Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 187 (2007) 75–90

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Simulation Machines for Checking Action System Refinements

Graeme Smith[1] and Kirsten Winter[2]

School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Abstract

Action systems provide a formal approach to modelling parallel and reactive systems. They have a well established theory of refinement supported by simulation-based proof rules. This paper introduces an automatic approach for verifying action system refinements utilising standard CTL model checking. To do this, we encode each of the simulation conditions as a *simulation machine*, a Kripke structure on which the proof obligation can be discharged by checking that an associated CTL property holds. This procedure transforms each simulation condition into a model checking problem. Each simulation condition can then be model checked in isolation, or, if desired, together with the other simulation conditions by combining the simulation machines and the CTL properties.

Keywords: Refinement, model checking, action systems, CTL.

1 Introduction

Action systems [2] are a mature formalism for the specification and step-wise development of parallel and reactive systems. They are capable of modelling systems with terminating, non-terminating and aborting behaviours. Action system refinement [3,1] is defined in terms of the sequential refinement calculus [5]. It is a very general notion of refinement allowing significant changes to the design of the system in each refinement step, i.e., the number of actions and the roles of particular actions can entirely change. This is facilitated by the partitioning of actions in the abstract and concrete specifications into those which are externally observable and those which are internal *stuttering* actions. Simulation-based proof rules for action system refinement have been presented by Back and von Wright [4].

[1] Email: smith@itee.uq.edu.au

[2] Email: kirsten@itee.uq.edu.au

doi:10.1016/j.entcs.2006.08.045

The need for tools to support refinement is well recognised. Without such support, refinement is impractical for all but the most critical systems. Traditionally, such tool support has been based on interactive theorem provers. This includes explicit support for action systems [14], as well as the refinement calculus [7]. More recently, advances in automatic verification technologies, including decision procedures and model checking, has seen steps towards fully automatic approaches to verifying refinements [11,6,12]. In particular, Smith and Derrick [12] show how the simulation proof obligations for Z refinement can be encoded in a standard model checker.

In this paper, we adopt a similar approach for the more general action system refinement: Rather than encoding the abstract and concrete systems directly into the model checker, we encode each simulation condition as a Kripke structure, or total state transition system, referred to as a *simulation machine* together with a property formalised in the branching-time temporal logic CTL [10]. This is done in such a way that the proof obligation holds exactly when the CTL property holds for the simulation machine. The latter can be automatically verified using a CTL model checker. This approach avoids the need to build the entire state space of both systems in most cases, and simplifies the properties that need to be checked.

The paper is organised as follows: In Section 2 we provide as preliminaries an overview of action systems and action system refinement as well as a brief introduction to the temporal logic CTL. In Section 3 we discuss ways of representing the action system simulation conditions as a model checking problem and present our approach. In Section 4 we illustrate our approach through a case study and its encoding in the input notation of the SAL model checking tools [9]. We conclude with a discussion of limitations and future work in Section 5.

2 Preliminaries

2.1 Action systems

Action systems [2] are a formalism for modelling parallel and reactive systems. An action system model describes a machine consisting of an initialisation and a set of actions, each of which is a guarded command (comprising a guard, which enables the action when satisfied, and a statement). The actions are repeatedly executed until none of the actions are enabled and the machine terminates.

An action system $\mathcal{A}$ is of the form:

$$\mathcal{A} = |[\textbf{var}\ x : X \bullet x := x_0;\quad \textbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_n\ \textbf{od}\ ]|: z : Z$$

A state of an action system has two components, the *local* state and the the *global* state. In $\mathcal{A}$ above the local state is given by the variable x of type X which is initialised to x_0. The global state is given by the variable z of type Z. The actions A_1, …, A_n are executed in an interleaved fashion: one of the enabled actions is chosen nondeterministically at each step, until none of the actions are enabled. The system terminates if its final action terminates. The system aborts if in its final action a precondition of the statement fails and the action cannot terminate.

The machine $\mathcal{A}$ can be seen as a tuple (A_0, A), with initialisation command A_0 and action A, where A is the composition of single statements within the **do_od** loop above. The initialisation condition is denoted by pA_0. Predicate nA denotes the *next-state relation* of an always terminating statement A. The *enabledness guard* of action A is denoted as gA: as long as gA is satisfied action A can execute. The predicate $\neg gA$ thus models termination of action A. The *termination guard* of action A is denoted as tA. It models that action A is terminating properly. The negation, $\neg tA$, models that action A is aborting. DO_A denotes the termination guard of an iteration of A, i.e., $DO_A = \textbf{do}\ A\ \textbf{od}\ \mathit{true}$. A^n is used as a shorthand for iterating action A n times and A^* is the demonic choice over all n-fold iterations over A.

Some of the actions in A are singled out as *stuttering actions* $A_\natural$. A stuttering action always terminates and leaves the global state unchanged, i.e., $nA_\natural(a, u)(a', u') \Rightarrow (u' = u)$. Non-stuttering actions are called *change actions* $A_\sharp$.

