Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 201 (2008) 155–175
www.elsevier.com/locate/entcs

Using Model Checking to Automatically Find Retrieve Relations

John Derrick[1]

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

Graeme Smith[2]

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

Abstract

Downward and upward simulations form a sound and jointly complete methodology for verifying relational data refinement in state-based specification languages such as Z and B. In previous work, we showed how both downward and upward simulation conditions can be discharged using a CTL model checker. The approach was implemented in the SAL tool suite. Given the retrieve relation, each of the simulation conditions can be proven fully automatically. It has been recognised, however, that finding retrieve relations is often very hard. In this paper, we show how it is feasible to use the SAL model checkers to also generate retrieve relations.

Keywords: Data refinement, Z, simulations, retrieve relation, SAL, model checking, tool support.

1 Introduction

Data refinement [8] is a formal notion of development based around the idea that a concrete specification can be substituted for an abstract one as long as its behaviour is consistent with that defined in the abstract specification. In a state-based setting, as typified by the formal specification languages Z [29] and B [1], data refinements are usually verified by defining a relation, referred

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

1571-0661 © 2008 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2008.02.019

to as the *retrieve relation*, between the states of the two specifications and proving a set of downward and/or upward simulation conditions.

Proving these conditions by hand, even for simple systems, is at best tedious and at worst error-prone. Hence, tool support for proving data refinements is generally considered necessary. Smith and Wildman [27] have shown how Z can be encoded in the input notation of the SAL tool suite [6]. Building on this work, we have shown how both the downward and upward simulation conditions for proving Z refinements can be discharged using SAL's CTL model checker [26]. Given the retrieve relation, each of the simulation conditions can be proven fully automatically.

It has been recognised, however, that finding retrieve relations is often very hard [4,10,22,3]. Robinson [20,23] using animation techniques to check data refinements, for example, has found published refinement case studies where retrieve relations are incorrect. It is, therefore, desirable to have refinement checking tool support which does not require the manual provision of a retrieve relation.

A number of existing approaches already achieve this. For example, there have been a number of encodings of subsets of Z-based languages in the CSP model checker FDR [12,19,16]. FDR checks refinements by comparing the failures/divergences semantics of the specifications [24]; an approach which is equivalent to simulation-based refinement [15,13] but does not require a retrieve relation. Although FDR does have an expressive functional language as part of its input notation, it was developed for a process algebra, rather than a state-based notation. Encoding such notations in FDR is therefore not straightforward [9]; to date, there is no full encoding of Z in FDR for example.

The other aforementioned approaches attempt to derive a valid retrieve relation, the existence of which implies refinement holds. Bolton [3], for example, uses the Alloy Analyzer [14], a SAT-based verification tool, to automatically find retrieve relations for data refinements in Z and Object-Z [25]. This approach requires, however, that the relational semantics of the specification, rather than the specification itself, be encoded in the Alloy Analyzer. It is not clear, therefore, whether the translation from a specification to this encoding can be automated.

Leuschel and Butler [18] have developed an algorithm for constructing a retrieve relation between the states of an abstract and concrete B specification. The algorithm which traverses the state spaces of the specifications has been integrated into the ProB model checker [17]. Their approach is limited, however, to either trace or singleton failures refinement, which are either weaker (trace) or stronger (singleton failures) than standard data refinement, and so does not guarantee the same notion of consistency between the concrete and

abstract specification.

Robinson [21,22] has presented two algorithms for finding retrieve relations between the concrete and abstract states of specifications written in the action systems formalism [2]. The first [21] populates a relation between concrete and abstract states and then deletes states until either a retrieve relation is found, or it is shown that one cannot be found. The second [22], like the approach of Leuschel and Butler, attempts to construct a retrieve relation while traversing the abstract and concrete state spaces. Both of Robinson's algorithms support standard data refinement. However, while the former supports both downward and upward simulation, the latter only supports downward simulation.

In this paper we investigate ways to automatically find retrieve relations for Z specifications using SAL's model checkers. In Section 2 we discuss the relevant background providing brief introductions to the SAL notation, the temporal logics LTL and CTL, and Z data refinement. In Section 3 we describe an encoding of the first algorithm of Robinson, adapted to Z. This is followed by Section 4 where, using our previous work on model checking Z refinements, we show how a retrieve relation can be found directly. We discuss the approaches and their limitations in Section 5.

2 Background

2.1 SAL

SAL [6] is a tool-suite for the analysis and verification of systems specified as state-transition systems. Its aim is to allow different verification tools to be combined, all working on an input language designed as a format into which programming and specification languages can be translated. The input language provides a range of features to support this aim, and can, in fact, be used as a specification language in its own right. The tool-suite currently comprises a simulator and four model checkers, including LTL and CTL model checkers which we use here.

