Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 205–224
www.elsevier.com/locate/entcs

Model Checking Downward Simulations

Graeme Smith[1]

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

John Derrick[2]

Department of Computer Science,
University of Sheffield,
Sheffield, UK

Abstract

This paper shows how downward simulation can be checked using existing temporal logic model checkers. In particular, we show how the branching time temporal logic CTL can be used to encode the standard downward simulation conditions. We do this for both a blocking, or guarded, interpretation of operations (often used when specifying reactive systems) as well as the more common non-blocking interpretation of operations used in many state-based specification languages (for modelling sequential systems). The approach is general enough to use with any state-based specification language, and any CTL model checker in which the language can be encoded.

Keywords: Refinement, model checking, CTL.

1 Introduction

Data refinement for state-based formalisms is usually checked by proving that the concrete system simulates the abstract one [7]. The notion of simulation is captured by downward and upward simulation rules comprising conditions relating the possible initialisations and transitions of the concrete and abstract systems. Proving these conditions by hand, even for simple systems, is at

[1] Email: smith@itee.uq.edu.au
[2] Email: J.Derrick@dcs.shef.ac.uk

doi:10.1016/j.entcs.2005.04.032

best tedious and at worst error-prone. Hence, tool support for proving data refinements is generally considered necessary.

Most existing tool support involves the use of theorem provers. These tools require the user to devise most of the proof steps and hence can help the user gain a deep understanding of the refinement. However, they also require a great deal of expertise in mathematical proof. Furthermore, when the user is not able to prove a condition, this may be either because the condition is not true (and the refinement does not hold), or because the proof required is too difficult for the user (and the refinement may hold). Distinguishing between these two cases is often difficult [4].

Model checking [3], as opposed to theorem proving, is a fully automatic technique for determining whether a specified system satisfies a given property. A model checker exhaustively checks the state space of a system to determine whether or not the property holds. In the latter case, a model checker will usually provide a counter-example, or witness, providing insight into why the property does not hold.

Model checkers were originally restricted to finite systems, and to simple notations suited to modelling systems where the complexity lay in the control structure, rather than the data, e.g., hardware systems and communication protocols. Recent advances mean that these restrictions are no longer absolute. Automatic techniques for property-preserving abstraction [10,16,2] and bounded model checking [5] are two means of allowing systems with infinite state spaces to be checked. Powerful automatic decision procedures allow model-checker languages to support high-level specification constructs such as lambda expressions, set comprehensions and universal and existential quantifiers [6]. Hence, it is possible to model check specifications written in high-level languages [18].

It has become possible therefore to consider using model checkers to prove data refinements. There are two main challenges in doing so. Firstly, a model checker expects a single system on which to check properties. Hence, we need to combine the abstract and concrete systems into one system. Secondly, the properties checked are usually behavioural properties, i.e., properties of paths through the system's states. Therefore, we need to express the simulation conditions as behavioural properties. How this is done depends on how the abstract and concrete systems are combined. Hence, the two challenges are interrelated.

We have found that the branching time temporal logic CTL [8] is particularly suited to modelling simulation rules. In this paper, we show how CTL model checkers, e.g., NuSMV [1] or SAL [5], can be used to check the standard conditions for downward simulation. We do this both for systems

with a blocking, or guarded, interpretation of operations, as well as those with the more common non-blocking interpretation [7]. We plan to extend the approach to upward simulation in future work. In Section 2, we introduce the temporal logic CTL. In Section 3, we describe how downward simulation can be checked under a blocking semantics, and in Section 4 extend these ideas to a non-blocking semantics. Our approach is not specific to a particular state-based specification language nor a particular CTL model checker. In Section 5, we discuss our experience with instantiating our approach with Z specifications [20] in SAL. We conclude in Section 6.

2 CTL

Temporal logics are used to define properties of Kripke structures. Kripke structures are essentially state transitions systems with a *total* transition relation, i.e., where every state has at least one transition enabled. An infinite sequence of states through a Kripke structure (where each state is related to its successor by the transition relation) is referred to as a *path*.

CTL [8] is a branching time temporal logic meaning that its formulae are interpreted over all paths beginning in a given state of the Kripke structure. We write $M, s_0 \vDash f$ to denote that for Kripke structure M, the CTL formula f holds in state s_0.

Syntactically, we divide CTL formulae into three categories:

(i) those whose outermost operator, if any, is not a temporal operator,

(ii) those whose outer most operator is a temporal operator (**X** (next), **U** (until), **F** (eventually) or **G** (always)) prefixed with the existential path quantifier **E**, and

(iii) those whose outer most operator is a temporal operator prefixed with the universal path quantifier **A**.

The formulae in category (i) comprise atomic propositions on the states of the Kripke structure, as well as logical combinations of other CTL formulae from categories (i), (ii) and (iii). Specifically, if f_1 and f_2 are CTL formulae then so are $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$, $f_1 \Rightarrow f_2$ and $f_1 \Leftrightarrow f_2$.

The formulae in category (ii) express properties which are true on *at least one* path of the Kripke structure M starting from s_0. For example, $\mathbf{EX}\ f_1$, where f_1 is a CTL formula, states that for at least one path starting from the state s_0, f_1 holds in the *next* state. Similarly, $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ states that for at least one path starting from s_0, f_1 holds *until* some state where f_2 holds. Also, $\mathbf{EF}\ f_1$ states that for at least one path starting from s_0, f_1 *eventually* holds, and $\mathbf{EG}\ f_1$ states that for at least one path starting from s_0, f_1 *always* holds.

