two producers in their access to the buffer. Our independence relation I depends on the state (e.g., l_2 and c_2 are independent if the buffer is not full). According to I, we chose to store in every history element the variables y_1, y_2, s and $blen$ (the buffer's current length). The chopper recognizes the following prefixes as convenient: (l_1),(l_2, l_3, l_4),(m_1),(m_2, m_3, m_4),(c_1, c_2). The predictions we used are: "PRED l_3 l_4 AFTER l_2", "PRED m_3 m_4 AFTER m_2", and "PRED c_2 AFTER c_1". A window of length 4 was sufficient to prove the reduction, for a buffer length N=2. Verifying the equivalence (Properties 1-7) on the data-less version of the transducer took 105 minutes and 303MB of memory[4] on a 1.2GHz machine, and verifying the independence relation (Property 8) on the original system (i.e., with the buffer's data bits but without the transducer) took less than 10 seconds.

In the second stage, we model these sequences as atomic transitions (and we adjust I accordingly), and prove that every computation is equivalent to one in which every occurrence of l_2 (which now includes l_3 and l_4) is immediately followed by c_1 (which now includes c_2), and similarly for m_2. This time, a window of length 3 was sufficient to prove the reduction, for the same buffer length. Verifying the equivalence on the data-less version took 2 minutes and 14MB of memory, and verifying the independence relation on the full version of the original system without the transducer took less than 5 seconds.

To further test this approach, we split the first stage into two sub-stages: In the first one we prove equivalence to computations in which the transition pairs (l_3, l_4), (m_3, m_4) and (c_1, c_2) occur atomically. The second sub-stage proves equivalence to computations in which (l_2, l_3, l_4) and (m_2, m_3, m_4) occur atomically, as in the first big stage above. The first sub-stage took 36 minutes and used 89MB and the second one took 6 minutes and used 37MB, so together they took 42 minutes instead of the 105 minutes required previously, and used less than one third of the memory space.

This example's sources, the `cnv2smv` tool and a skeleton input file appear in [1].

[4] This is attributed to the asynchronous nature of the system, which is not particularly suited to the SMV model checker. An additional "running" variable is created by SMV for every transition of the transducer, and the resulting BDDs are not as compact as for hardware verification.

5 Discussion and Future Work

This paper has presented, for the first time, a general framework and tool for model checking conformance of a model to scenario-based or transaction-based specifications. The automatically generated transducer, including the window into the history and the added transitions for chopping and swapping events in the window, of course adds to the size of the state-space and transition relation. This should be viewed as the price for making the strong claim of computation equivalence in such specifications, analogous to the cross-product of the model with an automaton seen in automata-based model checking of linear-time temporal properties.

Although in theory the number of state variables is increased by $L \times (|V| + \log |\mathcal{T}|)$, the effective increase can be greatly reduced by exploiting optimizations, just as in other model checking tasks. Even in the prototype implementation described here, the transducer's transitions are prioritized in order to minimize the length of the needed window, and facilities are provided for abstracting away unneeded state variables from the history, reducing the amount of information kept there to the minimum needed to support the history-related operations. Usually, this means that each window element has only a few state bits in addition to the $\log |\mathcal{T}|$ needed to represent a transition. The possibility of using predictions to chop a scenario before it has completed, and using an anti-event to ensure that predicted events eventually occur, is also crucial in reducing the size of the window.

Moreover, it should be noted that not all of the transducer states exploit the full size of the window, and many transitions simply cannot occur from some states. A careful tuning of the variable ordering could use this fact to facilitate the construction of more compact BDDs. An explicit-state model checker, in principle, could also exploit this fact by using a dynamically sized representation of the states reached during the explicit search.

In future work, the influence on complexity of modelling decisions, such as how to abstract the state, and which predictions to make, needs to be better understood. Further compressing the contents of the window is crucial. Splitting the description of a long scenario into shorter sub-scenarios, and proving the equivalence in several steps, as seen in the example, also offers significant benefits. Other research directions include further development of notations for expressing scenarios, including connections to existing modelling techniques like UML.

References

1. `http://www.cs.technion.ac.il/Labs/ssdl/pub/CNV`.
2. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 340–351. Springer-Verlag, 1997.
3. T. Braun, A. Condon, A. J . Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving sequential consistency by model checking. In *Proc. 6th IEEE High Level Design Validation and Test Workshop, (HLDVT'01)*, pages 103–108, December 2001.
4. M. Glusman and S. Katz. A mechanized proof environment for the convenient computations proof method. *Formal Methods in System Design*. To appear. Available at `http://www.cs.technion.ac.il/Labs/ssdl/pub/conv_PVS`.

5. M. Glusman and S. Katz. Mechanizing proofs of computation equivalence. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification, (CAV'99)*, volume 1633 of *LNCS*, pages 354–367. Springer-Verlag, 1999.
6. T. A. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification, (CAV'99)*, volume 1633 of *LNCS*, pages 301–315. Springer-Verlag, 1999.
7. G. J. Holzmann and D. Peled. The state of SPIN. In *Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 385–389. Springer-Verlag, 1996.
8. B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. *Distributed Computing*, 12:129–149, 1999.
9. S. Katz. Refinement with global equivalence proofs in temporal logic. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–78. American Mathematical Society, 1997.
10. S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.
11. S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.
12. M. Kwiatkowska. *Fairness for Non-Interleaving Concurrency*. PhD thesis, Dept. of Computing Studies, Leicester, 1989.
13. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Safety*. Springer-Verlag, 1995.
14. A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and editors G. Rozenburg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 279–324. Springer-Verlag, 1986.
15. Ken L. McMillan. *Getting Started With SMV*. Cadence Berkley Labs, 2001 Addison St. Berkley, CA, March 1999.
16. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
17. A. Valmari. A stubborn attack on state explosion. In *Proc. 2nd. Workshop on Computer-Aided Verification*, volume 531 of *LNCS*, pages 156–165. Springer-Verlag, 1990.
18. P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, 1993.