A specification is given as a SAL context. For example, below we have a context *alloc* representing the specification of a unique number allocator. A context groups together a number of definitions which include types, constants and modules for describing the state transition system. In our example we declare a type NAT, a subrange of the naturals whose maximum element is defined by the constant MAX. The module *main* declares a local variable *as*, a set of NAT [3], and an output variable *out* of type NAT (representing the

[3] See Smith and Wildman [27] for an explanation of the use of sets and set notation in SAL.

current output from the system).

```
alloc: CONTEXT =
BEGIN

  MAX: NATURAL = 4;
  NAT: TYPE = [0..MAX];

  main: MODULE =
  BEGIN
    LOCAL as: set{NAT}!Set
    OUTPUT out: NAT
    INITIALIZATION
      as = set{NAT}!empty_set
    TRANSITION
      [aop: NOT(set{NAT}!contains?(as,out'))
            --> as' = set{NAT}!union(as,{out'});
                out' IN {n:NAT|true}
       []
       skip: true -->
      ]
  END;

END
```

The initialisation of the system requires *as* to be the empty set. There are two transitions labelled *aop* and *skip*. The latter makes the system deadlock free; a condition that is necessary for the correct functioning of the model checkers. The former chooses a new output value out' which is not contained in *as*.

Both transitions are specified using guarded commands of the form `guard --> assignments`. The guard is an arbitrary predicate which may refer to primed (i.e., post-state) variables. A guarded command is able to occur whenever there exists values of the primed variables which satisfy both the guard and the assignments. The assignments may be nondeterministic as illustrated by the assignment to out' in the transition *aop*. Any variables not explicitly assigned a value are unchanged.

We can think of the SAL specification above as a finite-state implementation of the following Z specification [27].

AState

$as : \mathbb{P}\,\mathbb{N}$

AInit

$as = \varnothing$

$$
\begin{array}{|l}
\hline
AOp \\
\Delta AState \\
out! : \mathbb{N} \\
\hline
out! \notin as \\
as' = as \cup \{out!\} \\
\hline
\end{array}
$$

2.2 Temporal logic

The SAL model checkers verify temporal properties of system specifications. Specifically, they support proving properties expressed in LTL and CTL. LTL [11] is a linear-time temporal logic. Its formulae are generated from the following rules of syntax.

atomic propositions are formulae
if P and Q are formulae, then $\neg\, P$ and $P \wedge Q$ are formulae
if P and Q are formulae, then $\mathbf{X}\ P$ and $P\ \mathbf{U}\ Q$ are formulae

The operator $\mathbf{X}$ is read as "next", and $\mathbf{U}$ as "until": $P\ \mathbf{U}\ Q$ states that P is true until Q becomes true (and Q must eventually become true). The following abbreviations are also commonly used.

$$
\begin{array}{lcl}
P \vee Q & \equiv & \neg\, (\neg\, P \wedge \neg\, Q) \\
P \Rightarrow Q & \equiv & \neg\, P \vee Q \\
\mathbf{F}\ P & \equiv & true\ \mathbf{U}\ P \quad \text{(read "eventually P")} \\
\mathbf{G}\ P & \equiv & \neg\, \mathbf{F}\ \neg\, P \quad \text{(read "always P")}
\end{array}
$$

An alternative to LTL is CTL [11], which is a branching-time temporal logic. Its formulae are state formulae generated from the following rules of syntax.

state formulae:
atomic propositions are state formulae
if P and Q are state formulae, then $\neg\, P$ and $P \wedge Q$ are state formulae
if P is a path formula, then $\mathbf{E}P$ is a state formula

path formulae:
if P and Q are state formulae, then $\mathbf{X}\ P$ and $P\ \mathbf{U}\ Q$ are path formulae

$\mathbf{X}$ and $\mathbf{U}$ have the same interpretations as in LTL, and $\mathbf{E}$ is an existential quantifier for paths. It is read as "there exists a path (from the current state)". As well as the abbreviations defined for LTL above, the following abbreviation for universal path quantification is commonly used.

$$\mathbf{A}P \quad \equiv \quad \neg\mathbf{E}\neg P \quad \text{(read "for all paths (from the current state)")}$$

The semantics of these logics are defined over Kripke structures, which are state transitions systems with a *total* transition relation, i.e., where every state has at least one transition enabled.

The semantics of LTL is defined on *paths*, i.e., infinite sequences of states of a temporal structure where each pair of consecutive states is related by the transition relation of the temporal structure. Given a Kripke structure A and LTL property P, $A \models P$ in LTL means that P holds on all paths originating from initial states of A.

Whereas the semantics of LTL is defined on paths, that of CTL is defined on states. Thus given a Kripke structure A and CTL property P, $A \models P$ in CTL means that P holds on all initial states of A.