2.1.1 Action system refinement

In order to prove trace refinement[3] between an abstract system $\mathcal{A} = (A_0, A)$ and a more concrete system $\mathcal{C} = (C_0, C)$, Back and von Wright introduce simulation-based proof rules [4]. Let R denote the abstraction relation between states of $\mathcal{C}$ and states of $\mathcal{A}$. An abstract system $\mathcal{A}$ can be simulated by a concrete system $\mathcal{C}$ (in a forward fashion) if actions in $\mathcal{A}$ and $\mathcal{C}$ can be decomposed into change actions and stuttering actions such that the following simulation conditions hold:

(i) Any initialisation followed by stuttering actions in $\mathcal{C}$ simulates initialisation followed by stuttering actions in $\mathcal{A}$.

$$\begin{aligned} (1)\quad & pC_0(c, u) \wedge nC_\natural^m(c, u)(c', u') \\ & \quad \Rightarrow \exists\, n, a, a'. R(a', c', u') \wedge pA_0(a, u) \wedge nA_\natural^n(a, u)(a', u') \end{aligned}$$

(ii) Any change action in $\mathcal{C}$ followed by stuttering actions simulates some change action in $\mathcal{A}$ followed by stuttering actions.

$$\begin{aligned} (2)\quad & R(a, c, u) \wedge (nC_\sharp;\ nC_\natural^m)(c, u)(c', u') \\ & \quad \Rightarrow \neg tA(a, u) \vee (\exists\, n, a'. R(a', c', u') \wedge (nA_\sharp;\ nA_\natural^n)(a, u)(a', u')) \end{aligned}$$

(iii) Any aborting state in $\mathcal{C}$ is related to aborting states in $\mathcal{A}$.

$$(3)\quad R(a, c, u) \wedge \neg tC(c, u) \ \Rightarrow\ \neg tA(a, u)$$

(iv) Any terminating state in $\mathcal{C}$ is related to terminating or aborting states in $\mathcal{A}$.

[3] The trace semantics of action systems captures the computational, and not just the input-output, system behaviour.

(4) $R(a, c, u) \wedge \neg gC(c, u) \Rightarrow \neg tA(a, u) \vee \neg gA(a, u)$

(v) Any state in $\mathcal{C}$ from which infinite stuttering is possible is related to states in $\mathcal{A}$ which are either aborting or from which infinite stuttering is possible.

(5) $R(a, c, u) \wedge \neg DO_{C_\natural}(c, u) \Rightarrow \neg tA(a, u) \vee \neg DO_{A_\natural}(a, u)$

If we can prove these simulation conditions for an abstract system $\mathcal{A}$ and a concrete system $\mathcal{C}$ as above then the trace refinement $\mathcal{A} \sqsubseteq \mathcal{C}$ is valid. This method is referred to as *forward simulation*.

Simulation conditions also exist for *backward simulation*. Our results so far, however, consider only forward simulation conditions. Model checking of backward simulation conditions will be an issue of our future work.

2.2 The temporal logic CTL

CTL [10] is a branching time temporal logic which is defined with respect to Kripke structures. A Kripke structure is a state transition system with a total transition relation. Let M be a Kripke structure and $\mathcal{V}$ a set of atomic propositions. A labelling function L maps each state in M to the set of atomic proposition that is satisfied in the state. A valid CTL formula is related to a state s in M, i.e., it is a state formula which is built from state and path formulas:

Definition 2.1 Syntax of CTL
state formulas:
i.) If $\varphi \in \mathcal{V}$, then φ is a state formula.
ii.) If φ and ψ are state formulas, then $\neg\varphi$ and $\varphi \vee \psi$ are state formulas.
iii.) If φ is a path formula, then $\mathbf{E}\varphi$ is a state formula.

path formulas:
i.) If φ and ψ are state formulas, then $\mathbf{X}\varphi$ and $\varphi\ \mathbf{U}\ \psi$ are path formulas.

E is an existential quantifier for paths, **X** refers to the next state, and **U** is an until operator for paths: $\varphi\ \mathbf{U}\ \psi$ states that φ is true until ψ becomes true (and ψ must eventually become true). Some additional operators are used as abbreviations: *boolean operators:* $\varphi \wedge \psi \Leftrightarrow \neg(\neg\varphi \vee \neg\psi)$ and $\varphi \Rightarrow \psi \Leftrightarrow \neg\varphi \vee \psi$, *eventually:* $\mathbf{F}\varphi \Leftrightarrow (\mathit{true}\ \mathbf{U}\ \varphi)$, *always:* $\mathbf{G}\varphi \Leftrightarrow \neg\mathbf{F}\neg\varphi$, and the universal quantifier over all paths (*for all paths*): $\mathbf{A}\varphi \Leftrightarrow \neg\mathbf{E}\neg\varphi$.