The formulae in category (iii) express properties which are true on *all* paths of M starting from s_0. For example, $\mathbf{AX}\ f_1$, where f_1 is a CTL formula, states that for all paths starting from the state s_0, f_1 holds in the *next* state. Similarly, $\mathbf{A}[f_1\ \mathbf{U}\ f_2]$ states that for all paths starting from s_0, f_1 holds *until* some state where f_2 holds. Also, $\mathbf{AF}\ f_1$ states that for all paths starting from s_0, f_1 *eventually* holds, and $\mathbf{AG}\ f_1$ states that for all paths starting from s_0, f_1 *always* holds.

More formally, the semantics of the CTL formulae introduced above can be given as follows (where p is an atomic proposition and f_1 and f_2 are CTL formulae).

Semantics of CTL

(i)

$$
\begin{array}{lll}
M, s_0 \vDash p & \text{iff} & p \text{ is true on } s_0 \\
M, s_0 \vDash \neg\, f_1 & \text{iff} & \text{not } M, s_0 \vDash f_1 \\
M, s_0 \vDash f_1 \wedge f_2 & \text{iff} & M, s_0 \vDash f_1 \text{ and } M, s_0 \vDash f_2 \\
M, s_0 \vDash f_1 \vee f_2 & \text{iff} & M, s_0 \vDash \neg\, (\neg\, f_1 \wedge \neg\, f_2) \\
M, s_0 \vDash f_1 \Rightarrow f_2 & \text{iff} & M, s_0 \vDash \neg\, f_1 \vee f_2 \\
M, s_0 \vDash f_1 \Leftrightarrow f_2 & \text{iff} & M, s_0 \vDash (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)
\end{array}
$$

(ii)

$$
\begin{array}{lll}
M, s_0 \vDash \mathbf{EX}\ f_1 & \text{iff} & \text{for some path } (s_0, s_1, \ldots),\ M, s_1 \vDash f_1 \\
M, s_0 \vDash \mathbf{E}[f_1\ \mathbf{U}\ f_2] & \text{iff} & \text{for some path } (s_0, s_1, \ldots), \\
 & & \qquad \text{there exists an } i, \\
 & & \qquad\quad M, s_i \vDash f_2 \text{ and} \\
 & & \qquad\quad \text{for all } j < i,\ M, s_j \vDash f_1 \\
M, s_0 \vDash \mathbf{EF}\ f_1 & \text{iff} & M, s_0 \vDash \mathbf{E}[\text{true}\ \mathbf{U}\ f_1] \\
M, s_0 \vDash \mathbf{EG}\ f_1 & \text{iff} & M, s_0 \vDash \neg\, \mathbf{AF}\ \neg\, f_1
\end{array}
$$

(iii) $M, s_0 \models \mathbf{AX}\ f_1$ iff for all paths $(s_0, s_1, \ldots)$, $M, s_1 \models f_1$

$M, s_0 \models \mathbf{A}[f_1\ \mathbf{U}\ f_2]$ iff for all paths $(s_0, s_1, \ldots)$,
there exists an i,
$M, s_i \models f_2$ and
for all $j < i$, $M, s_j \models f_1$

$M, s_0 \models \mathbf{AF}\ f_1$ iff $M, s_0 \models \mathbf{A}[\text{true}\ \mathbf{U}\ f_1]$

$M, s_0 \models \mathbf{AG}\ f_1$ iff $M, s_0 \models \neg\ \mathbf{EF}\ \neg\ f_1$

3 Blocking semantics

The blocking, or guarded, semantics of state-transition systems is usually adopted for specifications of reactive systems. Under this semantics, an operation has a guard outside of which it cannot occur, i.e., it is 'blocked' outside the guard. It has been used for state-based notations aimed specifically at concurrent systems [12,17].

Let a specified system comprise a set of states S, a non-empty set of initial states $I \subseteq S$, and a finite set of operations, or transitions, $\{Op_1, \ldots, Op_n\}$, each of which is a relation between states in S [3]. Under the blocking semantics, downward simulation is then defined as follows [7].

Definition 3.1 A specification $C = (CS, CI, \{COp_1, \ldots, COp_n\})$ is a downward simulation of a specification $A = (AS, AI, \{AOp_1, \ldots, AOp_n\})$, if there exists a retrieve relation R between AS and CS such that the following hold for all $i \in 1\,..\,n$.

(i) $\forall\, c \in CI \bullet \exists\, a \in AI \bullet a\ R\ c$

(ii) $\forall\, a \in AS;\ c \in CS \bullet$
$a\ R\ c \Rightarrow ((\exists\, a' \in AS \bullet a\ AOp_i\ a') \Leftrightarrow (\exists\, c' \in CS \bullet c\ COp_i\ c'))$

(iii) $\forall\, a \in AS;\ c, c' \in CS \bullet$
$a\ R\ c \wedge c\ COp_i\ c' \Rightarrow (\exists\, a' \in AS \bullet a'\ R\ c' \wedge a\ AOp_i\ a')$

Condition 1 of Definition 3.1 is known as *initialisation*. It requires that for every concrete initial state there is an initial abstract state related by the retrieve relation R.