2.3 Refinement

Data refinement [7] is a formal notion of development, based around the idea that a concrete specification can be substituted for an abstract one as long as its behaviour is consistent with that defined in the abstract specification. An example refinement of the Z specification in Section 2.1 is given below.

CState

$cx : \mathbb{Z}$

CInit

$cx = -1$

COp

$\Delta CState$

$out! : \mathbb{N}$

$cx' = cx + 1$

$out! = cx'$

A finite-state implementation of this can be written in SAL's input language as follows.

```
concrete_alloc: CONTEXT =
BEGIN

MAX: NATURAL = 4;
NAT: TYPE = [0..MAX];
INT: TYPE = [-1..MAX];
```

```
main: MODULE =
  BEGIN
    LOCAL cx: INT
    OUTPUT out: NAT
    INITIALIZATION
      cx = -1
    TRANSITION
      [cop: true --> cx' = cx + 1;
                     out' = cx'
       []
       skip: true -->
      ]
  END;

END
```

In a state-based setting such as this data refinements are verified by defining a relation (called a *retrieve relation*) between the two specifications and verifying a set of simulation conditions. In general there are two forms the simulation rules take depending on the interpretation given to an operation, specifically that given to the operation's guard or precondition [8]. The two interpretations are often called the *blocking* and *non-blocking* semantics. We consider the latter, i.e., the standard, approach in this paper.

We give the definition of simulations over transition systems, and note that these are equivalent to those given in terms of Z directly [8]. 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. Under the non-blocking semantics, downward simulation is defined as follows [8] [4].

Definition 2.1 [Downward simulation]
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) $\forall\, a \in AS;\ c, c' \in CS \bullet (\exists\, a' \in AS \bullet a\ AOp_i\ a') \wedge a\ R\ c \wedge c\ COp_i\ c' \Rightarrow$

[4] For the sake of readability we have omitted the necessary quantification over the inputs and outputs in this and the following definition.

$$(\exists\, a' \in AS \bullet a'\ R\ c' \wedge a\ AOp_i\ a')$$

Condition 1 is known as *initialisation*, condition 2 is known as *applicability*, and condition 3 as *correctness*. To verify a data refinement it is sometimes necessary to use an alternative kind of simulation known as an *upward* simulation.

Definition 2.2 [Upward simulation]
A specification $C = (CS, CI, \{COp_1, \ldots, COp_n\})$ is an upward 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) $\forall\, a \in AS;\ c \in CI \bullet a\ R\ c \Rightarrow a \in AI$

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

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

(iv) $\forall\, c \in CS \bullet \exists\, a \in AS \bullet a\ R\ c$

Note that in an upward simulation totality of the retrieve relation is required (condition 4). In general, both types of simulation are needed to form a complete methodology for verifying refinements [7].

The above refinement of the unique number allocator can be verified using downward simulation. To do so, it is necessary to define the correct retrieve relation between the abstract and concrete specifications. In this example the following will suffice:

R

$AState$ $CState$
$as = \{n : \mathbb{N} \mid n \leqslant cx\}$

Although this retrieve relation is obvious, retrieve relations can, in general, be very subtle to establish correctly. The next sections discuss approaches by which a retrieve relation can be derived automatically using SAL's model checkers.

3 An algorithm for finding a retrieve relation

Robinson [22,21] has presented two algorithms for finding retrieve relations for action system refinements. His first algorithm [21] requires the complete abstract and concrete transition systems to be provided as the input. It can be used for both downward and upward simulations. The second algorithm [22] takes a more incremental approach, building the abstract and concrete transition systems as it executes. One of the reasons for developing this second algorithm was to avoid the need to supply the complete transition systems in advance. However, the approach is limited to downward simulation.

In this section, we look at encoding Robinson's first algorithm, adapted to Z, in SAL's model checkers. An investigation of the second algorithm is left for future work.

3.1 The algorithm

The basic strategy of Robinson's first algorithm is to populate a relation R in such a way that it contains all possible retrieve relations, and then to delete mappings until either a retrieve relation is found, or it can be shown that there is no retrieve relation. We adapted the algorithm for downward simulation in Z as follows. Given a concrete specification $C = (CS, CI, \{COp_1, \ldots, COp_n\})$ and an abstract specification $A = (AS, AI, \{AOp_1, \ldots, AOp_n\})$, the steps of the algorithm for downward simulation are:

1. Populate the relation R with all mappings between concrete and abstract states that satisfy the applicability condition. Since all retrieve relations must satisfy this condition, R will contain the mappings of all possible retrieve relations.

$$R = \{c : CS;\ a : AS \mid \forall i : 1 \mathinner{..} n \bullet \\ (\exists a' : AS \bullet a\ AOp_i\ a') \Rightarrow (\exists c' : CS \bullet c\ COp_i\ c') \bullet (c, a)\}$$

2. Check if R satisfies the initialisation condition.

$$\forall c : \mathrm{dom}\ CI \bullet \exists a \in AI \bullet (c, a) \in R$$

If not, there cannot be a retrieve relation. Hence, the algorithm terminates.

