A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking

Edd Turner[1], Michael Butler[2] and Michael Leuschel[3]

[1]Department of Computing, University of Surrey,
[2]Electronics and Computer Science, University of Southampton,
[3]Institut für Informatik, Heinrich-Heine Universität Düsseldorf

Abstract. Symmetry reduction is a model checking technique that can help alleviate the problem of state space explosion, by preventing redundant state space exploration. In previous work, we have developed three effective approaches to symmetry reduction for B that have been implemented into the ProB model checker, and we have proved the soundness of our state symmetries. However, it is also important to show our techniques are sound with respect to standard model checking, at the algorithmic level. In this paper, we present a retrospective B development that addresses this issue through a series of B refinements. This work also demonstrates the valuable insights into a system that can be gained through formal modelling.

Keywords: B, refinement, model-checking, symmetry reduction

1 Introduction

The B language is an established formal modelling notation whose salient feature is its support for the incremental refinement of abstract specifications into concrete implementations. A B specification (machine) comprises a collection of variables and operations that may manipulate these variables, together with an invariant on the variables.

Formal verification in B typically requires the use of semi-automatic theorem provers (e.g., B4Free [1], Atelier-B [2], the B-Toolkit [3] and Rodin [4]) to prove that the operations of a machine preserve the invariant, and that each refinement is valid. Model checking is a valuable, alternative approach that can perform these tasks automatically, as with the ProB model checker [5].

Previously, we have focused on addressing the *state space explosion* challenge that faces model checking [6,7,8]. This is where a linear increase in the size of a specification leads to a combinatorial increase in the number of states that the model checker must explore. The impact is that checking large specifications becomes intractable. Our work relied on the identification of *symmetric* states that satisfy the same predicates [6, Theorem 1], and the implementation of an augmented model checking algorithm in ProB that checks only one state from each symmetry class. Experimental results were encouraging and have been shown to reduce the time of model checking by up to three orders of magnitude, e.g., [8]. Moreover, these techniques have been integrated into the final version of the tool. Complementary to this work, it is also important to guarantee the soundness of our approaches, with respect to standard model checking. That is, if standard model checking exhausts its search space without finding an error, called a *counterexample*, then it must be guaranteed that symmetry reduced checking exhausts its constrained search space without finding a counterexample. In [6], we sketched a proof that shows this. In this paper, we go a step further and present a complete B development that shows the soundness of our methods through B refinement. In doing so, we provide details of the model checking algorithms used in terms of their key variables, and we make clear the system properties that contribute to the soundness result.

The B development we present was specified and proved interactively using B4Free's graphical interface, Click'n Prove [9]. Alternatively, we could have used the next generation of B, Event-B [10] and the Rodin tool [4]. However, we did not find our choice inhibited development. Instead, as is common to formal modelling in general, the most time-consuming aspect was the iterative process of finding a suitable abstraction of the system that captures the information we required, in addition to discovering invariants important for refinement.

We proceed by presenting the abstract specification for model checking in Section 2 and an immediate refinement in Section 3. Section 4 provides a refinement machine whose behaviour closely models standard model checking in ProB. Next, the refinement in Section 5.1 adheres to our style of symmetry reduction implemented in [6], and Section 5.2 gives a refinement that matches our symmetry reduction strategy used in [7,8]. Finally, we provide a discussion of our work in Section 6. For clarity of presentation, each machine is broken into several parts, which are individually explained. Each machine specifies the same set of operations, as required by B refinement, although they are only included in the commentary when necessary.

2 An Abstract Specification for Model Checking

The abstract specification, *mc0*, introduces the sets and constants that are required to capture the overall behaviour of a model checking procedure, as used by ProB. These are used to specify two mutually exclusive events that determine when model checking can terminate. We begin by introducing the sets, constants and properties used by this machine. The B encoding is given in Figure 1.

The *mc0* machine uses two sets, S and $ANSWER$. Deferred set, S denotes all possible states of the system being model checked (i.e., the cartesian product of types of the machine variables). Given that bounds are placed on system types during model checking in ProB, $|S|$ is finite. The enumerated set, $ANSWER$ denotes the two, mutually exclusive, choices of message that are output once model checking terminates; either *Pass* (the reachable search space has been exhausted without finding a state that violates the invariant, i.e., a counterexample), or *Fail* (a reachable counterexample has been found).

```
MACHINE mc0
SETS
   S; ANSWER = {Pass,Fail}
CONSTANTS
   i, /* special initial state */
   tr, /* transition relation */
   inv, /* states satisfying invariant */
   reach /* reachable states */
PROPERTIES
   tr ∈ S ↔ S ∧
   inv ∈ ℙ(S) ∧
   i ∈ inv ∧
   i ∉ ran(tr) ∧
   /* the reachable states */
   reach ∈ ℙ(S) ∧
   i ∈ reach ∧
   /* reach is a fix-point */
   tr[reach] ⊆ reach ∧

   /* reach is the smallest fix-point of
   the reachable states */
   ∀(r).(r ∈ ℙ(S) ∧
      i ∈ r ∧
      tr[r] ⊆ r ⇒
         reach ⊆ r)
```

