Proving temporal properties of Z specifications using abstraction

Graeme Smith and Kirsten Winter

Software Verification Research Centre
University of Queensland 4072, Australia
smith@svrc.uq.edu.au kirsten@svrc.uq.edu.au

Abstract. This paper presents a systematic approach to proving temporal properties of arbitrary Z specifications. The approach involves (i) transforming the Z specification to an abstract *temporal structure* (or state transition system), (ii) applying a model checker to the temporal structure, (iii) determining whether the temporal structure is too abstract based on the model checking result and (iv) refining the temporal structure where necessary. The approach is based on existing work from the model checking literature, adapting it to Z.

1 Introduction

Specifications in Z [Spi92], and related languages such as Object-Z [Smi00], often involve predicates of arbitrary complexity and have infinite state spaces. Consequently, tool support for proving properties of such specifications has focussed on theorem proving [KSW96,Saa97,TM95], rather than automated techniques such as model checking [CGP00] which are limited with respect to the notation supported and the size of the state space of the specification.

To extend the limits of model checking, much research in the past decade has focussed on *abstraction* as a means of state space reduction [CGL94,LGS+95]. A system model with a large, or infinite, state space is transformed to one with a reduced state space suitable for model checking. This is done based on the notion of *abstract interpretation* (originally developed to derive abstract semantics of programming languages [CC79]). In essence, abstraction is the inverse of downward simulation data refinement [DB01]. Hence, any properties which are preserved by such refinement and can be proved true for the abstract model are also true for the concrete model. Properties preserved by downward simulation are of the form that something is true on all abstract behaviours. A property which states something is true on one, or a limited number, of abstract behaviours is not, in general, preserved.

For abstraction to be practically useful, two further issues need to be addressed. Firstly, the derivation of the abstract model needs to be at least systematic, and at best automatic. Otherwise, we are left with a significant intellectual task and lose the primary benefit of model checking. Secondly, we need to deal

with the case where a property of interest is proved false for the abstract model. In this case, nothing can be deduced about the concrete model which (like a refinement) may have additional properties due to a decrease in nondeterminism. These issues have been addressed by the model checking community (e.g., [GS97,SS99,CGJ+00]) and, in this paper, we draw on these results to provide a practical approach to abstraction for Z.

Our goal is to present a systematic approach to abstraction of Z specifications that could be supported by a theorem prover. The approach is not specific to any particular theorem prover, nor is the abstract model produced aimed at any particular model checker. The approach could be used with any suitable combination of such tools. We begin in Section 2 with a summary of the relevant results from the model checking community. In Section 3, we present an approach to abstraction for Z specifications and illustrate it by an example in Section 4. In Section 5, we show, via the example, how to deal with properties that are proved false for the abstract system. Future directions are discussed in Section 6.

2 Background

Abstraction as a means of state space reduction has been a topic of research in the model checking community for some time. Early work provided a theoretical framework for abstraction. For example, Clarke, Grumberg and Long [CGL94] provide a method for transforming finite state programs to an abstract transition system. They prove that the properties of the abstract model expressed in a restricted temporal logic which only allows properties that state that something is true on all abstract behaviours (namely the universal fragment of CTL*, $\forall$CTL*) are also properties of the program, and hence that the abstract model can be used to verify the program. A similar approach is described by Loiseaux et al. [LGS+95] for a slightly more expressive temporal logic (a restricted version of the μ-calculus).

The general idea is to group states in the concrete model into equivalence classes and map these via an abstraction function, $\boldsymbol{Abs}$, to states of the abstract model (see Figure 1).

2.1 Finding the abstraction function

The abstraction function should be chosen in such a way that the property of interest can easily be proved in the abstract model. A number of papers, including that of Clarke, Grumberg and Long [CGL94], suggest guidance for finding such functions (e.g., [Jac94,WVF97]). For example, Jackson [Jac94] in his approach for abstracting Z specifications, suggests using the properties to be proved as a basis for defining a suitable abstraction function. Assume we want to prove that a variable $y : \mathbb{N}$ is less than 10. Of course, we could split the state space into the equivalence classes $y < 0$, $y = 0$ and $y > 0$. However, for this property it is more useful to use the equivalence classes $y < 10$, $y = 10$ and $y > 10$.

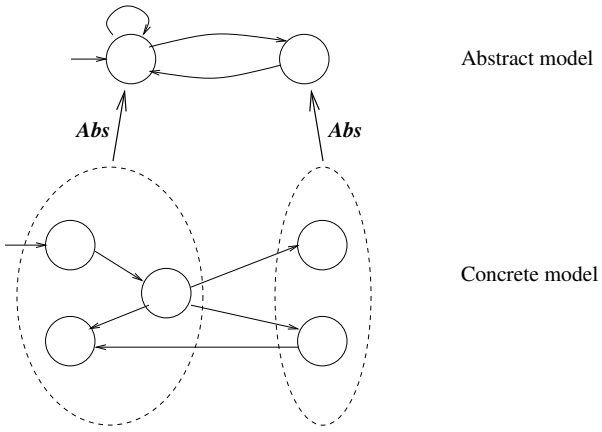


Fig. 1. Abstraction using equivalence classes of states

However, none of the approaches mentioned above provide a systematic way of defining an abstraction function. This problem is overcome by Graf and Saïdi [GS97]. They present an automated approach to abstraction which does not require an abstraction function to be defined in advance. Instead, it requires a set of predicates on which to base the abstraction process.