3. Attempt to delete a mapping for which correctness does not hold [5].

[5] Again for readability, we have omitted the necessary quantification over the inputs and outputs.

$$\exists\, c : CS, a : AS \bullet (c, a) \in R \wedge$$
$$(\exists\, i : 1 \,..\, n \bullet (\exists\, a' : AS \bullet a\ AOp_i\ a') \wedge$$
$$(\exists\, c' : CS \bullet COp_i \wedge (\nexists a' : AS \bullet AOp_i \wedge (c', a') \in R)))$$
$$R' = R \setminus \{(c, a)\}$$

If this is possible, then return to step 2. If it is not possible, we have a relation R for which applicability, initialisation and correctness hold. Hence, we have a retrieve relation.

It is straightforward to also adapt the algorithm to find upward simulations. In this case, it is necessary to check totality at step 2.

3.2 Encoding the algorithm

The above algorithm can be encoded in SAL's LTL model checker. The basic idea is to initialise a variable R according to step 1 above, and then have a transition which performs deletions according to step 3. A Boolean variable *R_ok* is defined to enable the checks in step 2. Since a precondition of model checking is that the system's transition relation is total, we need to also include a *skip* event (for cases when the delete transition is not enabled). An additional variable *ev* of type $\{del, skip\}$ is used to indicate whether the last event was a deletion or a skip. Given this model, the existence of a retrieve relation is proved by showing that the following LTL formula holds:

$$\mathbf{F}\ (R\_ok \wedge \neg\, \mathbf{X}\ (ev = del))$$

That is, a retrieve relation exists if we eventually reach a state (after any number of deletions) where *R_ok* (and hence the initialisation condition) holds, and we cannot perform any further deletions (and hence all mappings in R satisfy the correctness condition).

The fact that a retrieve relation exists is enough to know that we have a refinement. However, if we wished to know the retrieve relation, we could simply negate the property. Then, if a retrieve relation exists, the model checker will provide us with a counter-example which ends in a state with a retrieve relation. It will, in fact, be the weakest retrieve relation [22].

The SAL encoding is straightforward. We provide types *AS* and *CS* denoting the set of abstract and concrete states respectively. These types are generally tuple types, each element being the type of a state variable. For example, if the abstract state has two variables, one of type X and one of type Y, then in SAL we would have:

$$AS : TYPE = [X, Y]$$

In the case where there is only one state variable of type X, however, a tuple type is not required. AS is simply defined as:

$$AS : TYPE = X$$

Then we can define the type of R as a set of tuples of type $[CS, AS]$. We can also define the initialisation of R, the condition R_ok, and the deletion transition directly as above. A complete encoding for the unique number allocator refinement of Section 2 is shown below.

```
alloc_ltl: CONTEXT =
BEGIN

  MAX: NATURAL = 4;
  NAT: TYPE = [0..MAX];
  INT: TYPE = [-1..MAX];
  CS: TYPE = INT;
  AS: TYPE = set{NAT}!Set;
  EVENT: TYPE = {del,skip};

  main: MODULE =
  BEGIN
    LOCAL R: set{[CS,AS]}!Set
    LOCAL Rok: BOOLEAN
    LOCAL ev: EVENT
    DEFINITION
      Rok = (FORALL (c:CS) : c = -1 =>
               (EXISTS (a:AS) :
                  a = set{NAT}!empty_set AND
                  set{[CS,AS]}!contains?(R,(c,a))));
    INITIALIZATION
      R = {s:[CS,AS]| EXISTS (a:AS,outa:NAT) :
             (NOT(set{NAT}!contains?(s.2,outa)) AND
             a = set{NAT}!union(s.2,{outa}))
             =>
               (EXISTS (c:CS,outc:NAT) :
                 c = s.1+1 AND outc = c)};
    TRANSITION
      [del: (EXISTS (s:[AS,CS]) :
               set{[CS,AS]}!contains?(R,s) AND
               (EXISTS (a:AS,outa:NAT) :
                 NOT(set{NAT}!contains?(s.2,outa)) AND
```

```
              a = set{NAT}!union(s.2,{outa}))
            AND
            (EXISTS (c:CS,outc:NAT) :
              c = s.1+1 AND outc = s.1 AND
              NOT(EXISTS (a1:AS):
                NOT(set{NAT}!contains?(s.2,outc)) AND
                a1 = set{NAT}!union(s.2,{outc}) AND
                set{[CS,AS]}!contains?(R,(c,a1))))
            AND
            R' = set{[CS,AS]}!remove(R,s)
          --> R' IN {s:set{[CS,AS]}!Set|true};
              ev' = del
      []
      skip: true --> ev' = skip]
  END;

  refine: THEOREM main |- F(Rok AND NOT X(ev = del));

END
```