Fig. 1. The Sets, Constants and Properties of the Abstract Machine, *mc0*

There are four important constants used for the abstract specification (see also Figure 3). Defining the behaviour of the system is *tr*, the transition relation over states in *S*. The set of correctness conditions checked by the algorithm is defined implicitly through *inv*; the subset of *S* satisfying the correctness conditions. Such an approach is sufficient for the standard model checking of B systems in ProB, since checking involves only the evaluation of the invariant for the variables values[1]. A special state, *i*, is used to indicate the case where the variables used by the specification have not yet been initialised. Successors of *i* represent the initialisation of a machine, $tr[\{i\}]$. It follows that *i* is always the root state of the search space. The set of states encountered during model checking, denoted *reach*, is defined by a fix-point on *tr*, where $tr[reach] \subseteq reach$, i.e., the successor of a reachable state is also reachable. Further, we specify *reach* as the least fix-point of *tr*.

```
ok ← pass ≘
      WHEN reach ⊆ inv
      THEN ok := Pass
      END;

ok ← fail ≘
      WHEN reach ⊈ inv
      THEN ok := Fail
      END
```

Fig. 2. The Operations of the Abstract Machine, *mc0*

The operations of *mc0* are given in Figure 2. These include *pass* and *fail*, which are mutually exclusive events that specify the conditions under which model checking terminates. The *pass* operation is enabled if all reachable states satisfy the correctness conditions used during checking ($reach \subseteq inv$). In which case, the *Pass* message is specified as a return parameter. Conversely, *fail* is enabled if the set of reachable states do not satisfy the correctness conditions, and the *Fail* message is output by the algorithm. In contrast to an implementation

[1] ProB also supports the bounded verification of LTL formulae [11].

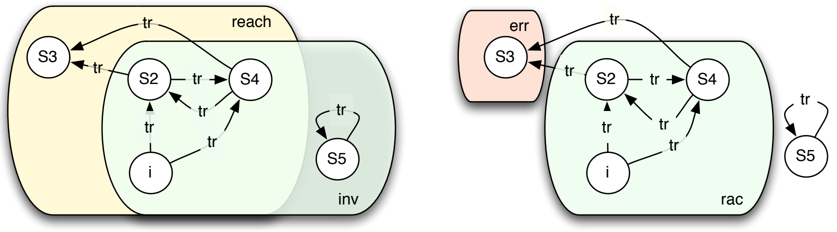

Fig. 3. Illustrating the constants of mc0 and variables of mc1

REFINEMENT *mc1*
REFINES *mc0*
VARIABLES
  rac, /* *reached and checked* */
  err /* *reached errors* */
INVARIANT
  $rac \subseteq reach \wedge$
  $rac \subseteq inv \wedge$
  $i \in rac \wedge$
  $err \subseteq reach \setminus inv$
INITIALISATION
  $rac := \{i\} \ ||$
  $err := \varnothing$

Fig. 4. The Variables, Invariant and Initialisation of *mc1*

of a model checking algorithm, this abstract specification either immediately passes or fails. However, this is sufficient since its single goal is to capture the key properties of the procedure. Details used by an implementation, such as variable information, are given in refinements of *mc0*.

3 Refinement Level 1

Let us now present *mc1*, the first level of refinement for *mc0* (i.e., $mc0 \sqsubseteq mc1$). This refinement introduces two key variables and two events that will be required in an implementation of a model checking algorithm. Their use is generalised so that later refinements can specify their precise roles during both standard and symmetry reduced model checking. We found that this modularised the proof effort required for these algorithms. Note that this generalisation was devised *after* developing and attempting to prove the separate models for the two algorithms (presented in Sections 4 and 5), when we realised that this refinement was a common abstraction that facilitates proof. Figure 4 presents the new variables, invariant and initialisation clauses of *mc1* (see also Figure 3).