From these predicates, Graf and Saïdi form a set of *monomials*. A monomial is a conjunction of, for each predicate, either the predicate or its negation. For example, given the predicates p_1 and p_2, the monomials are $p_1 \wedge p_2$, $p_1 \wedge \neg\, p_2$, $\neg\, p_1 \wedge p_2$ and $\neg\, p_1 \wedge \neg\, p_2$. Since the monomials partition the complete concrete state space, they can be used as the equivalence classes of the concrete states.

Choosing the predicates can be based on the user's understanding of the system. However, Graf and Saïdi offer general guidance for choosing them.

- Using predicates which appear in the property to be proved results in an abstract model where states either satisfy these predicates or not.
- Using the predicates in the guards of transitions simplifies the abstraction process. Since the information of enabledness of transitions is encoded in the abstract state space the possible transitions of the abstract model are easily determined.
- In general, *atomic predicates*, e.g., $x = 2$ and $y = 3$ rather than $x = 2 \Rightarrow y = 3$, result in a better level of abstraction.

The second point listed above, using the predicates in the guards, is essential. Otherwise, properties proved for the abstract model may not be true for the concrete model, i.e., we do not have a proper abstraction. For example, in Figure 2 since the equivalence class containing concrete states s_1 and s_2 does not reflect the guard of the transition, we are able to prove that the abstract model always progresses to state t_2. The corresponding property on the concrete model, that it always progresses to state s_3, is however not true.

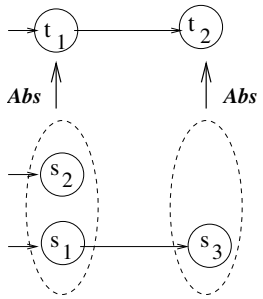


Fig. 2. Equivalence classes not reflecting transition guards

2.2 False counter-example detection and refinement

When we model-check an abstract model, the model checker might successfully prove the property of interest. In this case, we know the property holds for our concrete model as well. However, if the model checker disproves the property by producing a counter-example, we cannot determine anything about the concrete model. The model checker has effectively proved that something is not true on at least one abstract behaviour. Such properties on one, or a limited number of behaviours, are not necessarily true of our concrete model. Hence, the counter-example may also be a counter-example of the concrete model, or it may only be a counter-example of the less deterministic abstract model. In the latter case, we call it a *false counter-example*.

The counter-examples produced by most existing model checkers are a sequence of states starting from an initial state. These sequences are either finite or, if infinite, involve a loop back to a previous state (see Figure 3).

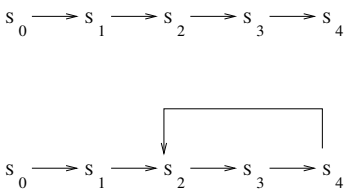


Fig. 3. Finite and infinite (looping) counter-examples

If we get a false counter-example, the abstract model needs to be refined closer to the concrete model. Clarke et al. [CGJ+00] introduce algorithms for automatically detecting false counter-examples of the kinds in Figure 3. This is done based on the fact that a false counter-example contains an abstract state whose corresponding concrete state is not reachable in the concrete model.

Given a false counter-example, they also automatically derive a refinement of the abstract model in which the false counter-example is avoided. This is done by finding the last abstract state in the false counter-example whose corresponding equivalence class of concrete states is reachable in the concrete model. This equivalence class is split based on whether or not a state is reachable via the counter-example (see Figure 4 in which s_4 and s_5 are separated from s_6).

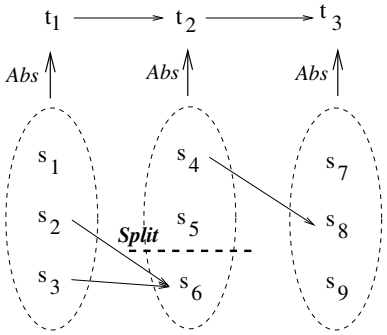


Fig. 4. Splitting of equivalence classes

For deriving the abstract model, Clarke et al. follow the approach of Graf and Saïdi [GS97] of using a set of predicates. However, rather than using them to generate an abstract model, they use them to generate an abstraction function which is then used to compute the abstract model.

3 Abstraction of Z specifications

An operation in Z does not have a guard, but a precondition outside of which the operation can occur changing the state arbitrarily. Such arbitrary state changes make it impossible to prove most interesting temporal properties (over all behaviours). We restrict our approach, therefore, to Z specifications where operations are *totalised*. That is, for each possible pre-state, the operation explicitly specifies a post-state which in some cases may be an error state.

The precondition of such a totalised operation is equivalent to true. Hence, whether the operation is interpreted as having a precondition (in the Z sense) or guard is irrelevant (the operation can occur from any pre-state). In this work, we choose to treat them as having guards. This is done to extend the applicability of our approach to Object-Z (where operations are guarded [Smi00]) as well as non-standard (behavioural) interpretations of Z with guarded operations [DB01].