This encoding proves to be quite inefficient, partly due to the extensive use of quantifiers in the guard of the delete transition. Using a 2.16 GHz dual-core CPU PC with 2GB of RAM[6] it is possible to verify the existence of a retrieve relation in 5.5 seconds when MAX is 2. Of this time, only 0.016 seconds is actual verification time, the rest is the preprocessing of the model which includes, among other things, unfolding quantified expressions. When MAX is increased to 3, the check takes 220.781 seconds. The verification time is only 0.437 seconds. When MAX is increased to 4, the check cannot be performed due to the preprocessing step exhausting the available memory.

It is possible to replace the existentially quantified variables s, a, $outa$, c and $outc$ in the delete transition by SAL input variables. Input variables in SAL are nondeterministically assigned a value before each transition. The delete transition is hence able to occur when the chosen values satisfy its guard. In all other cases, the skip transition will occur. With this change to the specification above, the total checking time for $MAX = 2$ is reduced to 0.812 seconds (with 0.125 seconds of this being verification time). However, when MAX is increased to 3, the check takes 907.047 seconds of which 903.188 seconds is verification time. The problem in this case is not the preprocessing step, but the verification exhausting the available memory and requiring

[6] All subsequent checks mentioned in this paper were performed on the same machine.

extensive swapping to proceed. This problem means SAL is again unable to complete the check when MAX is increased to 4.

4 Finding the retrieve relation directly

As shown in our previous work [26], the simulation conditions for data refinement can be encoded in SAL. Specifically, the branching time temporal logic CTL can be used to encode the standard downward and upward simulation conditions under both the blocking and non-blocking interpretations. Given a retrieve relation, these encodings can be used to automatically verify data refinements.

In this section, we provide an overview of the approach for downward simulation, and show how it can be extended to verify data refinements when a retrieve relation is not provided.

4.1 Checking downward simulation with a retrieve relation

Consider the two Z specifications of the unique number allocator of Section 2. To check the refinement with a CTL model checker, we begin by combining the abstract and concrete specifications into one specification by merging their state variables and operations. In general, we assume the variable and operation names of the abstract and concrete specifications are disjoint as in this example (if they were not, they could be made disjoint by a systematic renaming).

We than add an operation *InitA* which sets the variables from the abstract state to satisfy the abstract initial state schema's predicate, and a boolean variable R which is true precisely when the provided retrieve relation holds. Hence, for the example we have:

State

$as : \mathbb{P}\,\mathbb{N}$
$cx : \mathbb{Z}$
$R : \mathbb{B}$

$R \Leftrightarrow (as = \{n : \mathbb{N} \mid n \leqslant cx\})$

InitA

$\Delta State$

$as' = \varnothing$
$cx' = cx$

AOp

$\Delta State$
$out! : \mathbb{N}$

$out! \notin as$
$as' = as \cup \{out!\}$
$cx' = cx$

COp

$\Delta State$
$out! : \mathbb{N}$

$cx' = cx + 1$
$out! = cx'$
$as' = as$

Note that the operations *InitA* and *AOp* do not change the part of the state from the concrete specification, and the operation *COp* does not change the part from the abstract specification. Note also that the specification's initial state is left unconstrained. These features of the specification are necessary to check the simulation conditions as explained below.

This combined system is encoded in SAL in the usual fashion with a *skip* transition to ensure freedom from deadlock. We also add a variable *ev* of type $\{inita, aop, cop, skip\}$ to record the event (i.e., transition) which last occurred. The restriction on R in the state schema can be encoded in a SAL definition clause.

The *InitA* operation is introduced in order to check the initialisation simulation condition: whether an abstract initial state related to a given concrete initial state exists. The appropriate initialisation check is the following CTL property.

$$cx = -1 \Rightarrow \mathbf{EX}\ (ev = inita \wedge R)$$

That is, whenever the concrete part of the initial state satisfies the concrete state schema's predicate, *InitA* (which does not change the concrete part of the state) can occur such that R holds.

To check applicability we use the transitions corresponding to the abstract and concrete operations. The applicability condition holds if whenever an abstract operation can be performed, the corresponding concrete operation can be performed from any related concrete state. Thus the CTL check is as follows.

$$R \Rightarrow (\mathbf{EX}\ (ev = aop) \Rightarrow \mathbf{EX}\ (ev = cop))$$

Finally for correctness, it is required that whenever an abstract operation can occur, then if the corresponding concrete operation can occur from a related concrete state, any state reached by performing the concrete operation is related to an abstract state that can be reached by performing the abstract operation.