Condition 2 is known as *applicability*. It requires that the abstract operations are only enabled in states related to concrete states where the corre-

[3] Input and output parameters of operations can be embedded in the states of S as described by Smith and Winter [19].

sponding concrete operations are enabled, and vice versa.

Condition 3 is known as *correctness*. It requires that whenever a concrete operation can result in a state change (t, t'), for any abstract state s related to t, the corresponding abstract operation can result in (s, s') such that s' is related to t'. That is, the effect of the concrete operation is consistent with the requirements of the corresponding abstract operation.

3.1 General approach

In this section, we discuss a general approach to checking downward simulations under a blocking semantics with a CTL model checker. We first provide systems for checking each of the downward simulation conditions individually. We then combine these systems into one in which all the conditions can be checked simultaneously. This latter system can also be used to check the conditions individually; something which is useful for finding problems when a refinement does not hold.

We have only considered constraints in our models that are *necessary* for checking the conditions. Further constraints could be added to make the state spaces of the models smaller, and hence model checking more efficient. Determining the optimal constraints to achieve this end, however, is left as future work.

To illustrate our approach, consider the following simple abstract and concrete specifications given in the style of Z [20]. The abstract system has a variable x which is initially 0 and may be incremented by 1 or 2. The concrete system has a variable y which is initially 0 and may be incremented by 1. Both systems have an upper bound on their variables of 10.

$$A \mathrel{\widehat=} [x : 0 \mathinner{..} 10] \qquad C \mathrel{\widehat=} [y : 0 \mathinner{..} 10]$$
$$AInit \mathrel{\widehat=} [A \mid x = 0] \qquad CInit \mathrel{\widehat=} [C \mid y = 0]$$
$$AOp \mathrel{\widehat=} [\Delta A \mid x' = x + 1 \lor x' = x + 2] \qquad COp \mathrel{\widehat=} [\Delta C \mid y' = y + 1]$$

The specification C in this example is a downward simulation of A under the retrieve relation $x = y$.

As mentioned in the introduction, we need to combine these systems in order to check the simulation conditions. As would be expected, the conditions of Definition 3.1 refer to both the abstract and concrete states. Hence, a combined system must have access to both. We will assume the state variables of the abstract and concrete systems are disjoint as in the example above. If they were not, they could be made disjoint by a systematic renaming. For example, a variable x could be renamed to $A.x$ in the abstract specification and $C.x$ in the concrete specification. The combined state can then include the

variables from both systems. For our example, we would have the declarations $x : 0..10$ and $y : 0 \ldots 10$ in the state of our combined system.

3.1.1 Initialisation

We begin by considering the initialisation condition. This condition requires that for each concrete initial state, we are able to find an abstract initial state related by the retrieve relation R. Hence, we require a means of having access to all concrete initial states in our combined system. One way to do this is to initialise the combined system so that the concrete part of the state is initialised. For our example, our combined system's state would be

$$M_{init} \mathrel{\widehat{=}} [x : 0 \ldots 10;\ y : 0 \ldots 10]$$

and would be initialised as follows.

$$Init_{init} \mathrel{\widehat{=}} [M_{init} \mid y = 0]$$

To check whether an abstract initial state related to a given concrete initial state exists, we introduce an operation $InitA_{init}$ which changes the abstract part of the state to an initial value and leaves the concrete part unchanged. For our example, this would be the following.

$$InitA_{init} \mathrel{\widehat{=}} [\Delta M_{init} \mid x' = 0 \wedge y' = y]$$

Note that since $InitA_{init}$ is enabled in any state, the transition relation is total as required for a Kripke structure. This will be true for any abstract specification except degenerate cases where there are no abstract initial states. For now, we will assume that the user will not provide such degenerate abstract specifications. We will return to this issue at the end of this section.

Given the above operation, the initialisation condition holds if the operation can be performed such that the resulting abstract and concrete parts of the state are related by R. For our example, this check is expressed in CTL as follows.

$$\mathbf{EX}\ x = y$$

That is, there exists a next state such that $x = y$. Note that the CTL operator **EX** allows us to determine whether it is possible to perform an operation and reach a particular state. This ability to existentially quantify over next states is what makes CTL particularly useful for capturing simulation conditions. If there were more than one abstract initial state, then the use of **EX** above means that only one of these need be related to the concrete initial state.

Our approach to checking initialisation is summarised (in a Z style) below: A and C represent the abstract and concrete states, respectively, $AInit$ and $CInit$ represent the abstract and concrete initialisations, respectively, and R represents the retrieve relation.