We follow the approach of using a set of predicates to derive an abstract Z specification. Since all operation guards are true, we use only the atomic predicates in the property to be proved. (For Object-Z or guarded interpretations

of Z, we would also need to use the atomic predicates in the operation guards to avoid the problem illustrated in Figure 2.) Using the monomials of these atomic predicates, we form equivalence classes of the state space of the concrete model. An explicit abstraction function then relates these equivalence classes to single abstract states. This abstraction function is used to derive an abstract model. The soundness of our process will be proven with respect to Z downward simulation laws [DB01].

For expressing system properties, we adopt Linear Temporal Logic (LTL) [Eme90], the temporal operators of which are explained below.

$\mathbf{G}\ \phi$ Now and at all points in the future, it is the case that ϕ is true.
$\mathbf{F}\ \phi$ At some point in the future, it will be the case that ϕ is true.
$\mathbf{X}\ \phi$ In the next state, it will be the case that ϕ is true.
$\phi\ \mathbf{U}\ \psi$ At some point in the future, it will be the case that ψ is true, and from now until then it is always the case that ϕ is true.

An LTL property holds for a specification, if it holds for all *paths*, i.e., sequences of states, that the specified system can undergo.

3.1 The abstraction process

Properties expressed in LTL are preserved by downward simulation data refinement [DB01]. This is because the properties have to hold for all paths that the specified system can undergo. Since downward simulation only eliminates paths (by reducing nondeterminism) and does not allow new ones to be added, properties which are true on all paths of an abstract specification, will be true on all paths of a refinement of that specification.

Hence, a suitable abstraction will be one that can be refined to the original specification using downward simulation. Under a guarded interpretation of operations, a Z specification with state schema $CState$, initial state schema $CInit$ and operations $COp_1 \ldots COp_n$ is a downward simulation of an abstract specification with state schema $AState$, initial state schema $AInit$ and operations $AOp_1 \ldots AOp_n$, if there is a retrieve relation $Retr$ such that the following conditions hold [DB01].

$$\begin{array}{ll}
\textbf{Initialisation} & \forall\, CInit \bullet \exists\, AInit \bullet Retr \\
\textbf{Applicability} & \forall\, AState;\ CState \bullet \\
& \qquad Retr \Rightarrow (\mathrm{pre}\, AOp_i \Leftrightarrow \mathrm{pre}\, COp_i) \quad \text{for } i \in 1..n \\
\textbf{Correctness} & \forall\, AState;\ CState;\ CState' \bullet \\
& \qquad Retr \wedge COp_i \Rightarrow (\exists\, AState' \bullet Retr' \wedge AOp_i) \quad \text{for } i \in 1..n
\end{array}$$

These conditions are used to show the soundness of our abstraction process below.

Assume the monomials, built from the atomic predicates of the property to be proved, are $p_1, \ldots, p_n$. Then the state schema of the abstract specification is:

AState

$s : State$

where $State ::= s_1 \mid \ldots \mid s_n$, and the abstraction function is defined by a schema:

Abs

$AState$

$CState$

$p_1 \Rightarrow s = s_1$

$\ldots$

$p_n \Rightarrow s = s_n$

Informally, the abstraction function is a mapping $\{p_1 \mapsto s_1, \ldots, p_n \mapsto s_n\}$.

The initial state schema and operations of the abstract system are derived such that the original specification refines the abstract specification under a retrieve relation which is the inverse of the abstraction function (and hence defined by the same schema, i.e., $Retr = Abs$).

The initialisation condition for downward simulation requires that there is an abstract initial state related to each concrete initial state. Since the monomials partition the concrete state space, they represent all concrete states. Therefore, there will be an abstract state corresponding to each concrete initial state. To ensure that this abstract state is also an abstract initial state, we define the abstract initial state schema as:

AInit

$AState$

$\exists\, CInit \bullet Abs$

With this definition of $AInit$, the initialisation condition becomes:

$$
\begin{aligned}
& \forall\, CInit \bullet \exists\, AState \mid (\exists\, CInit \bullet Abs) \bullet Abs \\
\equiv\ & \forall\, CInit \bullet \exists\, AState \bullet (\exists\, CInit \bullet Abs) \land Abs \\
\Leftarrow\ & \forall\, CInit \bullet \exists\, AState;\ CInit \bullet Abs \land Abs \\
\equiv\ & \forall\, CInit \bullet \exists\, AState;\ CInit \bullet Abs
\end{aligned}
$$

If there are no initial concrete states then the above is immediately true. If there is a concrete initial state then there is an abstract state related to it due to the abstraction function being based on monomials (and hence $\exists\, AState;\ CInit \bullet Abs$ is true).