From the syntax definition above it is possible to derive three basic temporal logic operators to model state formulas, **EX**, **EU**, and **EG**. The semantics of CTL state formulas is inductively defined over the structure of state formulas as follows:

Definition 2.2 Semantics of CTL

$$s \models \varphi \quad \Leftrightarrow \varphi \in L(s), \text{if } \varphi \in \mathcal{V}$$

$$s \models \neg\varphi \quad \Leftrightarrow s \not\models \varphi$$

$$s \models \varphi_1 \vee \varphi_2 \quad \Leftrightarrow s \models \varphi_1 \text{ or } s \models \varphi_2$$

$$s \models \mathbf{EX}\,\varphi \quad \Leftrightarrow \begin{cases} \text{there is a path } \pi\text{, starting at state } s\text{, such that} \\ s_1 \text{ is the next state in } \pi \text{ and } s_1 \models \varphi \text{ holds} \end{cases}$$

$$s \models \mathbf{E}(\varphi_1\ \mathbf{U}\ \varphi_2) \quad \Leftrightarrow \begin{cases} \text{there is a path } \pi\text{, starting at state } s\text{, such that} \\ \text{there exists a } k \geq 0\text{, with } s_k \models \varphi_2, \text{and for all } 0 \leq j < \\ k,\ s_j \models \varphi_1 \text{holds} \end{cases}$$

$$s \models \mathbf{EG}\,\varphi \quad \Leftrightarrow \begin{cases} \text{there is a path } \pi\text{, starting at state } s\text{, such that for} \\ \text{all } k \geq 0,\ s_k \models \varphi \text{ holds.} \end{cases}$$

Five more operators are used frequently to specify CTL state formulas. They are defined based on the basic operators:

$$\mathbf{EF}\,\varphi \Leftrightarrow \mathbf{E}(\mathit{true}\ \mathbf{U}\ \varphi)$$

$$\mathbf{AX}\,\varphi \Leftrightarrow \neg\mathbf{EX}\,(\neg\varphi)$$

$$\mathbf{AG}\,\varphi \Leftrightarrow \neg\mathbf{EF}\,(\neg\varphi) \ \Leftrightarrow\ \neg(\mathbf{E}(\mathit{true}\ \mathbf{U}\ \neg\varphi))$$

$$\mathbf{AF}\,\varphi \Leftrightarrow \neg\mathbf{EG}\,(\neg\varphi)$$

$$\mathbf{A}(\varphi\ \mathbf{U}\ \psi) \Leftrightarrow \neg\mathbf{E}(\neg\psi\ \mathbf{U}\ \neg\varphi \wedge \neg\psi) \wedge \neg\mathbf{EG}\,(\neg\psi)$$

We use the temporal logic CTL to formalise properties that have to hold for each simulation machine in order to fulfil the corresponding simulation rule. Since CTL supports the existential quantification of paths of the system, it proves a suitable formalism.

3 Representing simulation rules

The idea for simulation machines arose out of earlier attempts at automating refinement via model checking in a more straightforward manner. In these earlier attempts, the state spaces and actions of the abstract and concrete systems are merged to produce a combined system on which to check refinement. Given action systems

$$\mathcal{A} = |[\mathbf{var}\ x : X \bullet x := x_0;\ \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_n\ \mathbf{od}\ ]|: z : Z$$

and

$$\mathcal{C} = |[\mathbf{var}\ y : Y \bullet y := y_0;\ \mathbf{do}\ C_1\ [\!]\ \ldots\ [\!]\ C_m\ \mathbf{od}\ ]|: z : Z$$

the combined system is

$$\begin{aligned}\mathcal{AC} = |[&\mathbf{var}\ x : X,\ y : Y \bullet x := x_0;\ y := y_0;\\ &\mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_n\ [\!]\ C_1\ [\!]\ \ldots\ [\!]\ C_m\ \mathbf{od}\ ]|: z : Z\end{aligned}$$

That is, the actions of $\mathcal{A}$ and $\mathcal{C}$ are interleaved in the combined system and only affect the part of the state corresponding to that of their original system. Where necessary local variables are systematically renamed to avoid name clashes. Also, *skip* transitions, i.e., transitions that do not change the state, are added to any states in which no actions are enabled, making the system a Kripke structure with a total transition relation (a necessary precondition for model checking).

Given this combined system, let us consider the initialisation simulation condition (1) on page 3. It requires that each concrete state that is reachable from an initial concrete state by a finite number of stuttering steps is related, via the abstraction relation R, to an abstract state that is reachable from an abstract initial state via stuttering steps.