System: $M_{init} \mathrel{\widehat=} [A;\ C]$
$Init_{init} \mathrel{\widehat=} [M_{init} \mid CInit]$
$InitA_{init} \mathrel{\widehat=} [\Delta M_{init} \mid AInit \wedge \Xi C]$

Initialisation check: $\mathbf{EX}\ R$

3.1.2 Applicability

We now consider the applicability condition. To check applicability, we need to be able to determine whether each of the abstract and concrete operations can occur. CTL only allows propositions referring to state variables, however, not operations. Hence, we introduce a variable ev to the combined state to denote the name of the last operation that occurred, and we use a different font for the values of type ev. For our example, the state of the combined system for checking applicability is as follows (the *Choose* operation is explained below).

$$M_{app} \mathrel{\widehat=} [x : 0 \mathinner{..} 10;\ y : 0 \mathinner{..} 10;\ ev : \{\mathsf{AOp}, \mathsf{COp}, \mathsf{Choose}\}]$$

The applicability condition requires that an abstract operation can occur from an abstract state exactly when the corresponding concrete operation can occur from a concrete state related to the abstract state by the retrieve relation R. Hence, we require a means of having access to all combined states where the abstract and concrete parts are related. Note that the condition does not require the abstract and concrete states to be reachable; a point we will return to in Section 5. Once again we can do this by an appropriate initialisation of our combined system; in this case, to all states where R holds. For our example, we would initialise the combined system as follows. (The initial value of ev is not important and is left unspecified.)

$$Init_{app} \mathrel{\widehat=} [M_{app} \mid x = y]$$

We then introduce operations corresponding to the abstract and concrete operations. For our example, we have the following.

$$AOp_{app} \mathrel{\widehat=} [\Delta M_{app} \mid (x' = x + 1 \vee x' = x + 2) \wedge ev' = \mathsf{AOp}]$$
$$COp_{app} \mathrel{\widehat=} [\Delta M_{app} \mid y' = y + 1 \wedge ev' = \mathsf{COp}]$$

Since, in general, there will not be an operation enabled in all states, e.g., for the example, neither AOp_{app} nor COp_{app} are enabled when $x = 10$ and $y = 10$, we need to introduce a further operation to ensure the transition relation is total. This operation *Choose* is always enabled and simply chooses a new state; the actual state is not important and is left unspecified.

$$Choose_{app} \mathrel{\widehat{=}} [\Delta M_{app} \mid ev' = \mathsf{Choose}]$$

The specification now represents a Kripke structure and the applicability condition holds if whenever an abstract operation can be performed, the corresponding concrete operation can be performed, and vice versa. For our example, this check can be expressed in CTL as follows.

$$\mathbf{EX}\ (ev = \mathsf{AOp}) \Leftrightarrow \mathbf{EX}\ (ev = \mathsf{COp})$$

(Note that we write $ev = \mathsf{AOp}$ and not $ev = \mathsf{AOp_{app}}$ since the type of ev is a set of names, not the actual operations themselves.)

Our approach to checking applicability is summarised (in Z style) below: $AOp_1, \ldots, AOp_n$ represent the abstract operations and $COp_1, \ldots, COp_n$, the corresponding concrete operations.

$$\begin{array}{ll}
\text{System:} & M_{app} \mathrel{\widehat{=}} [A;\ C;\ ev : \{\mathsf{AOp_1}, \ldots, \mathsf{AOp_n}, \mathsf{COp_1}, \ldots, \mathsf{COp_n}, \mathsf{Choose}\}] \\
& Init_{app} \mathrel{\widehat{=}} [M_{app} \mid R] \\
& AOp_{1,app} \mathrel{\widehat{=}} [\Delta M_{app} \mid AOp_1 \wedge ev' = \mathsf{AOp_1}] \\
& \vdots \\
& AOp_{n,app} \mathrel{\widehat{=}} [\Delta M_{app} \mid AOp_n \wedge ev' = \mathsf{AOp_n}] \\
& COp_{1,app} \mathrel{\widehat{=}} [\Delta M_{app} \mid COp_1 \wedge ev' = \mathsf{COp_1}] \\
& \vdots \\
& COp_{n,app} \mathrel{\widehat{=}} [\Delta M_{app} \mid COp_n \wedge ev' = \mathsf{COp_n}] \\
& Choose_{app} \mathrel{\widehat{=}} [\Delta M_{app} \mid ev' = \mathsf{Choose}]
\end{array}$$

$$\begin{array}{ll}
\text{Applicability check:} & (\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Leftrightarrow \mathbf{EX}\ (ev = \mathsf{COp_1})) \wedge \\
& \vdots \\
& (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Leftrightarrow \mathbf{EX}\ (ev = \mathsf{COp_n}))
\end{array}$$

3.1.3 Correctness

We finally come to correctness. As with applicability, we need to be able to refer to the occurrence of operations to check correctness. Hence, we again introduce a variable ev to denote the last operation that occurred. The state

of the combined system for our example is as for checking the applicability condition.

$$M_{corr} \mathrel{\widehat{=}} M_{app}$$

The correctness condition requires that an abstract operation can occur from an abstract state when the corresponding concrete operation can occur from a concrete state related to the abstract state by the retrieve relation R. Hence, again we initialise our combined system to states where R holds. For our example, we would initialise the combined system as for the applicability condition.

$$Init_{corr} \mathrel{\widehat{=}} Init_{app}$$

The correctness condition requires, furthermore, that any state reached by performing the concrete operation is related by R to an abstract state reached by performing the abstract operation. We can capture this in CTL if the operations in the combined system corresponding to the abstract operations do not change the concrete state, and those corresponding to the concrete operations do not change the abstract state. That is, for our example, we have the following operations.

$$AOp_{corr} \mathrel{\widehat{=}} [AOp_{app} \mid y' = y]$$
$$COp_{corr} \mathrel{\widehat{=}} [COp_{app} \mid x' = x]$$

This allows us to perform the operations COp_{corr} and AOp_{corr} in sequence so that the abstract part of the final state reached is identical to that which could have been reached by performing only AOp_{corr}, and the concrete part is identical to that which could have been reached by performing only COp_{corr}.