For each operation, we require that the applicability and correctness conditions hold. This will be true when an abstract operation starts and ends only in states which are related to those equivalence classes containing concrete states

that serve as start and end points of the corresponding concrete operation[1]. Hence, an abstract operation AOp_i is defined as:

$$
\begin{array}{|l}
\hline
AOp_i \\
\Delta AState \\
\hline
\exists\, CState;\ CState' \bullet COp_i \land Abs \land Abs' \\
\hline
\end{array}
$$

The applicability condition becomes:

$$
\begin{array}{ll}
& \forall\, AState;\ CState \bullet Abs \Rightarrow \\
& \quad ((\exists\, AState' \bullet \exists\, CState;\ CState' \bullet COp_i \land Abs \land Abs') \Leftrightarrow \text{pre}\, COp_i) \\
\equiv & \forall\, AState;\ CState \bullet Abs \Rightarrow \\
& \quad ((\exists\, CState;\ CState' \bullet COp_i \land Abs \land (\exists\, AState' \bullet Abs')) \Leftrightarrow \text{pre}\, COp_i)
\end{array}
$$

Since there is an abstract state related to each concrete state due to the abstraction function being based on monomials, the above is equivalent to:

$$
\begin{array}{ll}
& \forall\, AState;\ CState \bullet Abs \Rightarrow \\
& \quad ((\exists\, CState;\ CState' \bullet COp_i \land Abs) \Leftrightarrow \text{pre}\, COp_i) \\
\equiv & \forall\, AState;\ CState \bullet Abs \Rightarrow \\
& \quad ((\exists\, CState \bullet Abs \land (\exists\, CState' \bullet COp_i)) \Leftrightarrow \text{pre}\, COp_i) \\
\equiv & \forall\, AState;\ CState \bullet Abs \Rightarrow \\
& \quad ((\exists\, CState \bullet Abs \land \text{pre}\, COp_i) \Leftrightarrow \text{pre}\, COp_i)
\end{array}
$$

Since $\text{pre}\, COp_i$ is true under our restriction that operations be totalised, the above is trivially true. (For Object-Z or guarded interpretations of Z, $\text{pre}\, COp_i$ would not necessarily be true. In this case, since the monomials defining the abstraction function would be based on atomic predicates in the operation guards, the same operations would be enabled from any two concrete states related to the same abstract state. Hence, the above would be true.)

The correctness condition becomes:

$$
\begin{array}{ll}
& \forall\, AState;\ CState;\ CState' \bullet Abs \land COp_i \Rightarrow \\
& \quad (\exists\, AState' \bullet Abs' \land (\exists\, CState;\ CState' \bullet COp_i \land Abs \land Abs')) \\
\Leftarrow & \forall\, AState;\ CState;\ CState' \bullet Abs \land COp_i \Rightarrow \\
& \quad (\exists\, AState' \bullet Abs' \land COp_i \land Abs \land Abs') \\
\equiv & \forall\, AState;\ CState;\ CState' \bullet Abs \land COp_i \Rightarrow \\
& \quad Abs \land COp_i \land (\exists\, AState' \bullet Abs' \land Abs') \\
\equiv & \forall\, AState;\ CState;\ CState' \bullet Abs \land COp_i \Rightarrow (\exists\, AState' \bullet Abs') \\
\Leftarrow & \forall\, CState' \bullet (\exists\, AState' \bullet Abs')
\end{array}
$$

Since there is an abstract state related to each concrete state due to the abstraction function being based on monomials, the above is true.

[1] However, not all of the concrete states in the equivalence class need to satisfy this condition.

3.2 Handling inputs and outputs

The approach as presented so far only works for specifications without inputs and outputs. We could have accounted for inputs and outputs by basing our abstraction function on a refinement definition which included them [DB01]. Instead, we embed them in the specification state. Since LTL properties only involve state variables, this embedding allows us to prove properties about inputs and outputs in our approach.

The embedding is done in such a way that no new properties are introduced on the existing state variables. Each input and output variable appears as a distinct state variable whose name is the same as that of the original variable[2] and whose type is the declared type of the input or output variable extended with an undefined element $\bot$.

Initially, all outputs are equal to $\bot$ and the values of inputs are unconstrained. For each operation, if an input or output is declared by the operation (before the embedding) then no additional constraints are placed on it. In the case of an output, the originally declared variable is renamed to a post-state (primed) variable. If an input or output is not declared by an operation, its value in the pre-state, in the case of inputs, and post-state, in the case of outputs, is $\bot$. This will be illustrated in the example in the next section.

The new specification after the embedding has no new paths modulo the embedded variables. Nor does it have fewer paths than the original specification. Although inputs are set to particular (unspecified) values initially and after each operation, post-states exist for all possible values allowing the specification to proceed as before. The unspecified nature of the inputs models the fact that it is the environment of the specified system that is choosing them. Hence, it only makes sense to use inputs in assumptions of a property we wish to prove, e.g., on the left-hand side of an implication.

4 Unique Number Allocator Example

To illustrate our approach, we introduce a simple example of an infinite state system. The system is a unique number allocator which accepts requests for a strictly positive number and sends them to the requester. It is specified as having two variables: *used* denoting the numbers that it has already allocated, and *alloc* denoting the number, if any, it has allocated but not yet sent.

Allocator

$$used : \mathbb{P}\,\mathbb{N}_1$$
$$alloc : \mathbb{F}\,\mathbb{N}_1$$

$$\#alloc \leqslant 1$$

Initially, no numbers have been allocated.

[2] The original specification must not use the same name for inputs or outputs in different operations unless they have the same type.

Init

$$used = \varnothing$$
$$alloc = \varnothing$$

The operation *Request* specifies that whenever *alloc* is empty and $used \neq \mathbb{N}_1$, a request for a new number can be made. A new number (not previously allocated) is placed in *alloc* and added to *used*. When *alloc* is not empty or $used = \mathbb{N}_1$, the operation leaves the state unchanged.