To capture such properties in CTL, we first add an additional action to our combined system that enables us to reinitialise the abstract state. This action is always enabled. We then add an auxiliary variable *act* with values *cstutt*, *astutt*, *cchange*, *achange*, *ainit* and *none* to record that the last action that took place was a concrete stuttering action, an abstract stuttering action, a concrete change action, an abstract change action, the abstract state initialisation action, or no action (only true on initialisation), respectively. Simulation condition (1) can then be expressed as follows:

$$\begin{aligned}(6)\qquad &\mathbf{A}(\mathbf{EX}(act = ainit \land \mathbf{EX}(\mathbf{E}(act = astutt\ \mathbf{U}\ R))\\ &\qquad\qquad\qquad \mathbf{U}\ \neg\ (act = none \lor act = cstutt))\end{aligned}$$

This property states that until we have an action which is not a concrete stuttering action, it is possible to perform an *ainit* action, and then abstract stuttering actions until R holds.

As illustrated by (6), the CTL properties needed to capture simulation conditions can become quite complex and subtle. This leads to two problems. Firstly, it is not always easy to see the relationship between the CTL property and the original proof obligation, a fact which may complicate the interpretation of counter-examples provided by model checking. For instance, (6) might seem stronger than (1) at first glance since the "until" operator requires its right-hand side eventually be satisfied. Hence, the property requires that an action other than a concrete stuttering action eventually occurs. This is not a requirement in (1). However, the *ainit* action is always enabled so this requirement can alway be satisfied in our combined model $\mathcal{AC}$ (even if no actions apart from concrete stuttering actions are enabled in the original systems). Secondly, and more importantly, to model check such complex properties is computationally expensive, and in the worst case would render model checking infeasible.

It is possible to simplify the property, however, by carefully choosing the right auxiliary variables to add to our combined system. For example, we can add a boolean auxiliary variable cs that is true when we are in a state reached only by concrete stuttering actions after concrete state initialisation (i.e., $cs \Leftrightarrow pC_0(c,u);\ nC_\natural^m(c,u)(c',u')$). The value of cs would be true initially (since m can be 0). Similarly, we can add an auxiliary variable as that is true when we are in a state reached only by concrete state initialisation followed by concrete stuttering actions followed by abstract state (re)initialisation and then by abstract stuttering actions (i.e., $as \Leftrightarrow pC_0(c,u);\ nC_\natural^m(c,u)(c',u');\ pA_0(a,u);\ nA_\natural^n(a,u)(a',u')$). The value of as would be false initially (since it requires the abstract state be reinitialised). Both values would be updated by actions in $\mathcal{AC}$ corresponding to their intended meaning. In particular, both values would be set to false if any change action occurred. Additionally, the auxiliary variables become part of the guards of actions in $\mathcal{AC}$. That is, we restrict the behaviour of $\mathcal{AC}$ to those behaviours that allow us to prove simulation condition (1).

The values of the auxiliary variables, cs and as, at any time define a *meta-state*, i.e., one in which a number of states of the original combined system are possible. A transition system in terms of these meta-states is given in Figure 1a where *CStutt* and *AStutt* denote concrete and abstract stuttering actions, respectively, and *CChange* and *AChange* denote concrete and abstract change actions, respectively, and *AInit* denotes the abstract state initialisation action.

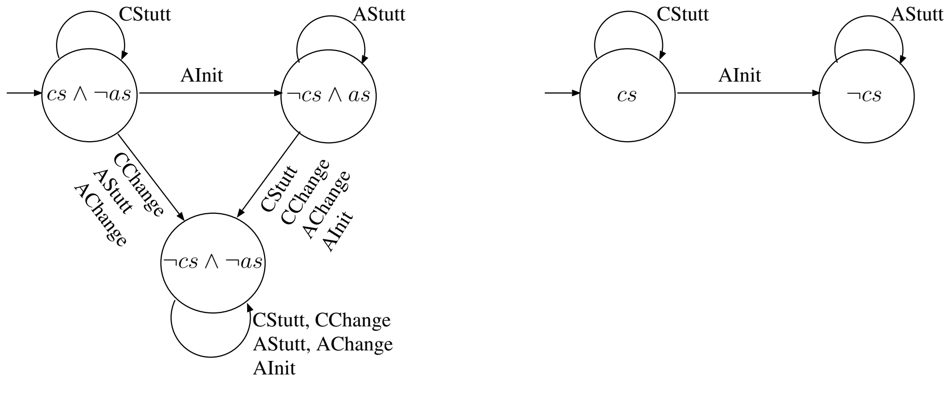


Fig. 1. a.) Meta-state transition system b.) Simplified meta-state transition system

The desired property with respect to the meta-state transition system is simply expressed by

$$\mathbf{AG}(cs \Rightarrow \mathbf{EX}(as \wedge \mathbf{EF}(as \wedge R))) \tag{7}$$

That is, for all paths, if cs is true then in the next step we can make as true such that it is possible that eventually as will still be true in a state where R is true. As well as being simpler than (6), this property has a closer correspondence to the form of the original simulation condition (1) on page 3.