Variable *rac* is introduced to store all states reached by model checking so far, which satisfy the correctness conditions. Conversely, *err* stores those states reached by model checking that violate the correctness conditions.

Regarding the operations, *mc1* introduces two events used during the *traversal* of the state space. The operation, *add_inv*, models the checking of states that satisfy the correctness conditions (and in later refinement machines also determines states yet to be checked). Conversely, *add_err*, models the checking of counterexamples. We separate the events for state space traversal since we find this style convenient for proof. The operations of *mc1* are given in Figure 5.

```
$add\_inv \mathrel{\widehat{=}}$ /* new event */
ANY $ss$ WHERE
    $ss \subseteq reach \setminus rac \land$
    $ss \subseteq inv \land$
    $ss \neq \varnothing$
THEN
    $rac := rac \cup ss$
END;

$add\_err \mathrel{\widehat{=}}$ /* new event */
ANY $ss$ WHERE
    $ss \subseteq reach \setminus rac \land$
    $ss \neq \varnothing \land$
    $ss \cap inv = \varnothing$
THEN
    $err := err \cup ss$
END;

$ok \leftarrow pass \mathrel{\widehat{=}}$
WHEN
    $reach \subseteq rac$
THEN
    $ok := Pass$
END;

$ok \leftarrow fail \mathrel{\widehat{=}}$
WHEN
    $err \neq \varnothing$
THEN
    $ok := Fail$
END
```

Fig. 5. The Operations of *mc1*

Observe that the *add_inv* event selects a non-empty subset from the reachable states, which are yet to be reached, and which also satisfy the correctness conditions. This subset is added to *rac*, ensuring they will not be encountered again. Similarly, the *add_err* event selects a non-empty subset from the reachable states, yet to be reached, but which contain no elements satisfying the correctness conditions, i.e., are invariant violations. These violations are added to *err* for a permanent record.

We refine the *pass* operation by specifying its guard as $reach \subseteq rac$. That is, *pass* should become enabled when the reachable search space has been fully covered by *add_inv*. To show the validity of a refined event, we must prove that the guard of the abstract operation (G) can be derived from the new guard (G') together with the machine invariant (Inv), i.e., $Inv \land G' \Rightarrow G$ [12]. This is straightforward, since $rac \subseteq inv \land reach \subseteq rac \Rightarrow reach \subseteq inv$. We also change the guard of the *fail* operation to simply, $err \neq \varnothing$, which is intuitive because its satisfaction indicates an error has been encountered by the *add_err* operation. Proving that this refinement is valid is also simple since, $err \subseteq reach \setminus inv \land err \neq \varnothing \Rightarrow reach \not\subseteq inv$.

Given that we are model checking a finite state system, it is desirable to prove the termination of the state space exploration algorithm specified in *mc1*, which occurs when *pass* or *fail* enables. This can be shown by providing the variant, $\mid reach \setminus (rac \cup err) \mid$, which represents the number of remaining states yet to be explored. Then, we note that successive applications of *add_inv* and *add_err* decreases the value of the variant progressively, until at some point no new states can be added to *rac* or *err*, and therefore, *add_inv* or *add_err* can

```
REFINEMENT mc2
REFINES mc1
VARIABLES
    unex, /* reached not fully explored */
    rac,  /* reached and checked */
    err   /* reached errors */
INVARIANT
    unex ⊆ rac ∧
    tr[rac \ unex] ⊆ rac ∪ err
INITIALISATION
    unex := {i} ||
    rac := {i}  ||err := ∅
```

Fig. 6. The Variables, Invariant and Initialisation of *mc2*

no longer be enabled. This ensures that *pass* or *fail* will eventually engage. In the case where errors exist, *fail* enables. If *add_inv* and *add_err* block, then all reachable states have been checked, without error, and *pass* enables. Thus, we have shown the algorithm specified in *mc1* terminates. The addition of variants to a system is not supported in classical B and its B provers[2]. However, we have provided a variant here to help illustrate the validity of *mc1*.

4 Refinement for Standard Model Checking

The B machines *mc0* and *mc1* given in the previous sections are specified at a high level: certain details are not included that would be required for an implementation of the algorithm. This section addresses this issue through a single refinement of *mc1* that specifies more closely the standard model checking algorithm, and as a consequence, highlights several key properties. Figure 6 shows the variables, invariant and initialisation clauses of this machine.

As can be seen, *mc2* introduces a single variable, *unex*. The purpose of this variable is to store all states reached by model checking so far, which satisfy the correctness conditions, but whose successors are yet to be determined. Moreover, it is defined as a subset of *rac*, since each state it stores will be reached via the transition relation from the root state i, and subsequently checked.