Request

$$\Delta Allocator$$

$$alloc = \varnothing \land used \neq \mathbb{N}_1 \Rightarrow$$
$$(\exists\, n : \mathbb{N}_1 \bullet n \notin used \land alloc' = \{n\} \land used' = used \cup \{n\})$$
$$alloc \neq \varnothing \lor used = \mathbb{N}_1 \Rightarrow alloc' = alloc \land used' = used$$

The operation *Send* specifies that whenever *alloc* is not empty, its element may be sent (and removed from *alloc*). When *alloc* is empty the operation outputs the value zero to indicate an error and leaves the state unchanged.

Send

$$\Delta Allocator$$
$$n! : \mathbb{N}$$

$$alloc = \{n!\} \Rightarrow alloc' = \varnothing \land used' = used$$
$$alloc = \varnothing \Rightarrow n! = 0 \land alloc' = alloc \land used' = used$$

We embed the inputs and outputs, in this case just the output $n! : \mathbb{N}$, in the specification as follows.

$Allocator_{IO}$

$$Allocator$$
$$n! : \mathbb{N}^{\perp}$$

$Init_{IO}$

$$Init$$

$$n! = \perp$$

$Request_{IO}$

$$Request$$

$$n!' = \perp$$

$Send_{IO}$

$$Send[n!'/n!]$$

where $\mathbb{N}^{\perp}$ is the set of natural numbers extended with the undefined value $\perp$. Note that we only restrict the value of $n!$ initially and *after* operations. Hence, $n!$ is renamed to $n!'$ in the operation *Send* above.

One property that we want for the unique number allocator is that we never send a given value $v : \mathbb{N}_1$ twice. This can be expressed in LTL as follows.

$$\mathbf{G}\ (n! \neq v \lor \mathbf{X}\ (\mathbf{G}\ n! \neq v))$$

That is, it is always the case that either $n! \neq v$ or (if $n! = v$) in the next state, it will always be the case that $n! \neq v$.

The property only has one atomic predicate $n! \neq v$ and hence the set of monomials is $\{n! \neq v, n! = v\}$. Following the process in Section 3 gives the abstract state schema:

AState

$s : State$

where $State ::= s_1 \mid s_2$ and abstraction function:

Abs

$AState$

$Allocator_{IO}$

$n! \neq v \Rightarrow s = s_1$

$n! = v \Rightarrow s = s_2$

The initial state schema is:

AInit

$AState$

$\exists\, used, alloc : \mathbb{P}\,\mathbb{N}_1;\ n! : \mathbb{N}^{\perp} \mid \#alloc \leqslant 1 \land n! = \perp \bullet$

$n! \neq v \Rightarrow s = s_1 \land$

$n! = v \Rightarrow s = s_2$

which simplifies to:

AInit

$AState$

$s = s_1$

The abstract operations similarly simplify to:

ARequest

$\Delta AState$

$s' = s_1$

ASend

$\Delta AState$

---

true

Finally, the property is also abstracted to:

$\mathbf{G}\ (s = s_1 \lor \mathbf{X}\ (\mathbf{G}\ s = s_1))$

The entire abstraction process is systematic and potentially automatable. The simplifications of the abstract schemas could be done using a theorem prover either automatically, or in the case of more complicated specifications, with

some user guidance. The conversion of the abstract specification into the input format of a model checker is also potentially automatable since its state schema comprises a single variable with a finite set of values (representing possible states) and its operations define transitions between these values. In other words, the abstract specification defines a simple *temporal structure* (or state transition system) as shown in Figure 5.

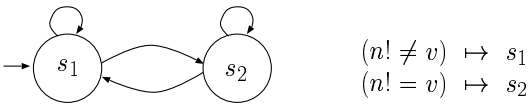


Fig. 5. Temporal structure of the abstract specification of the unique number allocator

Hence, we should be able to model check the abstract specification to see if our desired property holds. What we will find, and what will become evident from Figure 1, is that our abstract specification is too abstract to prove this property. Hence, the model checker will return a counter-example for the abstract system which is not a counter-example for our original specification. Dealing with such counter-examples is the topic of the next section.

5 False counter-example detection and refinement

As we are targeting the use of standard model checkers, we assume counter-examples returned by model checking are either a finite or infinite (looping) sequence of states as shown in Figure 3. To determine whether or not they are false counter-examples, we follow an approach that is inspired by the work of Clarke et al. [CGJ+00]. We show here the process to follow for finite sequences of states. Clarke et al. argue that the process for infinite (looping) sequences of states is a minor variation on that for finite sequences.

5.1 False counter-example detection

Assume our counter-example is a sequence of states $\langle t_0, \ldots, t_m \rangle$ where $t_0, \ldots, t_m : \mathit{State}$. We need to find a sequence of operations in the original (concrete) specification that passes through concrete states related to the abstract states in the counter-example. If such a sequence of operations cannot be found we can conclude that the corresponding states are not reachable in the concrete model and thus, the counter-example is a false counter-example.

The initial concrete states related to the abstract state t_0 are given by the schema C_0.

C_0

$CInit$

$\exists\, s : State \bullet s = t_0 \land Abs$

The concrete states related to t_1 that can be reached from a state in C_0 are given by the schema C_1.