We again need a 'choose' operation to ensure the transition relation is total.

$$Choose_{corr} \mathrel{\widehat{=}} Choose_{app}$$

The correctness condition then holds if, after a concrete operation is performed, the corresponding abstract operation can be performed and result in a state where R holds. For our example, this check can be expressed in CTL as follows.

$$\mathbf{AX}\ (ev = \mathsf{COp} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp} \wedge x = y))$$

Note the use of the CTL operator $\mathbf{AX}$ to ensure that all post-states of COp_{corr} are considered. The $\mathbf{EX}$ operator is in the scope of the $\mathbf{AX}$ operator and hence

quantifies over next states of states reached by performing COp_{corr}. (For this reason we need not be concerned with the value of ev after the initialisation, since the **AX** looks at only those states reached by performing COp_{corr}.)

Our approach to checking correctness is summarised (in Z style) below.

$$
\begin{array}{ll}
\text{System:} & M_{corr} \mathrel{\widehat=} [A;\ C;\ ev : \{\mathsf{AOp_1}, \ldots, \mathsf{AOp_n}, \mathsf{COp_1}, \ldots, \mathsf{COp_n}, \mathsf{Choose}\}] \\
& Init_{corr} \mathrel{\widehat=} [M_{corr} \mid R] \\
& AOp_{1,corr} \mathrel{\widehat=} [\Delta M_{corr} \mid AOp_1 \land \Xi C \land ev' = \mathsf{AOp_1}] \\
& \vdots \\
& AOp_{n,corr} \mathrel{\widehat=} [\Delta M_{corr} \mid AOp_n \land \Xi C \land ev' = \mathsf{AOp_n}] \\
& COp_{1,corr} \mathrel{\widehat=} [\Delta M_{corr} \mid COp_1 \land \Xi A \land ev' = \mathsf{COp_1}] \\
& \vdots \\
& COp_{n,corr} \mathrel{\widehat=} [\Delta M_{corr} \mid COp_n \land \Xi A \land ev' = \mathsf{COp_n}] \\
& Choose_{corr} \mathrel{\widehat=} [\Delta M_{corr} \mid ev' = \mathsf{Choose}]
\end{array}
$$

$$
\begin{array}{ll}
\text{Correctness check:} & \mathbf{AX}\ (ev = \mathsf{COp_1} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_1} \land R)) \land \\
& \vdots \\
& \mathbf{AX}\ (ev = \mathsf{COp_n} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_n} \land R))
\end{array}
$$

3.1.4 Downward simulation

We have shown how to construct three systems each of which can be used to check one of the downward simulation conditions. To check all conditions simultaneously, we now propose a system which combines those above.

The only difference between the system for the applicability condition M_{app} and that for the correctness condition M_{corr}, is the inclusion in M_{corr} of the constraint that abstract states do not change in the operations corresponding to the concrete operations, and vice versa. These constraints have no effect on the verity of the CTL formula for checking applicability. Hence, both applicability and correctness can be checked on M_{corr}.

To check the initialisation condition requires the addition of the *InitA* operation. It must also be included in the type of ev to allow transitions corresponding to the operation to be identified. For our example, the state required is as follows.

$$M_{ds} \mathrel{\widehat=} [x : 0 \mathinner{..} 10;\ y : 0 \mathinner{..} 10;\ ev : \{\mathsf{InitA}, \mathsf{AOp}, \mathsf{COp}, \mathsf{Choose}\}]$$

The initialisation check also requires a different set of initial states. To accommodate all checks, we widen the initialisation to allow all required initial states. For our example, the initialisation is as follows.

$$Init_{ds} \mathrel{\widehat{=}} [M_{ds} \mid (y = 0 \lor x = y)]$$

The operations appear as before, except for the inclusion of the predicate $ev' = \mathsf{InitA}$ in $InitA$. For our example, we have the following.

$$\begin{array}{l} InitA_{ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid x' = 0 \land y' = y \land ev' = \mathsf{InitA}] \\ AOp_{ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid (x' = x + 1 \lor x' = x + 2) \land y' = y \land ev' = \mathsf{AOp}] \\ COp_{ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid y' = y + 1 \land x' = x \land ev' = \mathsf{COp}] \\ Choose_{ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid ev' = \mathsf{Choose}] \end{array}$$

We can then check each of the conditions as before, when restricted to the appropriate initial states. For our example, the initialisation check is performed on initial states in which $y = 0$.

$$y = 0 \Rightarrow \mathbf{EX}\ (ev = \mathsf{InitA} \land x = y)$$

The applicability check is performed on states in which $x = y$.

$$x = y \Rightarrow (\mathbf{EX}\ (ev = \mathsf{AOp}) \Leftrightarrow \mathbf{EX}\ (ev = \mathsf{COp}))$$

Similarly, the correctness check is performed on states in which $x = y$.

$$x = y \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp} \land x = y))$$

Our approach to checking downward simulation is summarised (in Z style) below.