We capture this in CTL by using the transitions corresponding to the

abstract and concrete operations. Those corresponding to the abstract operations do not change the concrete variables. Similarly, those corresponding to the concrete operations do not change the abstract variables. This allows us to perform the transitions corresponding to a concrete operation COp and abstract operation AOp 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, and the concrete part is identical to that which could have been reached by performing only COp. The check for the correctness condition is then as follows (where we have added in the necessary quantification for the outputs).

$$\begin{array}{l} R \Rightarrow (\mathbf{EX}\ (ev = aop) \Rightarrow \\ \quad (\forall\, n : \mathbb{N} \bullet \mathbf{AX}\ (ev = cop \wedge out = n \Rightarrow \\ \qquad\qquad \mathbf{EX}\ (ev = aop \wedge out = n \wedge R)))) \end{array}$$

The full CTL property to check downward simulation is the conjunction of the above properties.

$$\begin{array}{l} cx = -1 \Rightarrow \mathbf{EX}\ (ev = inita \wedge R) \wedge \\ R \Rightarrow (\mathbf{EX}\ (ev = aop) \Rightarrow \mathbf{EX}\ (ev = cop)) \wedge \\ R \Rightarrow (\mathbf{EX}\ (ev = aop) \Rightarrow \\ \quad (\forall\, n : \mathbb{N} \bullet \mathbf{AX}\ (ev = cop \wedge out = n \Rightarrow \\ \qquad\qquad \mathbf{EX}\ (ev = aop \wedge out = n \wedge R)))) \end{array}$$

A more involved explanation of the approach together with a larger case study can be found in [26]. The approach is relatively efficient. For the allocator example, we can perform the check for a maximum natural number of 23 before memory limitations prevent the check proceeding.

4.2 Checking downward simulation without a retrieve relation

In the approach above the CTL property must hold on all initial states of the system, i.e, all possible pairs of abstract and concrete states. It relies on the fact that the same retrieve relation (that captured by the state variable R) is used when checking the property for each such pair of states.

Rather than providing a retrieve relation as part of the encoding, we can get the model checker to find a retrieve relation which satisfies the CTL property for all pairs of states. This is done by changing the type of variable R to be a set of pairs of concrete and abstract states and adding two new transitions to our system. The first *chooseR* nondeterministically chooses a value for R. The second *chooseState* nondeterministically chooses a state. We then need to prove the existence of a path where after performing *chooseR*, we can prove

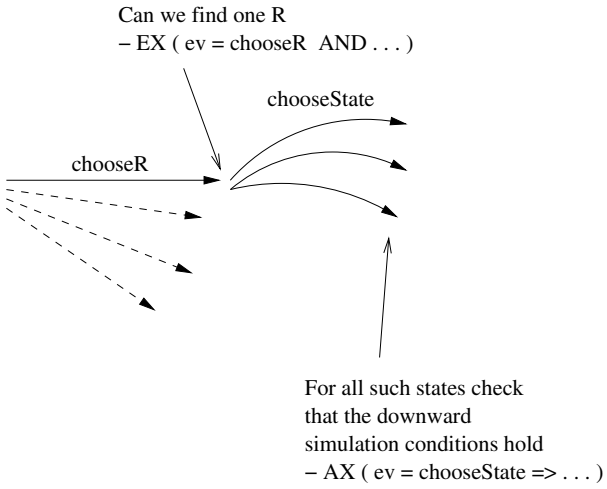

Fig. 1. Conditions required for downward simulation

the CTL conditions above for all states resulting from *chooseState*. This is represented schematically in Figure 1.

Thus in our example SAL encoding described above, we add types AS and CS corresponding to the abstract and concrete state spaces allowing us to declare R as a set of pairs of concrete and abstract states (as in Section 3.2). Since *chooseR* is always enabled, we remove the *skip* transition. We also change the type of event to be $\{inita, aop, cop, chooseR, chooseState\}$ in order that we may redefine the required CTL property as follows (where (c, a) represents the pair of concrete and abstract states, e.g., (cx, as) in the example).