In addition, note that a new invariant condition is added: $tr[rac \setminus unex] \subseteq rac \cup err$. This constitutes the basis of proving when model checking can terminate, given that no violations exist. To clarify its use, we first present the behaviour of the operations in this machine, given in Figure 7.

We introduce the *remove* operation to remove a state from *unex* whenever all of its successors have been reached, and therefore are elements of *rac*. The repeated application of *remove* will cause *unex* to diminish in size, indicating that fewer transitions remain to be explored. This can be expressed formally as a simple variant, $\mid unex \mid$, whose size decreases upon the action of *remove*.

The *add_inv* event of *mc1* is refined to select a single state from *unex* (a state whose transitions have not yet all been traversed), and computes a single successor of it (*s2*) that satisfies the correctness conditions. The successor is added to both *unex* and *rac*. In the case where the successor is an invariant violation, it is added to only *err* in the *add_err* operation. Addition to either *unex* or *rac* would, otherwise, break the invariant, $unex \subseteq rac \wedge rac \subseteq inv$.

[2] Event-B and its associated provers provide support for variants.

```
$add\_inv \;\widehat{=}$
ANY $s1,s2$ WHERE
    $s1 \in unex\ \wedge$
    $s2 \in inv\ \wedge$
    $s1 \mapsto s2 \in tr\ \wedge$
    $s2 \notin rac\ \wedge$
    $err = \varnothing$
THEN
    $unex := unex \cup \{s2\}\ ||$
    $rac := rac \cup \{s2\}$
END;

$add\_err \;\widehat{=}$
ANY $s1,s2$ WHERE
    $s1 \in unex\ \wedge$
    $s2 \notin inv\ \wedge$
    $s1 \mapsto s2 \in tr\ \wedge$
    $err = \varnothing$
THEN
    $err := err \cup \{s2\}$
END;
```

```
    $remove \;\widehat{=}$ /* new event */
    ANY $s1$ WHERE
        $s1 \in unex\ \wedge$
        /* all s1's successors checked */
        $\mathrm{tr}[\{s1\}] \subseteq rac\ \wedge$
        $err = \varnothing$
    THEN
        $unex := unex \setminus \{s1\}$
    END;
$ok \leftarrow pass \;\widehat{=}$
    WHEN
        $unex = \varnothing\ \wedge$
         $err = \varnothing$
    THEN
        $ok := Pass$
    END;
$ok \leftarrow fail \;\widehat{=}$
    WHEN $err \neq \varnothing$
    THEN
        $ok := Fail$
    END
```

Fig. 7. The Operations of *mc2*

A number of assertions are also specified in *mc2*, to verify the preservation of responsiveness of the specified model checking algorithm[3]. We do not show them because they simply consist of a disjunction of the guards of each operation. Their proof with B4Free guarantees that there is always at least one enabled operation, e.g., model checking has not yet finished, so one can perform either *add_inv*, *add_err* or *remove*, or conversely, state space exploration has terminated and either *pass* or *fail* is enabled.

Given the responsiveness of this machine, in addition to the previous variants specified for the *add_inv*, *add_err* and *remove* operations, which show that eventually these operations are all blocked, we can deduce that either *pass* or *fail* will eventually be enabled. This relies on *pass* and *fail* being valid refinements of their abstract specification. This is trivial for the *fail* operation, since it remains unchanged from *mc1*. The goal for the *pass* operation is to show that $Inv \wedge unex = \varnothing \wedge err = \varnothing \Rightarrow reach \subseteq rac$. By choosing the appropriate invariant, we have:

$$\begin{aligned} & tr[rac \setminus unex] \subseteq rac \cup err \\ & = tr[rac] \subseteq rac && \textit{Given } unex = \varnothing \text{ and } err = \varnothing \end{aligned}$$

That is, rac is a fix-point of tr. Since $reach$ is the least fix-point, we can conclude that $reach \subseteq rac$.

[3] An assertion in B is an expression over the sets, constants, properties, variables or invariant clauses of a B machine. They enable one to form corollaries in B. By proving an assertion, it is made available for use inside other proof activities.

The overall chain of refinement developed for standard model checking consists of: *mc0* $\sqsubseteq$ *mc1* $\sqsubseteq$ *mc2*. That is, *mc2* is a valid refinement of *mc0*. Therefore, the model checking algorithm specified in *mc2* is sound with respect to the abstract specification of model checking. In the next section, we introduce the notion of symmetry reduction into our specifications.