$$\begin{array}{ll} \text{System:} & M_{ds} \mathrel{\widehat{=}} [A;\ C; \\ & \qquad ev : \{\mathsf{InitA}, \mathsf{AOp_1}, \ldots, \mathsf{AOp_n}, \mathsf{COp_1}, \ldots, \mathsf{COp_n}, \mathsf{Choose}\}] \\ & Init_{ds} \mathrel{\widehat{=}} [M_{ds} \mid CInit \lor R] \\ & InitA_{ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid AInit \land \Xi C \land ev' = \mathsf{InitA}] \\ & AOp_{1,ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid AOp_1 \land \Xi C \land ev' = \mathsf{AOp_1}] \\ & \vdots \\ & AOp_{n,ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid AOp_n \land \Xi C \land ev' = \mathsf{AOp_n}] \\ & COp_{1,ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid COp_1 \land \Xi A \land ev' = \mathsf{COp_1}] \\ & \vdots \\ & COp_{n,ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid COp_n \land \Xi A \land ev' = \mathsf{COp_n}] \\ & Choose_{ds} \mathrel{\widehat{=}} [\Delta M_{ds} \mid ev' = \mathsf{Choose}] \end{array}$$

Downward simulation check:

$$
\begin{array}{l}
(CInit \Rightarrow \mathbf{EX}\ (ev = \mathsf{InitA} \wedge R)) \\
\wedge \\
(R \Rightarrow (\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Leftrightarrow \mathbf{EX}\ (ev = \mathsf{COp_1}) \wedge \\
\qquad \vdots \\
\qquad \mathbf{EX}\ (ev = \mathsf{AOp_n}) \Leftrightarrow \mathbf{EX}\ (ev = \mathsf{COp_n}))) \\
\wedge \\
(R \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_1} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_1} \wedge R)) \wedge \\
\qquad \vdots \\
\qquad \mathbf{AX}\ (ev = \mathsf{COp_n} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_n} \wedge R)))
\end{array}
$$

The *Choose* operation ensures that the transition relation is total as required. Hence, this approach will work even for degenerate abstract specifications in which there are no initial states. Alternatively, we could make the existence of abstract initial states a requirement on the approach, and hence be able to remove the *Choose* operation from the system above.

4 Non-blocking semantics

The non-blocking semantics of state-transitions systems is the most common one. It is used for most popular state-based specification languages such as Z [20]. Under this semantics, an operation has a precondition outside of which its behaviour is undefined. Its main use is in the specification of sequential systems.

Given a system as described at the beginning of Section 3, under a non-blocking semantics, downward simulation is defined as follows [7].

Definition 4.1 A specification $C = (CS, CI, \{COp_1, \ldots, COp_n\})$ is a downward simulation of a specification $A = (AS, AI, \{AOp_1, \ldots, AOp_n\})$, if there exists a retrieve relation R between AS and CS such that the following hold for all $i \in 1 \,..\, n$.

(i) $\forall\, c \in CI \bullet \exists\, a \in AI \bullet a\ R\ c$

(ii) $\forall\, a \in AS;\ c \in CS \bullet$
$a\ R\ c \Rightarrow ((\exists\, a' \in AS \bullet a\ AOp_i\ a') \Rightarrow (\exists\, c' \in CS \bullet c\ COp_i\ c'))$

(iii) $\forall\, a \in AS;\ c, c' \in CS \bullet$
$(\exists\, a' \in AS \bullet a\ AOp_i\ a') \wedge a\ R\ c \wedge c\ COP_i\ c' \Rightarrow$
$(\exists\, a' \in AS \bullet a'\ R\ c' \wedge a\ AOp_i\ a')$

Condition 1 of Definition 4.1 is the initialisation condition. It requires that for every concrete initial state there is an initial abstract state related by the

retrieve relation R.

Condition 2 is the applicability condition. It requires that abstract operations are only enabled in states related to concrete states where the corresponding concrete operations are enabled.

Condition 3 is the correctness condition. It requires that whenever a concrete operation can result in a state change (t, t'), for any abstract state s related to t, if the corresponding abstract operation is enabled then it can result in (s, s') such that s' is related to t'.

4.1 General approach

The general approach to checking downward simulation under a non-blocking semantics using a CTL model checker is very similar to that for a blocking semantics. We use the same systems for the individual conditions, as well as for checking all the conditions simultaneously.

4.1.1 Initialisation

The initialisation condition is identical to that of the blocking semantics. Hence, it is checked in an identical fashion.

4.1.2 Applicability

The applicability condition only differs from that of the blocking semantics by having implication rather than equivalence between the predicates stating that the abstract and concrete operations are enabled. Under the same system M_{app} used for the blocking semantics, we can express the applicability check for our example of the previous section as follows.

$$\mathbf{EX}\ (ev = \mathsf{AOp}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp})$$

More generally, for abstract operations $AOp_1, \ldots, AOp_n$ and corresponding concrete operations $COp_1, \ldots, COp_n$, we have the following CTL formula.

$$\begin{array}{l}(\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp_1})) \wedge \\ \vdots \\ (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp_n}))\end{array}$$

4.1.3 Correctness

The correctness condition is similar to that of the blocking model but has an extra antecedent which requires that the abstract operation is enabled.

This is required since abstract operations are not always enabled when the corresponding concrete ones are.