The change actions have no affect on the verity of the CTL property (7), and so could be dropped from the transition system of Figure 1a. The resulting machine executes only initialisation and stuttering actions of the abstract and concrete systems which is sufficient for proving simulation condition (1). This allows us to also drop one of the auxiliary variables, as shown in Figure 1b, and to simplify the property to

$$\mathbf{AG}(cs \Rightarrow \mathbf{EX}(\neg cs \wedge \mathbf{EF}(R))) \tag{8}$$

The resulting transition system no longer reflects the behaviour of the combined systems $\mathcal{AC}$ (it captures only a controlled subset of this behaviour), yet still allows us to discharge the proof obligation of the simulation condition. We refer to such a simplified transition system for checking a simulation proof obligation as a *simulation machine*. In the rest of this section, we provide a more precise description of simulation machines and present the simulation machines for checking each condition of forward simulation to verify action system refinements.

3.1 Simulation machines

Simulation machines are formed by merging an abstract and concrete system to form a combined system whose state can can be partitioned into an abstract part (identical to that of the abstract system) and a concrete part (identical to that of the concrete system). Actions of the abstract and concrete systems are interleaved in the combined system and only act on that part of the state corresponding to that of their original system. Additionally, simulation machines have four key features:

(i) They divide the system into "phases" captured by the values of auxiliary variables. In each phase, only a subset of the enabled actions of the abstract and concrete systems are allowed to occur. Phase changes may be triggered by certain actions of the abstract or concrete systems, or may be able to occur at any time via additional actions in the simulation machine.

(ii) They do not include actions of the abstract and concrete systems that are not relevant to the simulation condition that they are used to check.

(iii) Their initial states do not necessarily correspond to initial states of the abstract and concrete systems. Additional actions which initialise the abstract or concrete part of the state may therefore also be included in the machine if required.

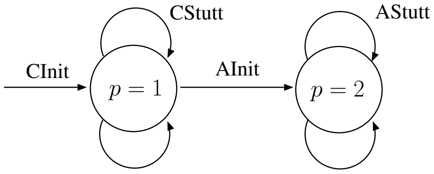
(iv) They include skip transitions in all phases that can only be exited by the occurrence of actions of the abstract and concrete systems. This ensures the machine is a Kripke structure: in any state it can either exit the state or skip. The CTL properties must cater for these skip actions.

The first two features have been illustrated in Figure 1b above. They are further illustrated along with the other features in the definitions of the simulation machines for each of the forward simulation conditions below. In order to combine the machines, and hence allow refinement to be checked in one step, we use a common auxiliary variable p, whose value ranges from 1 to 7, to represent the current phase in each of the machines.

3.1.1 Initialisation

As shown in Figure 1b, a simulation machine for checking the initialisation rule requires two phases (cs and $\neg\ cs$ in the figure), and only requires the stuttering abstract and concrete actions. Since we initialise the abstract part of the state when we move between the phases, initially the machine need only ensure the concrete part of the state is initialised. Skip transitions need to be added to each phase[4]. It can be readily verified that these transition have no affect on the property (7).

Hence, the simulation machine and CTL property required to discharge the initialisation simulation condition (1) is as shown below. We label the initialisation with *CInit* to indicate that the concrete part of the state is initialised on initialisation of the machine. The skip transitions are unlabelled.



$$\mathbf{AG}(p = 1 \Rightarrow \mathbf{EX}(p = 2 \wedge \mathbf{EF}(R))$$

3.1.2 Forward simulation

The main forward simulation condition (2) on page 3 requires that, given a concrete state c and a non-aborting abstract state a related via the abstraction relation R, each concrete state reachable from c by a concrete change action followed by a finite number of stuttering steps is related, via R, to an abstract state reachable from a by an abstract change action followed by a finite number of stuttering steps.

Since we are interested in actions that occur from states related by R in this case, we initialise our simulation machine to be in such a state. We then divide our system into 3 phases representing the cases where:

(i) no actions have occurred,

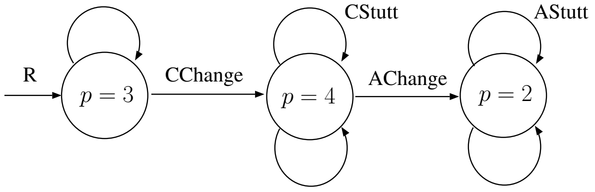
(ii) a concrete change action has occurred followed by a finite number of concrete stuttering actions,

(iii) a concrete change action has occurred followed by a finite number of concrete stuttering actions, followed by an abstract change action and a finite number of abstract stuttering actions.

Again skip actions need to be added to each phase.

[4] Although *AInit* is always enabled in the initial phase, the abstract initialisation condition may be unsatisfiable causing it to not be able to execute in a model checker implementation.

The simulation machine is below. Note that since the final phase allows the same behaviour as the phase $p = 2$ for the initialisation machine, we use this phase again in anticipation of combining the machines.