$$\begin{aligned}
&\mathbf{EX}\ (ev = chooseR \wedge \mathbf{AX}\ (ev = chooseState \Rightarrow \\
&\quad cx = -1 \Rightarrow \mathbf{EX}\ (ev = ainit \wedge (c, a) \in R) \wedge \\
&\quad (c, a) \in R \Rightarrow (\mathbf{EX}\ (ev = aop) \Rightarrow \mathbf{EX}\ (ev = cop)) \wedge \\
&\quad (c, a) \in R \Rightarrow (\mathbf{EX}\ (ev = aop) \Rightarrow \\
&\qquad (\forall n : NAT \bullet \mathbf{AX}\ (ev = cop \wedge out = n \Rightarrow \\
&\qquad\qquad \mathbf{EX}\ (ev = aop \wedge out = n \wedge (c, a) \in R))))))
\end{aligned}$$

A complete encoding for the unique number allocator refinement of Section 2 is shown below.

```
alloc_chooseR: CONTEXT =
BEGIN

  MAX: NATURAL = 4;
  NAT: TYPE = [0..MAX];
  INT: TYPE = [-1..MAX];
  EVENT: TYPE = {ainit, aop, cop, chooseR, chooseState};
```

```
AS: TYPE = set{NAT}!Set;
CS: TYPE = INT;

main: MODULE =
BEGIN
  LOCAL as: set{NAT}!Set
  LOCAL cx: NAT
  OUTPUT out: NAT
  LOCAL R: set{[CS,AS]}!Set
  LOCAL ev: EVENT
  DEFINITION
    R = (as = {n:NAT|n <= cx})
  TRANSITION
    [chooseR: true
              --> R' IN {s:set{[CS,AS]}!Set|true};
                  ev' = chooseR
     []
     chooseState: true
                  --> as' IN {s:set{NAT}!Set|true};
                      cx' IN {n:INT|true};
                      ev' = chooseState
     inita: true
            --> as' = set{NAT}!empty_set;
                ev' = inita
     []
     aop: NOT(set{NAT}!contains?(as,aout'))
          --> as' = set{NAT}!union(as,{aout'})
              out' IN {n:NAT|true};
              ev' = aop
     []
     cop: true
          --> cx' = cx + 1;
              out' = cx';
              ev' = cop
    ]
  END;

refine: THEOREM main |-  EX(chooseR AND AX(chooseState =>
          (cx = -1 => EX(ev = inita AND
                         set{[CS,AS]}!contains?(R,(cx,as))))
          AND
```

```
(set{[CS,AS]}!contains?(R,(cx,as)) =>
  (EX(ev = aop) => EX(ev = cop)))
AND
(set{[CS,AS]}!contains?(R,(cx,as)) =>
  (EX(ev = aop) =>
    (FORALL (n:NAT) : AX((ev = cop AND out = n) =>
      EX(ev = aop AND out = n AND
        set{[CS,AS]}!contains?(R,(cx,as))))))));

END
```

With this approach, when $MAX = 2$ the checking time is comparable with those based on Robinson's algorithm: the total time is 1.485 seconds and the verification time is 0.047 seconds. However, the approach performs much better than the other approaches when MAX is increased to 3. The total time then is 4.22 seconds with a verification time of 0.062 seconds. Furthermore, we can increase MAX to 4 without exhausting the available memory: the check can be performed in 18.812 seconds, 0.188 seconds of which is verification time.

For values of MAX above 4, the available memory is exhausted in the pre-processing stage due, in this case, to the complexity of the CTL property. The time efficiency of this approach with respect to the encodings of Robinson's algorithm is summarised below (all times are in seconds).

	MAX	verification time	total time
algorithm	2	0.047	5.5
(initial encoding)	3	0.437	220.781
	4	memory exhausted	
algorithm	2	0.125	0.812
(using inputs)	3	903.188	907.047
	4	memory exhausted	
direct approach	2	0.047	1.485
	3	0.062	4.22
	4	0.188	18.182
	5	memory exhausted	

5 Discussion

In this paper, we have shown that temporal logic model checking can be used to automatically find retrieve relations between abstract and concrete state-based specifications. The existence of such a retrieve relation implies the concrete specification is a data refinement of the abstract one. Hence, the approach enables fully automatic checking of data refinements.

While we have shown this is feasible, we have not yet provided a practical method. The size of the state spaces, and hence specifications, we can handle is quite limited. This is mostly due to the fact that we need to include the retrieve relation (a set of abstract and concrete state pairs) as part of the state-space of our model. In our previous work where we provided, rather than calculated, the retrieve relation [26], it could be represented by a boolean variable, rather than a set, resulting in much greater efficiency.

The efficiency of the approaches in this paper could be improved using the optimisation features supported by the model checkers we have used. We have not investigated this possibility, using only the basic functionality of the tools. However, it seems likely that this will not solve the problem for any but the smallest of specifications.