Under the same system M_{corr} used for the blocking semantics, we can express the correctness check for our example of the previous section as follows (again remembering ev is a set of names).

$$\mathbf{EX}\ (ev = \mathsf{AOp}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp} \wedge x = y))$$

More generally, we have the following CTL formula (where R represents the retrieve relation).

$$\begin{array}{l}(\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_1} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_1} \wedge R))) \wedge \\ \vdots \\ (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_n} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_n} \wedge R)))\end{array}$$

Recall that in this model for correctness, M_{corr} is initialised with R true, that is we are already in a state where A and C are related by the retrieve relation. Hence it is not necessary to prefix the formulae with $R \Rightarrow$. However, it is necessary below because the model for downward simulation, M_{ds}, can be initialised with either R true or *CInit* true.

4.1.4 Downward simulation

To check all the conditions simultaneously, we follow the approach explained for the blocking semantics. The general CTL formula for downward simulation checking is the following (where *CInit* represents the concrete initialisation).

$$\begin{array}{l}(\mathit{CInit} \Rightarrow \mathbf{EX}\ (ev = \mathsf{InitA} \wedge R)) \\ \wedge \\ (R \Rightarrow (\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp_1})) \wedge \\ \qquad \vdots \\ \qquad (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp_n}))) \\ \wedge \\ (R \Rightarrow (\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_1} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_1} \wedge R))) \\ \qquad \wedge \\ \qquad \vdots \\ \qquad (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_n} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_n} \wedge R))))\end{array}$$

5 Discussion

The approach presented in the previous section can be applied to any state-based specification language and any CTL model checker which supports the specification language. We have experimented with the approach using Z and its encoding in SAL [18].

So far our experiments have not utilised the full power of the SAL tools. We have therefore been restricted to specifications with a finite and relatively small state space. SAL supports many optimisations features, as well as a variable abstraction facility which can effectively be used to ignore variables not influencing a property we wish to prove. These features need to be investigated as a means of extending the size of the specifications we can handle. In addition, future versions of SAL are expected to support predicate abstraction [10,16,2]; we view this as essential for using our approach with much larger examples.

Using SAL we have been able to detect subtle errors with a minimum of effort once the specifications have been written in the appropriate input format. For example, in an initial version of the example on pp 271-273 of [21] the retrieve relation was too weak [4] which was not immediately obvious without resorting to completing the entire proof. The encoding of downward simulations in SAL picked up on the error with ease, although deciphering the counter-example required some understanding of the problem.

We are not the first to detect this error using tool support. It was previously discovered by Robinson who developed a means of automatically checking downward simulations using the Possum Z animator [15]. His approach relies on Possum's ability to evaluate complex Z predicates involving quantification over the abstract and concrete specifications' states. We have also encoded Robinson's approach in SAL and, while it is more efficient for very small state spaces, it becomes computationally more expensive than our approach when the size of the state space increases.

The issue in the above example was due to the retrieve relation not covering unreachable states, whereas the specifications were, in fact, related by downward simulation under a strengthened retrieve relation. We conjecture that this is a commonly occurring error since specifiers are usually more focused on the reachable states of their systems.

Unreachable states do not affect whether one specification is a downward simulation of another. The reason the downward simulation conditions consider unreachable states is that otherwise the reachable states would need to

[4] The error was corrected by the addition of the invariant to one of the specifications in subsequent versions of the text.

be calculated as part of discharging the conditions. This is not something easily done by hand, or with a theorem prover. It is, however, an intrinsic part of the model checking process, i.e., since model checking involves an exhaustive search over a system's (reachable) state space. Hence, we could restrict our approach to only check reachable states and, therefore, only require a retrieve relation that relates reachable states; one that would arguably fit better with the specifier's view of the system.

We would thus not be checking the full downward simulation conditions; however, the restriction to reachable states is sufficient to know that one specification is a downward simulation of another.

To do this, we would need to initialise the system to one in which the concrete state is initialised and not allow abstract operations to be performed before the operation which initialises the abstract state. This could be done by the inclusion of a "Boolean" variable b denoting whether or not the abstract state has been initialised as follows.

$$
\begin{array}{l}
M_{reach} \mathrel{\widehat{=}} [A;\ C;\ b : \{0,1\};\\
\qquad\qquad ev : \{\mathsf{Init}, \mathsf{InitA}, \mathsf{AOp_1}, \ldots, \mathsf{AOp_n}, \mathsf{COp_1}, \ldots, \mathsf{COp_n}, \mathsf{Choose}\}]\\
Init_{reach} \mathrel{\widehat{=}} [M_{reach} \mid CInit \land b = 0 \land ev = \mathsf{Init}]\\
InitA_{reach} \mathrel{\widehat{=}} [\Delta M_{reach} \mid b = 0 \land AInit \land \Xi C \land b' = 1 \land ev' = \mathsf{InitA}]\\
AOp_{1,reach} \mathrel{\widehat{=}} [\Delta M_{reach} \mid b = 1 \land AOp_1 \land \Xi C \land b' = b \land ev' = \mathsf{AOp_1}]\\
\vdots\\
AOp_{n,reach} \mathrel{\widehat{=}} [\Delta M_{reach} \mid b = 1 \land AOp_n \land \Xi C \land b' = b \land ev' = \mathsf{AOp_n}]\\
COp_{1,reach} \mathrel{\widehat{=}} [\Delta M_{reach} \mid COp_1 \land \Xi A \land b' = b \land ev' = \mathsf{COp_1}]\\
\vdots\\
COp_{n,reach} \mathrel{\widehat{=}} [\Delta M_{reach} \mid COp_n \land \Xi A \land b' = b \land ev' = \mathsf{COp_n}]\\
Choose_{reach} \mathrel{\widehat{=}} [\Delta M_{reach} \mid ev' = \mathsf{Choose}]
\end{array}
$$

Then, all states in M_{reach} would comprise reachable abstract and concrete states.