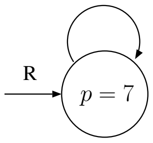
To simplify the CTL property associated with this simulation machine, we add a boolean variable *Aaborting* to our combined system state which is true precisely when the abstract part of the system is in an aborting state, i.e, when $\neg\ tA$ is true. The property is then:

$$\mathbf{AG}(p = 4 \Rightarrow \mathit{Aaborting} \vee \mathbf{EX}(p = 2 \wedge \mathbf{EF}(R)))$$

That is, for all paths (originating from a state where R holds), if we are in phase $p = 4$ then either the abstract part of the state is an aborting abstract state, or in the next step we can enter phase $p = 2$ such that eventually R is true.

3.1.3 Aborting states

The third simulation condition (3) on page 3 requires that aborting concrete states are related via R to aborting abstract states. Again we are interested in states related by R and we initialise our simulation machine to be in such a state. No actions are required to check this simulation condition and so the simulation machine is simply as shown below.



As above, we introduce a boolean variable *Aaborting* and a corresponding boolean variable *Caborting* which is true when the concrete part of the state is an aborting concrete state. The CTL property is then:

$$\mathit{Caborting} \Rightarrow \mathit{Aaborting}$$

That is, if (we are in a state where R holds and) the concrete part of the state is an aborting state then the abstract part of the state is an aborting state.

3.1.4 Terminating states

The fourth simulation condition (4) on page 4 requires that terminating concrete states are related via R to aborting or terminating abstract states. The simulation machine is identical to that above (and hence shares the phase $p = 7$).

We introduce a boolean variable *Aaborting* as above, as well as boolean variables *Aterminating* and *Cterminating* which are true precisely when the abstract and concrete parts of the state, respectively, are terminating states, i.e., when they satisfy $\neg\, gA$ and $\neg\, gC$ respectively. The required CTL property is then:

$$Cterminating \Rightarrow (Aaborting \vee Aterminating)$$

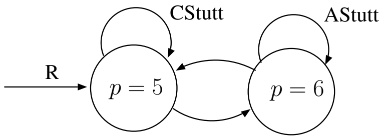
That is, if (we are in a state where R holds and) the concrete part of the state is a terminating state then the abstract part of the state is either an aborting or terminating state.

3.1.5 Infinite stuttering

The final simulation condition (5) on page 4 requires that concrete states from which infinite stuttering is possible are related to abstract states which are either aborting states or from which infinite stuttering is possible.

The simulation machine is initialised to states in which R holds. To check whether infinite concrete stuttering is possible, we only allow concrete stuttering steps in the initial phase and have a skip transition to a second phase to ensure totality. If it is possible to stay in the first phase, infinite concrete stuttering is possible.

The second phase is used to test for infinite abstract stuttering. It only allows abstract stuttering steps and can be exited by a skip transition back to the first phase. If it is possible to stay in the second phase, infinite abstract stuttering is possible. The simulation machine is shown below.



Note that in this case, skip transitions within the phases are not required due to the skip transitions which exit the phases. In fact, having skip transitions within the phases would cause a problem since they would make it always possible to stay in either phase (whether infinite stuttering was possible or not).

Given a boolean variable *Aaborting* as above, the CTL property associated with the machine is:

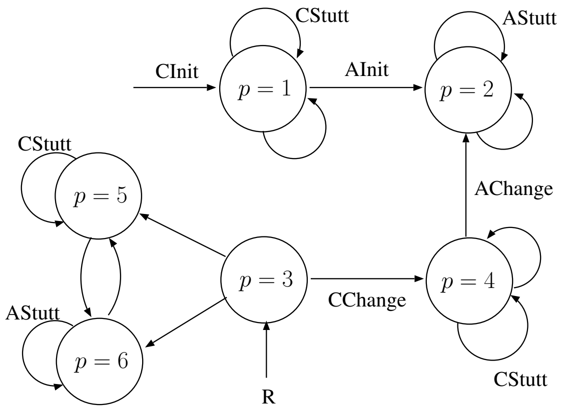
$$\mathbf{EG}(p = 5) \Rightarrow (Aaborting \vee \mathbf{EX}(\mathbf{EG}(p = 6)))$$

That is, if (R holds and) it is possible to always stay in phase $p = 5$, i.e., perform an infinite number of concrete stuttering actions, then either the abstract part of the state is an aborting state, or it is possible to enter and stay in phase $p = 6$, i.e., perform an infinite number of abstract stuttering actions.

3.2 Complete forward simulation check

The simulation machines presented so far enable us to check a forward simulation refinement by checking each of the simulation conditions individually. It is also possible to check a forward simulation refinement in one step by combining the simulation machines and associated CTL properties.

Firstly, as already noted, the phase $p = 2$ is shared by the first two simulation machines. Secondly, since the CTL properties regarding aborting and terminating concrete states refer to the initial state of a path only, their verity is not changed by enabling more actions in this initial state. Hence, phase $p = 7$ can be equated with phase $p = 3$. Finally, the skip transition within phase $p = 3$ can be made to exit this phase and enter either phase $p = 5$ or $p = 6$. The full simulation machine is shown below.