5 Refinements for Symmetry Reduced Model Checking

This section presents two refinement machines that specify symmetry reduced model checking through the refinement of *mc1* (Section 3), namely *rmc1* and *rmc2*. These refinements follow closely the specification of *mc2*, except they introduce the concept of symmetry between states of a system.

5.1 Level 1

The primary purpose of the first refinement machine for symmetry reduced model checking is to provide the first step towards integrating symmetry reduction into the B specification of standard model checking, whilst linking the variables used by the standard and reduced approaches. Through this machine, we also show that our original work in symmetry reduction [6] is sound with respect to the abstract specification of model checking. In this particular strategy, called *permutation flooding*, each unexplored state encountered is first checked against the invariant. Then, all states symmetric to it (which we have proved satisfy the same predicates) are computed and are added to the state space: these states are marked as explored so that model checking need not explicitly check them. The concept of state symmetries is specified using constants and properties, and is given in Figure 8.

The symmetries of a system are defined over the transition relation in terms of sets of special permutations (called *automorphisms*), denoted *aut*. We also specify two key properties of automorphisms, as given in [13, Chapter 14]:

- an inverse of an automorphism is itself an automorphism, and
- automorphisms preserve the transition relation (a result also shown in [6]).

In the context of this specification, we define that the special state i (representing the uninitialised machine) is symmetric only to itself. In addition, we specify a consequence of a result in [6, Corollary 1], which proves that symmetric states satisfy the same predicates. That is, a state satisfies the invariant, *iff* states symmetric to it also satisfy the invariant.

A valid automorphism p for the example from Figure 3 is shown in Figure 9 (dashed lines represent the transition relation), where $S2$ and $S4$ are permuted for each other and all other states are kept unchanged. In terms of a B machine, a state comprises the values of its variables. Intuitively, two states, such as $S2$ and $S4$, are symmetric if the values of one state can be transformed into those of the other. In addition, a sequence of state transitions (i.e., operations) possible from one state will also be possible from the other; this is also depicted in Figure 9.

REFINEMENT *rmc1*
REFINES *mc1*
CONSTANTS
  aut, /* *automorphisms on tr* */
  rep /* *representative function* */
PROPERTIES
  $aut \in \mathbb{P}(S \rightarrowtail\!\!\!\!\twoheadrightarrow S) \wedge$
  $id(S) \in aut \wedge$
  $\forall(p).(p \in aut \Rightarrow p^{-1} \in aut) \wedge$
  $\forall(p).(p \in aut \Rightarrow i \mapsto i \in p) \wedge$

  /* *automorphisms preserve invar.* */
  $\forall(p,s1,s2).(p \in aut \wedge$
    $s1 \mapsto s2 \in p \Rightarrow$
   $(s1 \in inv) \Leftrightarrow (s2 \in inv)) \wedge$

  $rep \in S \rightarrow S \wedge$

/* *P1: automorphisms preserve tr* */
$\forall(p,s1,s2).(p \in aut \wedge s1 \in S \wedge$
  $s2 \in S \Rightarrow$
 $(s1 \mapsto s2 \in tr) \Leftrightarrow$
 $(p(s1) \mapsto p(s2) \in tr)) \wedge$

/* *symmetries have same rep.* */
$\forall(p,s1,s2).(p \in aut \wedge$
  $s1 \mapsto s2 \in p \Rightarrow$
 $rep(s1) = rep(s2)) \wedge$

/* *s and rep(s) implies auto.* */
$\forall(s1,s2).(s1 \mapsto s2 \in rep \Rightarrow$
 $\exists(p).(p \in \mathrm{aut} \wedge s1 \mapsto s2 \in p)) \wedge$

/* *representatives are fix-points* */
$\forall(s).(s \in ran(rep) \Rightarrow rep(s) = s)$

Fig. 8. The Constants and Properties of the Machine, *rmc1*

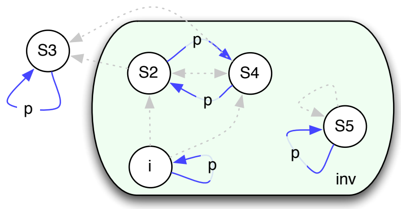

Fig. 9. An automorphism for Figure 3

The constant, *rep*, is introduced to model an algorithm that computes a unique representative for some given state, and is defined over the set of states S. We have implemented this function in ProB, which determines a representative state from the set of states symmetric to it [6].