C_1

$CState$

$\exists\, s : State \bullet s = t_1 \land Abs$

$\exists\, CState' \bullet (\exists\, C_0 \bullet COp_1 \lor \ldots \lor COp_n) \land \theta CState = \theta CState'$

This schema defines the states that are related to t_1 and which are post-states of one of the operations $COp_1, \ldots, COp_n$ when the pre-state is C_0. Note that the final conjunct of the second predicate equates the post-state variables to the variables of the schema's declaration part. The conjunct is out of the scope of the existentially quantified variables satisfying C_0.

Following this pattern, for any abstract state t_i $(1 \leqslant i \leqslant m)$ the concrete states related to t_i that can be reached from the initial state via concrete states related to $t_1, \ldots, t_{i-1}$ are given by the schema C_i.

C_i

$CState$

$\exists\, s : State \bullet s = t_i \land Abs$

$\exists\, CState' \bullet (\exists\, C_{i-1} \bullet COp_1 \lor \ldots \lor COp_n) \land \theta CState = \theta CState'$

If such a schema evaluates to false, there is no reachable concrete state related to the abstract state and hence we have a false counter-example.

5.2 The example revisited

The abstract specification of the unique number allocator in Section 4 is too abstract. It does not have the property that s can never equal s_2 twice (corresponding to the concrete property that $n!$ can never equal some $v : \mathbb{N}_1$ twice). Hence, if we tried to check this property with a model checker a counter-example would be returned. Model checkers generally return the counter-example that is found first (i.e., the shortest one). In the example, this is $\langle s_1, s_2, s_2 \rangle$ (see Figure 5).

Applying the above process to this counter-example, schema C_0 would be:

C_0

$Init_{IO}$

$\exists\, s : State \bullet$
$\quad s = s_1 \land$
$\quad n! \neq v \Rightarrow s = s_1 \land$
$\quad n! = v \Rightarrow s = s_2$

which simplifies to:

C_0

$Init_{IO}$

since $n!$ does not equal v in $Init_{IO}$; it equals $\bot$.

Schema C_1 would then be:

C_1

$Allocator_{IO}$

$\exists\, s : State \bullet$
$\quad s = s_2 \land$
$\quad n! \neq v \Rightarrow s = s_1 \land$
$\quad n! = v \Rightarrow s = s_2$
$\exists\, Allocator'_{IO} \bullet$
$\quad (\exists\, Init_{IO} \bullet Request_{IO} \lor Send_{IO}) \land \theta Allocator'_{IO} = \theta Allocator_{IO}$

which simplifies to:

C_1

$Allocator_{IO}$

---

false

since initially (when $alloc = \varnothing$), $Request_{IO}$ requires $n!'$ to equal $\bot$ and $Send_{IO}$ requires that it equals 0. Hence, $\langle s_1, s_2, s_2 \rangle$ is a false counter-example.

This process is again potentially automatable using a theorem prover to perform the simplifications either automatically or with some user guidance. False counter-example detection would also be possible using the algorithms underlying a Z animator such as Possum [DHT97].

5.3 Abstraction refinement

Once a false counter-example has been detected, we need to refine the abstract model to avoid this counter-example during subsequent model checking. Similarly to the approach of Clarke et al. [CGJ$^+$00], we need to split the abstract state that is related (via the abstraction function) to the last reachable concrete

state (see Figure 4). We separate those states of the corresponding equivalence class that can perform a transition to the next equivalence class from those that cannot. This separation results in two new equivalence classes which are then mapped into two new abstract states.

In our approach, the last reachable concrete state is defined by the schema C_i where C_{i+1} is the first schema whose predicate simplifies to false. We split the corresponding abstract state t_i into two: one state where there is a transition to the state t_{i+1} and another where there is no transition to the state t_{i+1}.

This manifests itself as replacement of the value of t_i by two new values s_{n+1} and s_{n+2} in type *State*, and a change in the abstraction function. The predicate $p \Rightarrow s = t_i$ will be replaced by two new predicates:

$$p \wedge (\exists\, CState' \bullet (COp_1 \vee \ldots \vee COp_n) \wedge q) \Rightarrow s = s_{n+1}$$

and

$$p \wedge (\nexists CState' \bullet (COp_1 \vee \ldots \vee COp_n) \wedge q) \Rightarrow s = s_{n+2}$$

The first predicate models the case where the concrete state satisfies the monomial p corresponding to abstract state t_i and there is a transition via a concrete operation to a concrete state satisfying the monomial q corresponding to abstract state t_{i+1}. The second predicate models the case where there is no such transition.

Given this new abstraction function, the refined abstract specification and property can then be constructed as before.

5.4 Refining the example

If we apply the above to our example and the counter-example of Section 5.2, the last reachable state is defined by the schema:

C_0

$Init_{IO}$

and we need to split the abstract state $s = s_1$ by changing the definition of *State* to:

$$State ::= s_3 \mid s_4 \mid s_2$$

and replacing the predicate $n! \neq v \Rightarrow s = s_1$ in *Abs* by:

$$n! \neq v \wedge (\exists\, Allocator'_{IO} \bullet (Request_{IO} \vee Send_{IO}) \wedge n = v) \Rightarrow s = s_3$$

which simplifies to $n! \neq v \wedge alloc = \{v\} \Rightarrow s_3$ and:

$$n! \neq v \wedge (\nexists Allocator'_{IO} \bullet (Request_{IO} \vee Send_{IO}) \wedge n = v) \Rightarrow s = s_4$$

which simplifies to $n! \neq v \wedge alloc \neq \{v\} \Rightarrow s_4$.