The CTL property needed to discharge all the refinement simulation conditions is the conjunction of each of the CTL properties for the individual simulation conditions with the following changes:

(i) The properties relating to aborting and terminating concrete states should only hold in phase $p = 3$, i.e., they no longer hold for all initial states since they do not necessarily hold in phase $p = 1$.

(ii) The antecedent of the property related to infinite stuttering needs to be prefixed with **EX** since phase $p = 5$ is no longer an initial state.

$$\begin{array}{l}
\mathbf{AG}(p = 1 \Rightarrow \mathbf{EX}(p = 2 \wedge \mathbf{EF}(R)) \wedge \\
(\mathbf{AG}(p = 4 \Rightarrow \mathit{Aaborting} \vee \mathbf{EX}(p = 2 \wedge \mathbf{EF}(R)))) \wedge \\
(p = 3 \wedge \mathit{Caborting} \Rightarrow \mathit{Aaborting}) \wedge \\
(p = 3 \wedge \mathit{Cterminating} \Rightarrow (\mathit{Aaborting} \vee \mathit{Aterminating})) \wedge \\
(\mathbf{EX}(\mathbf{EG}(p = 5)) \Rightarrow (\mathit{Aaborting} \vee \mathbf{EX}(\mathbf{EG}(p = 6))))
\end{array}$$

4 Case Study

Back [1] provides a small case study illustrating action system refinement. The abstract model $\mathcal{A}$ is given below. It specifies two processes updating a shared variable w (via actions $CS.0$ and $CS.1$).

```
A = |[var y.i : ℕ, cr.i : 𝔹 for i = 0,1 • cr.i := false   for i = 0,1;
       do
          cr.i → y.i := w + i + 1; w := y.i; cr.i := false   for i = 0,1        [CS.i]
        [] ¬cr.i → (cr.i := false [] cr.i := true   for i = 0,1)               [NS.i]
       od
     ]|: w : ℕ
```

The refinement results in a concrete model $\mathcal{C}$ that utilises Peterson's algorithm to ensure mutual exclusion when accessing the variable. Back splits up the updating actions $CS.i$ and introduces new variables: a local boolean $b.i$ that indicates the intention to enter the critical section, a shared variable t to indicate the willingness to retreat from accessing w (if $t = i$ then process i gives way for process $(1 - i)$), and a program counter for each process, $pc.i$, to control the sequence of actions. Actions $NS.i$ are unchanged, actions $CS.i$ are implemented by $CS'.i$ below and actions $BS.i$, $TS.i$ and $BR.i$ are additional stuttering actions that only change local variables and are thus not observable.

```
C = |[var b.i, cr.i : 𝔹, pc.i, y.i : ℕ   for i = 0,1, t : 0...1 •
           b.i := false; pc.i := 0; cr.i := false   for i = 0,1;
      do
         cr.i ∧ pc.i = 0 → b.i := true; pc.i := 1   for i = 0,1               [BS.i]
       [] pc.i = 1 → t := i; pc.i := 2   for i = 0,1                           [TS.i]
       [] pc.i = 2 ∧ (¬b.(1 − i) ∨ t = 1 − i) → y.i := w + i + 1;
              w := y.i; cr.i := false; pc.i := 3   for i = 0,1                [CS′.i]
       [] pc.i = 3 → pc.i := 0; b.i := false   for i = 0,1                     [BR.i]
       [] ¬cr.i → (cr.i := false [] cr.i := true)   for i = 0,1               [NS.i]
      od
    ]|: w : ℕ
```

4.1 Representing and checking action systems in SAL

We used our approach to verify the above refinement using the CTL model checker of the SAL tool suite [9]. Encoding action systems in the SAL language is straightforward since it also represents actions via guards and statements. Action $BS.0$ above, for instance, can be encoded as

```
BS0:  ccr0 AND cpc0=0 --> cb0'=TRUE; cpc0'=1;
```

To avoid name clashes between both models we extend all variables with a leading `a` (for variables in the abstract model) and `c` (for variables in the concrete model). Thus, variable $cr.0$ is encoded as `ccr0`.

The only complication that arises in the encoding is that types must be finite. Although SAL does support infinite types, in general the model checkers only work with finite types[5]. We restricted the type of $pc.i$ to be $0 \dots 3$ and that of $y.i$ and w to $0 \dots 10$.

To encode the complete simulation machine given in Section 3.2 we define variable $p : 1 \dots 6$. It allows us to restrict the behaviour of the machines to our intention. For example, $BS.0$ is a stuttering action in $\mathcal{C}$, i.e., belongs to $CStutt$. We have to restrict its occurrence accordingly to phases where $p \in \{1, 4, 5\}$. A stuttering action in $\mathcal{C}$ does not cause the simulation machine to change its phase, i.e., p remains unchanged:

```
    BS0:  (p=1 OR p=4 OR p=5) AND
                ccr0 AND cpc0=0 --> cb0'=TRUE; cpc0'=1;
```

Action $CChange$ in our simulation machine is given by actions $CS'.i$ and $NS.i$ in $\mathcal{C}$ above. For process 0 these are encoded as follows:

```
  CCS0:  p=3 AND cpc0=2 AND (NOT(cb1) OR ct=1)
        --> cy0'=cw+0+1; cw'=cy0; ccr0'=FALSE; cpc0'=3; p'=4;
```

and

```
  CNS0:  p=3 AND NOT(ccr0) --> ccr0' IN {b:BOOLEAN|true}; p'=4;
```

Note that the next state value of variable `ccr0` in action `CNS0` is chosen nondeterministically. Both actions change the phase of the simulation machine to $p = 4$.