It follows that checking one state during the reduced search, corresponds to checking all symmetric states in the standard search. The *rep* function in this refinement is constrained accordingly (the first 3 properties involving *rep*). Further, we specify representatives as fix-points. Assertions for *rmc1* are given in Figure 10, whose proof simplifies later proof activities required to guarantee its consistency and show that it is a valid refinement of *mc1*.

There are five assertions defined for this machine, of which the first four are relatively simple and follow from the properties of *aut* and *rep*. The last assertion requires proof that for any reachable state its representative state is also reachable. To show this it is instructive to present a fix-point proof over automorphisms, upon which *rep* is based. Using the property of automorphisms marked $P1$ in Figure 8, we begin by proving for any automorphism p, that $p[reach]$ is a fix-point of tr:

ASSERTIONS

```
/* representatives preserve invar. */      ∀(s1,s2).(s1 ↦ s2 ∈ tr ⇒
∀(s1,s2).(s1 ∈ S ∧                            ∃(ss2).(rep(s1) ↦ ss2 ∈ tr ∧
    s2 ∈ S ∧                                      rep(s2) = rep(ss2))) ∧
    s1 ↦ s2 ∈ rep ⇒
        ((s1 ∈ inv) ⇔ (s2 ∈ inv))) ∧       /* s is reachable iff
                                            rep(s) is reachable */
rep(i) = i ∧                               ∀(s).(s ∈ S ⇒
rep⁻¹[{i}] = {i} ∧                            ((s ∈ reach) ⇔ (rep(s) ∈ reach)))
```

Fig. 10. The Assertions of *rmc1*

$$\begin{array}{lll}
tr[p[reach]] \subseteq p[reach] & & \text{(A)}\\
\Leftrightarrow \forall\, y \cdot y \in tr[p[reach]] \Rightarrow y \in p[reach] & \textit{inclusion is universal} & \\
\Leftrightarrow (\exists\, x \cdot x \in p[reach] \wedge x \mapsto y \in tr) & \textit{quantify on } p & \\
\qquad\qquad\qquad\qquad \Rightarrow y \in p[reach] & & \\
\Leftrightarrow (\exists\, x \cdot p^{-1}(x) \in reach \wedge x \mapsto y \in tr) & p \textit{ is injective} & \\
\qquad\qquad\qquad\qquad \Rightarrow p^{-1}(y) \in reach & & \\
\Leftrightarrow (\exists\, x \cdot p^{-1}(x) \in reach \wedge p^{-1}(x) \mapsto p^{-1}(y) \in tr) & \textit{property P1} & \\
\qquad\qquad\qquad\qquad \Rightarrow p^{-1}(y) \in reach & & \\
\Leftrightarrow true & &
\end{array}$$

Equation (A) implies $p[reach]$ is a fix-point of tr. Thus, for an automorphism q:

$$reach \subseteq q[reach] \qquad \text{(B)}$$

By monotonicity, from (B) we get:

$$\begin{array}{lll}
q^{-1}[reach] \subseteq q^{-1}[q[reach]] & & \\
\Leftrightarrow q^{-1}[reach] \subseteq reach & q \textit{ is injective} & \text{(C)}
\end{array}$$

Instantiate q with p in (B) to get:

$$reach \subseteq p[reach] \qquad \text{(D)}$$

Instantiate q with p^{-1} in (C) to get:

$$\begin{array}{lll}
(p^{-1})^{-1}[reach] \subseteq reach & & \\
\Leftrightarrow p[reach] \subseteq reach & p \textit{ is injective} & \text{(E)}
\end{array}$$

Finally, from (D) and (E) we obtain the result $p[reach] = reach$. That is, all automorphisms preserve the reachable states.

Six variables are used by this machine, and are shown in Figure 11. Intuitively, they can be split into three pairs, where each pair consists of a variable used

VARIABLES
  /* *vars for standard checking* */
  rac,*unex*,*err*,
  /* *vars for reduced approach* */
  rrac,*runex*,*rerr*
INVARIANT
  $unex \subseteq rac \wedge$
  $rrac \subseteq ran(rep) \wedge$
  $rrac \subseteq rac \wedge$
  $runex \subseteq rrac \wedge$
  $rerr \subseteq err \wedge$
  $rep^{-1}[rrac] = rac \wedge$
  $rep^{-1}[runex] = unex \wedge$
  $rep^{-1}[rerr] = err \wedge$
  $tr[rac \setminus unex] \subseteq rac \cup err$
INITIALISATION
  $rac := \{i\} \mid\mid rrac := \{i\} \mid\mid$
  $unex := \{i\} \mid\mid runex := \{i\} \mid\mid$
  $err := \varnothing \mid\mid rerr := \varnothing$

Fig. 11. The Variables, Invariant and Initialisation of *rmc1*

in the B specification of standard model checking (*rac*, *unex* or *err*), and a corresponding variable introduced to specify reduced checking (*rrac*, *runex* or *rerr*). The key premise is to link each pair with some set of constraints, so that properties that apply to standard checking also apply to the reduced approach.