Following the abstraction process as before, the abstract initial state schema will simplify to:

Ainit

AState

$s = s_4$

and the operations to:

ARequest

$\Delta AState$

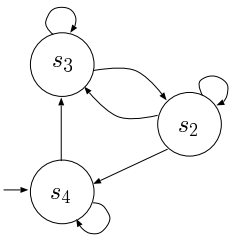
$s = s_3 \Rightarrow s' = s_3$
$s \in \{s_2, s_4\} \Rightarrow s' \in \{s_3, s_4\}$

ASend

$\Delta AState$

$s = s_3 \Rightarrow s' = s_2$
$s = s_4 \Rightarrow s' = s_4$

This abstract specification defines the temporal structure in Figure 6.



$$(n! = v) \mapsto s_2$$
$$(n! \neq v \wedge alloc = \{v\}) \mapsto s_3$$
$$(n! \neq v \wedge alloc \neq \{v\}) \mapsto s_4$$

Fig. 6. First refinement of the abstract specification of the unique number allocator

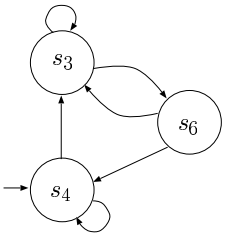
This structure avoids the above counter-example by not allowing a transition from the initial state s_4 to s_2. A transition to s_2 can only occur from s_3 which is related to concrete states where $alloc = \{v\}$. However, the property, now abstracted to:

$$\mathbf{G}\ (s \in \{s_3, s_4\} \vee \mathbf{X}\ (\mathbf{G}\ s \in \{s_3, s_4\}))$$

can still not be proved. In fact, three more refinements are necessary.

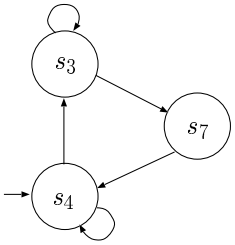
The first refinement, in response to the false counter-example $\langle s_4, s_3, s_2, s_2 \rangle$, results in splitting s_2 into states s_5 and s_6 related to the concrete states where $n! = v \wedge alloc = \{v\}$ and $n! = v \wedge alloc \neq \{v\}$ (the former of which is unreachable, i.e., has no ingoing transitions). The reachable sub-graph of the temporal structure is shown in Figure 7.

The second refinement, in response to the false counter-example $\langle s_4, s_3, s_6, s_3, s_6 \rangle$, results in splitting s_6 into states s_7 and s_8 related to the concrete states where $n! = v \wedge alloc \neq \{v\} \wedge v \in used$ and $n! = v \wedge alloc \neq \{v\} \wedge$



$$(n! \neq v \wedge alloc = \{v\}) \;\mapsto\; s_3$$
$$(n! \neq v \wedge alloc \neq \{v\}) \;\mapsto\; s_4$$
$$(n! = v \wedge alloc \neq \{v\}) \;\mapsto\; s_6$$

Fig. 7. Second refinement of the abstract specification of the unique number allocator

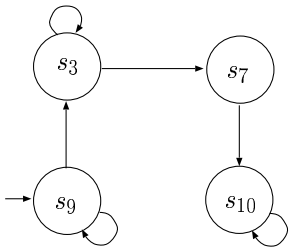


$$(n! \neq v \wedge alloc = \{v\}) \;\mapsto\; s_3$$
$$(n! \neq v \wedge alloc \neq \{v\}) \;\mapsto\; s_4$$
$$(n! = v \wedge alloc \neq \{v\} \wedge v \in used) \;\mapsto\; s_7$$

Fig. 8. Third refinement of the abstract specification of the unique number allocator

$v \notin used$ (the latter of which is unreachable). The reachable sub-graph of the temporal structure is shown in Figure 8.

The final refinement, in response to the false counter-example $\langle s_4, s_3, s_7, s_4, s_3, s_7 \rangle$, results in splitting s_4 into states s_9 and s_{10} related to the concrete states where $n! \neq v \wedge alloc \neq \{v\} \wedge v \notin used$ and $n! \neq v \wedge alloc \neq \{v\} \wedge v \in used$. The reachable sub-graph of the corresponding temporal structure is shown in Figure 9.



$$(n! \neq v \wedge alloc = \{v\}) \;\mapsto\; s_3$$
$$(n! = v \wedge alloc \neq \{v\} \wedge v \in used) \;\mapsto\; s_7$$
$$(n! \neq v \wedge alloc \neq \{v\} \wedge v \notin used) \;\mapsto\; s_9$$
$$(n! \neq v \wedge alloc \neq \{v\} \wedge v \in used) \;\mapsto\; s_{10}$$

Fig. 9. Final refinement of the abstract specification of the unique number allocator

The abstracted property:

$$\mathbf{G}\ (s \in \{s_3, s_9, s_{10}\} \vee \mathbf{X}\ (\mathbf{G}\ s \in \{s_3, s_9, s_{10}\}))$$

can be proved in this case and hence the process terminates.