Since the initial state of M_{reach} is restricted to those where the concrete part of the state is initialised, the condition for checking initialisation would be simplified as follows.

$$\mathbf{EX}\ (ev = \mathsf{InitA} \land R)$$

The applicability and correctness conditions would need to be modified so that the checks are performed on all states in M where R holds. This can be done for the non-blocking model by simply prefixing the formulae with the temporal operator $\mathbf{AG}$.

Applicability would be checked by the following formula.

$$\begin{array}{l}\mathbf{AG}\ (R \Rightarrow (\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp_1})) \wedge \\ \qquad\qquad \vdots \\ \qquad\qquad (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{COp_n})))\end{array}$$

Similarly, correctness would be checked by the following formula.

$$\begin{array}{l}\mathbf{AG}\ (R \Rightarrow \\ \qquad (\mathbf{EX}\ (ev = \mathsf{AOp_1}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_1} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_1} \wedge R))) \\ \qquad \wedge \\ \qquad \vdots \\ \qquad (\mathbf{EX}\ (ev = \mathsf{AOp_n}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{COp_n} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AOp_n} \wedge R))))\end{array}$$

For the blocking model, we would additionally have to check that $b = 1$ in the antecedent of the applicability and correctness checks. This is not necessary in the formulae above due to antecedents of the form $\mathbf{EX}\ (ev = \mathsf{AOp})$ which imply that $b = 1$.

We conclude this section with a brief discussion of another approach to automatically checking state-based refinement. There have been a number of encoding of subsets of Z-based languages in the CSP model checker FDR [9,14,13]. FDR is not a temporal logic model checker. Rather, it checks that refinement holds between two specifications. It does this by comparing the failures/divergences semantics of the specifications; an approach which is equivalent to simulation-based refinement [12,11].

This approach has the advantage that no retrieve relation is required. However, since it is not possible to check individual simulation conditions, finding problems when refinements do not hold may be more difficult. Also, since FDR was developed for a process algebra, rather than a state-based notation, encoding such notations is more difficult; to date, there is no full encoding of Z in FDR for example. We see the two approaches as being complementary.

6 Conclusion

In this paper, we have shown how downward simulation can be checked using existing temporal logic model checkers. In particular, we have shown how the branching time temporal logic CTL can be used to encode the standard downward simulation conditions. We did this for both a blocking, or guarded, interpretation of operations (often used when specifying reactive systems) as well as the more common non-blocking interpretation of operations used in many state-based specification languages (for modelling sequential systems).

We have used the approach to check downward simulation between Z spec-

ifications using the SAL CTL model checker. The approach, however, is general enough to use with any state-based specification language, and any CTL model checker in which the language can be encoded. We envisage the approach becoming more applicable as it takes advantage of the current efforts in the temporal logic model checking community to extend model checking to systems with larger, and even infinite, state spaces.

Acknowledgement

We would like to thank Leonardo de Moura and John Rushby for their help with our use of SAL. This work and paper has also benefited from helpful comments from Kirsten Winter and Ian Hayes.

References

[1] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[2] 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.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

[5] 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.

[6] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2), SRI International, 2003.

[7] J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, 2001.

[8] 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.

[9] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods (IFM'99)*, pages 315–334. Springer-Verlag, 1999.

[10] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[11] J. He. Process refinement. In J. McDermid, editor, *The Theory and Practice of Refinement*. Butterworths, 1989.

[12] M. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[13] G. Kassel and G. Smith. Model checking Object-Z classes: Some experiments with FDR. In *Asia-Pacific Software Engineering Conference (APSEC 2001)*. IEEE Computer Society Press, 2001.

[14] A. Mota and A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40:59–96, 2001.

[15] N. Robinson. Checking Z data refinement using an animation tool. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *International Conference of Z and B Users (ZB 2002)*, volume 2272 of *LNCS*, pages 62–81. Springer-Verlag, 2002.

[16] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 443–453. Springer-Verlag, 1999.

[17] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[18] G. Smith and L. Wildman. Model checking Z specifications using SAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *International Conference of Z and B Users (ZB 2005)*, volume 3455 of *LNCS*, pages 87–105. Springer-Verlag, 2005.

[19] G. Smith and K. Winter. Proving temporal properties of Z specificatons using abstraction. In D. Bert, J.P. Bowen, S. King, and M. Waldén, editors, *International Conference of Z and B Users (ZB 2003)*, volume 2651 of *LNCS*, pages 260–279. Springer-Verlag, 2003.

[20] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[21] J. Woodcock and J. Davies. *Using Z: Specification, refinement, and proof*. Prentice Hall, 1996.