The simulation machine also contains *skip* transitions that do not change any variable in the action system but may change the phase in the simulation machine. The skip from phase $p = 3$ to $p = 5$, for instance, is encoded as

```
            p=3 --> p'=5;
```

The encoded actions of the abstract and concrete systems are then combined using the SAL choice operator `[]`. Refinement relation R as well as all predicates used in the CTL properties (i.e., *Aaborting*, *Aterminating*, *CInit*, *Caborting*, and *Cterminating*) are encoded as definitions over the state variables of the system. At any phase of the simulation machine they can be evaluated to either true or false, e.g.,

```
    CInit = cb0=FALSE AND cb1=FALSE AND cpc0=0 AND cpc1=0 AND
              ccr0=FALSE AND ccr1=FALSE;
```

To encode the terminating conditions, we negate the disjunction of the relevant guards. To encode the aborting conditions, we disjoin the conjunctions of the relevant guards with the condition that the associated statement is not possible, e.g.,

[5] SAL has a bounded model checker which works with infinite types.

for the action $BS0$ we have

```
(ccr0 AND cpc0=0) AND
 FORALL (a:BOOLEAN, b:ProgramCounter):  NOT(a AND b=1)
```

where `ProgramCounter` is the type $0 \ldots 3$. The complete encoding of the case study can be found in [13].

The property associated with the simulation machine for the complete forward simulation refinement check for this case study could be checked to be valid. The checking process terminated in 3.55 seconds on a PC with a 3GHz Intel Pentium 4 processor and 512MB of RAM.

5 Conclusion

We have presented an approach to automatically verifying action system refinements. Although there has been previous work looking at automatic verification of Z-like refinements [11,6,12] this is the first approach, to our knowledge, for the more general action system refinements. Our approach can be used with any CTL model checker although the explicit support for action guards and high-level constructs such as quantifiers makes the SAL model checker [9] particularly suitable.

We are interested in extending the applicability of this work in two ways. Firstly, we have only considered forward simulation. This is by far the most common form of refinement, but for completeness we would like to extend our work to also cover backward simulation. Secondly, we are restricted to systems whose types are finite and not too large; otherwise model checking becomes infeasible. These limitations can be lifted, however, by utilising recent advances in the model checking field, e.g., automatic predicate abstraction [8] or bounded model checking [9]. We are particularly interested in looking at the former, and using the fact that the system structure (in terms of phases) and property we wish to prove are fixed, to simplify the abstraction process.

Acknowledgement

We acknowledge the support of Australian Research Council (ARC) Discovery Grants DP0345355 and DP0558408.

References

[1] R.J.R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, Computer Science Department, California Institute of Technology, 1992.

[2] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.

[3] R.J.R. Back and K. Sere. Superposition refinement of parallel algorithms. In K. Parker and G. Rose, editors, *Formal Description Techniques (FORTE IV)*, pages 475–493. North-Holland, 1992.

[4] R.J.R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Concurrency Theory (CONCUR '94)*, volume 836 of *LNCS*, pages 367–384. Springer-Verlag, 1994.

[5] R.J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[6] C. Bolton. Using the Alloy analyzer to verify data refinement in Z. In J. Derrick and E. Boiten, editors, *REFINE 2005*, volume 137, Issue 2 of *ENTCS*, pages 23–44. Elsevier, 2005.

[7] M. Butler, J. Grundy, T. Langbacka, R. Ruksenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 40–61. Springer-Verlag, 1997.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.

[9] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 496–500. Springer-Verlag, 2004.

[10] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier Science Publishers, 1990.

[11] M. Leuschel and M. Butler. Automatic refinement checking for B. In K. Lau and R. Banach, editors, *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 345–359. Springer-Verlag, 2005.

[12] G. Smith and J. Derrick. Model checking downward simulations. In J. Derrick and E. Boiten, editors, *REFINE 2005*, volume 137, Issue 2 of *ENTCS*, pages 205–224. Elsevier, 2005.

[13] G. Smith and K. Winter. Simulation machines for checking action system refinements. Technical Report SSE-2006-03, School of Information Technology and Electrical Engineering, University of Queensland, 2006.

[14] M. Waldén and K. Sere. Refining action systems within B-Tool. In *Formal Methods Europe (FME '96)*, volume 1051 of *LNCS*, pages 84–103. Springer-Verlag, 1996.