As with the standard approach to checking, the set of states reached during checking whose successors have not yet all been explored (*unex*), is a subset of the states encountered by model checking (*rac*); $unex \subseteq rac$. To link *rac* and *rrac*, we specify that $rrac \subseteq rac$ and $rep^{-1}[rrac] = rac$; the states symmetric to those of *rrac* are members of *rac*. We specify corresponding constraints for variables *unex*, *runex*, *err*, and *rerr*. In addition, $tr[rac \setminus unex] \subseteq rac \cup err$ is specified to simplify the detection of model checking termination when no counterexamples are found (i.e., when $unex = \varnothing$ and $err = \varnothing$, see *mc2* in Section 4). This will be proved correct in the next refinement using only *rrac*, *runex*, and *rerr*. The operations of *rmc1* are given in Figure 12.

Notice that this machine behaves in a similar way to *mc2*, which also refines *mc1*. The difference regarding the *add_inv* or *add_err* events, is that for each newly encountered state *s* we add its *representative* to *runex* (if *s* satisfies the invariant) or *rerr* (if *s* violates the invariant); while adding all symmetric states, $rep^{-1}[\{s\}]$ to *unex* or *err*. The *remove* operation follows this pattern, and removes a state from *runex* whenever the representatives of all of its successors have been encountered; while all symmetric states are then removed from *unex*.

Justification of the correctness of this refinement is similar to the standard case, presented in Section 4. This involved proving the enabledness preservation of operations, the validity of the refinement and that eventually *pass* or *fail* becomes enabled.

Soundness Result 1: The important observation of this refinement machine is that the style of state space traversal provided by the operations *add_inv*, *add_err* and *remove*, reflects the algorithm we used in our initial work on symmetry reduction in ProB, i.e., permutation flooding. For example, $rep^{-1}[\{rep(s2)\}]$ in the *add_inv* operation in Figure 12 represents all symmetric states of $s2$, which are used to *flood* the variables, *unex* and *rac*. We obtain the assurance that permutation flooding is sound with respect to the abstract specification of standard model checking, since $mc0 \sqsubseteq mc1 \sqsubseteq rmc1$.

```
$add\_inv \mathrel{\widehat{=}}$
**ANY** $s1, s2$ **WHERE**
    $s1 \in runex \wedge$
    $s2 \in inv \wedge$
    $s1 \mapsto s2 \in tr \wedge$
    $rep(s2) \notin rrac \wedge$
    $rerr = \varnothing$
**THEN**
    $runex := runex \cup \{rep(s2)\} \ ||$
    $unex := unex \cup rep^{-1}[\{rep(s2)\}] \ ||$
    $rrac := rrac \cup \{rep(s2)\} \ ||$
    $rac := rac \cup rep^{-1}[\{rep(s2)\}]$
**END**;
$add\_err \mathrel{\widehat{=}}$
**ANY** $s1, s2$ **WHERE**
    $s1 \in runex \wedge$
    $s2 \notin inv \wedge$
    $s1 \mapsto s2 \in tr \wedge$
    $rep(s2) \notin rrac \wedge$
    $rerr = \varnothing$
**THEN**
    $rerr := rerr \cup \{rep(s2)\} \ ||$
    $err := err \cup rep^{-1}[\{rep(s2)\}]$
**END**;
```

```
$remove \mathrel{\widehat{=}}$
**ANY** $s1$ **WHERE**
    $s1 \in runex \wedge$
    /* all s1's successors checked */
    $rep[tr[\{s1\}]] \subseteq rrac \wedge$
    $rerr = \varnothing$
**THEN**
    $runex := runex \setminus \{s1\} \ ||$
    $unex := unex \setminus rep^{-1}[\{s1\}]$
**END**;

$ok \leftarrow pass \mathrel{\widehat{=}}$
**WHEN**
    $rerr = \varnothing \wedge$
    $runex = \varnothing$
**THEN**
    $ok := Pass$
**END**;
$ok \leftarrow fail \mathrel{\widehat{=}}$
**WHEN**
    $rerr \neq \varnothing$
**THEN**
    $ok := Fail$
**END**
```