A more promising way forward would be to use a technique which allows us to reduce the size of our models while maintaining the verity of the properties we wish to prove. Such techniques include data abstraction [9], predicate abstraction [5,28] and data independence [24]. For example, data independence is based on the fact that sometimes properties we wish to prove are independent of the values associated with particular types. By these means it is possible to show that the refinement between the abstract and concrete unique number allocator specifications in this paper are independent of the maximum number allocated. Hence, verifying the refinement when this value is 2 infers the refinement for any other value. Exploration of this is left as future work.

Acknowledgement

John Derrick was supported by the Leverhulme Trust via a Research Fellowship for this work. Graeme Smith acknowledges the support of Australian Research Council (ARC) Discovery Grant, DP0558408.

References

[1] Abrial, J.-R., "The B-Book: Assigning Programs to Meanings," Cambridge University Press, 1996.

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

[3] Bolton, C., *Using the Alloy Analyzer to verify data refinement in Z*, Electronic Notes in Theoretical Computer Science **137** (2005), pp. 23–44.

[4] Butler, M., *On the use of data refinement in the development of secure communications systems*, Formal Aspects of Computing **14** (2002), pp. 2–34.

[5] Clarke, E., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Counterexample-guided abstraction refinement*, in: E. Emerson and A. Sistla, editors, *International Conference on Computer Aided Verification (CAV'00)*, LNCS **1855** (2000), pp. 154–169.

[6] de Moura, L., 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)*, LNCS **3114** (2004), pp. 496–500.

[7] de Roever, W.-P. and K. Engelhardt, "Data Refinement: Model-Oriented Proof Methods and their Comparison," Cambridge University Press, 1998.

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

[9] Derrick, J. and H. Wehrheim, *On using data abstractions for model checking refinements*, Acta Informatica **44** (2007), pp. 41–71.

[10] Doche, M. and A. Gravell, *Extraction of abstraction invariants for data refinement*, in: D. Bert, J. Bowen, M. Henson and K. Robinson, editors, *International Conference of Z and B Users (ZB 2002)*, LNCS **2272** (2002), pp. 102–139.

[11] Emerson, E., *Temporal and modal logic*, in: J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (volume B)*, Elsevier Science Publishers, 1990 pp. 996–1072.

[12] Fischer, C. 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)* (1999), pp. 315–334.

[13] He, J., *Process refinement*, in: J. McDermid, editor, *The Theory and Practice of Refinement* (1989).

[14] Jackson, D., *Alloy: a lightweight object modelling notation*, ACM Transactions on Software Engineering and Methodology **11** (2002), pp. 256–290.

[15] Josephs, M., *A state-based approach to communicating processes*, Distributed Computing **3** (1988), pp. 9–18.

[16] Kassel, G. and G. Smith, *Model checking Object-Z classes: Some experiments with FDR*, in: *Asia-Pacific Software Engineering Conference (APSEC 2001)* (2001).

[17] Leuschel, M. and M. Butler, *ProB: A model checker for B*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *Formal Methods Europe (FME 2003)*, LNCS **2805** (2003), pp. 855–874.

[18] Leuschel, M. and M. Butler, *Automatic refinement checking for B*, in: K. Lau and R. Banach, editors, *International Conference on Formal Engineering Methods (ICFEM 2005)*, LNCS **3785** (2005), pp. 345–359.

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

[20] Robinson, N., *Checking Z data refinement using an animation tool*, in: D. Bert, J. Bowen, M. Henson and K. Robinson, editors, *International Conference of Z and B Users (ZB 2002)*, LNCS **2272** (2002), pp. 62–81.

[21] Robinson, N., *Finding abstraction relations for data refinement*, Technical Report TR03-03, Software Verification Research Centre, The University of Queensland (2003).

[22] Robinson, N., *Incremental derivation of abstraction relations for data refinement*, in: J. Dong and J. Woodcock, editors, *International Conference on Formal Engineering Methods (ICFEM 2003)*, LNCS **2885** (2003), pp. 246–265.

[23] Robinson, N. and C. Fidge, *Animation of data refinements*, in: P. Strooper and P. Muenchaisri, editors, *Asia-Pacific Software Engineering Conference (APSEC 2002)* (2002), pp. 137–146.

[24] Roscoe, A., "The Theory and Practice of Concurrency," Series in Computer Science, Prentice Hall, 1998.

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

[26] Smith, G. and J. Derrick, *Verifying data refinements using a model checker*, Formal Aspects of Computing **18** (2006), pp. 264–287.

[27] Smith, G. 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)*, LNCS **3455** (2005), pp. 87–105.

[28] Smith, G. and K. Winter, *Proving temporal properties of Z specifications using abstraction*, in: D. Bert, J. Bowen, S. King and M. Waldén, editors, *International Conference of Z and B Users (ZB 2003)*, LNCS **2651** (2003), pp. 260–279.

[29] Spivey, J., "The Z Notation: A Reference Manual," Prentice Hall, 1992, 2nd edition.