As the example shows, a number of refinements may be required before a property can be proved (or a real counter-example found). However, the process of false counter-example detection and refinement is systematic and hence potentially automatable with theorem prover support for simplifying predicates.

6 Conclusion and Future Work

In this paper, we have suggested a methodology for model checking Z specifications through an iterative process that employs abstraction and stepwise refinement of large or even infinite models. As the first step of this process, we derive an abstraction function from the set of atomic predicates given in the property to be proved. This abstraction function allows us to generate an abstract model which can be treated by a model checker. By means of Z downward simulation laws, we proved that the derived abstraction function satisfies the conditions for a retrieve relation. Therefore, the generation of the abstract model is sound, i.e., all properties preserved by downward simulation and that hold in the abstract model are also properties of the concrete model.

However, if a property is not satisfied in the abstract model the output counter-example indicating the violation might be a false counter-example having no corresponding execution in the concrete model. In this case, the false counter-example is used to refine the abstract model into a model closer to the concrete model. This refined model will avoid the false counter-example and can be model checked again. We iterate these last two steps of model checking and refining the model until the property is either satisfied by the model or a real counter-example is found.

Other work on abstraction techniques for Z or related languages has been published. Jackson [Jac94] (for Z) as well as Wehrheim [Weh99] (for CSP-OZ), suggest some general user guidance on how to find a suitable abstraction function. However, they do not provide any systematic support for the generation of such an abstraction function. Mota et al. [MBS02] go further in that they suggest an algorithm for generating an abstraction function for the language CSP_Z. Whereas our approach for this generation works syntactically based on a combination of Z predicates, their approach is based on comparing states and the enabledness of operations in all executable paths of a given specification. States with the same operations enabled are considered as belonging to the same equivalence class. This procedure immediately raises the question of efficiency for more complex examples in which arbitrarily many executions, or executions that do not show any behavioural pattern that repeatedly occurs, have to be analysed. Moreover, Mota et al. do not handle false counter-example detection and the corresponding refinement of the abstract model.

The techniques we have adapted have been automated in the model checking community. We claim that our approach for Z is potentially automatable. The major difficulty arising is that arbitrarily complex predicates need to be simplified. While in many cases this could be done automatically with the aid of theorem prover tactics, there may still be a need for some user guidance. In any case, we need to consider modifications aimed at producing better abstractions and abstraction refinements and also examine the general efficiency of our approach.

Our approach is restricted to Z specifications with totalised operations since, without totalisation, arbitrary state changes result in specifications without many interesting temporal properties. This restriction allowed us to treat operations as being guarded, rather than as having preconditions outside of which their occurrence can change the state arbitrarily. This means our approach can also be used with guarded interpretation of Z [DB01] and with Object-Z [Smi00]. Using a structured notation like Object-Z would allow us to additionally use specification decomposition as a means of reducing complexity. This has been examined by Winter and Smith [WS03] and could be combined with this work to increase its effectiveness in dealing with large specifications.

Acknowledgements

This work was supported by a University of Queensland External Support Enabling Grant.

References

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis framework. In *6th ACM Symposium on Principles of Programming Languages*, 1979.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In A.P. Sistla E.A. Emerson, editor, *Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*. Springer-Verlag, 2000.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGP00] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, 2001.

[DHT97] P. Strooper D. Hazel and O. Traynor. Possum: An animator for the SUM specification language. In W. Wong and K. Leung, editors, *Asia Pacific Software Engineering Conference (APSEC 97)*, pages 42–51. IEEE Computer Society, 1997.

[Eme90] 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.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Int. Conf. on Computer Aided Verification (CAV 97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[Jac94] D. Jackson. Abstract model checking of infinite specifications. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Formal Methods Europe (FME'94)*, volume 873 of *LNCS*, pages 519–531. Springer-Verlag, 1994.

[KSW96] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs 96)*, volume 1125 of *LNCS*, pages 283–298. Springer-Verlag, 1996.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), 1995.

[MBS02] A. Mota, P. Borba, and A. Sampaio. Mechanical abstraction of CSP_Z processes. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods Europe (FME'2002)*, volume 2391 of *LNCS*, pages 163–183. Springer-Verlag, 2002.

[Saa97] M. Saaltink. The Z-Eves system. In J. Bowen, M. Hinchey, and D. Till, editors, *International Conference of Z User (ZUM 97)*, volume 1212 of *LNCS*, pages 72–85. Springer-Verlag, 1997.

[Smi00] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV 99)*, volume 1633 of *LNCS*, pages 443–454. Springer-Verlag, 1999.

[TM95] I. Toyn and J. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software - Practice and Experience*, 25(3):305–330, 1995.

[Weh99] H. Wehrheim. Data abstraction for CSP-OZ. In J. Woodcock and J. Wing, editors, *World Congress on Formal Methods (FM'99)*, volume 1709 of *LNCS*. Springer-Verlag, 1999.

[WS03] K. Winter and G. Smith. Compositional verification for Object-Z. In *3rd International Conference of Z and B Users (ZB 2003)*, LNCS. Springer-Verlag, 2003. This volume.

[WVF97] J. M. Wing and M. Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.