Fig. 12. The Operations of *rmc1*

5.2 Level 2

In the final refinement for symmetry reduction, we retain only three variables, *rrac*, *runex* and *rerr*, from the relatively detailed *rmc*1, upon which we specify a minimal set of constraints, as shown in Figure 13.

```
**REFINEMENT** $rmc2$

**REFINES** $rmc1$

**VARIABLES**
    $rrac, runex, rerr$

**INVARIANT**
    $i \in rrac \wedge$
    $rrac \subseteq ran(rep) \wedge$
    $rrac \subseteq rac \wedge$
    $runex \subseteq rrac \wedge$
    $rerr \subseteq err$

**INITIALISATION**
    $rrac := \{i\} \ ||$
    $runex := \{i\} \ ||$
    $rerr := \varnothing$
```

Fig. 13. The Variables, Invariant and Initialisation of *rmc2*

Observe that the specification of the variables remains the same as that given in *rmc1*, while all details of *rac*, *unex*, and *err* have been removed. The same applies for the operations of this machine: *add_inv*, *add_err* and *remove* are identical to those in *rmc1*, except that there are no assignments to *rac*, *unex*, and *err*. For this reason, we do not show the operations of this refinement.

We note that the style of state space traversal specified contrasts with that of *rmc1* and instead reflects more closely a classical symmetry reduction algorithm, which we used in [7,8]. Therefore, upon encountering an unexplored state (e.g., *s1* in *add_inv*), we compute and store only the unique representatives of its successors that have not yet been checked ($rep(s2)$); the model checking algorithm will never store two symmetric states, and it has less of a demand for memory. The disadvantage of implementing such a *rep* function is that it can be computationally expensive[4]. We also note that the proof of correctness for *rmc1* is echoed by this machine.

Soundness Result 2: The chain of refinement for our classical approach to symmetry reduced model checking consists of: $mc0 \sqsubseteq mc1 \sqsubseteq rmc1 \sqsubseteq rmc2$. Therefore, by the transitivity of refinement, our augmented algorithm is sound with respect to the abstract specification of model checking.

6 Concluding

We have presented a B development that shows through refinement the soundness of our previous methods for symmetry reduction in ProB, with respect to standard model checking. That is, if standard model checking exhausts its search space without finding a counterexample then symmetry reduced checking must also exhaust its quotient search space without finding a counterexample.

An abstract specification for model checking, *mc0*, is given in Section 2, which is refined by *mc1* in Section 3. From here, two separate two chains of refinement specify details of algorithms that implement the standard and reduced approaches. The refinement branch for the reduced approach includes *rmc1*, which reflects the style of model checking we adopted in [6], and *rmc2*, that reflects the style we used in [7] and [8]. Given that both chains refine the abstract specification, we obtain our desired soundness result.

The system was specified using B and the Click'n Prove tool, although it would have been possible to use Event-B. Indeed, our use of guarded B operations is characteristic of events in Event-B. In addition, we could have utilised the tool support of Event-B when guaranteeing the model checking algorithms eventually terminate having found a counterexample (*fail*) or without finding a counterexample (*pass*), after exploring the reachable state space. This task involved using variants to ensure the *add_inv*, *add_err* and *remove* operations eventually relinquish control (giving *pass* and *fail* an opportunity to be enabled), and proving the preservation of operation enabledness for the system. Despite this, we did not find using B impeded the development process. We recognise though, that if our development had become more complex (e.g., requiring decomposition), it would have been beneficial to use Event-B and its tools.

[4] This *rep* function is based upon algorithms for determining isomorphic graphs, for which there is currently no known polynomial time algorithm.

The B development presented captures properties of model checking that are sufficient to show the overall soundness of our approaches to symmetry reduction. In these specifications, we have removed the details of the algorithms that select a unique representative from a class of symmetric states; as modelled by the *rep* function. Proving that our implementations correctly compute representatives currently remains as future work and would require developing additional formal models. We do not believe this would be difficult for our permutation flooding approach, since the implementation relies on a simple, but effective, permutation function. However, we do think this would be challenging for our two other implementations, which use complex algorithms for determining graph isomorphism, and were inspired by the work of McKay [14]. Additional future work is to extend our B development by adding labels to *tr* so that properties can be proved over paths of the system. This would provide a basis for proving the soundness of refinement via model checking. Finally, it would also be valuable to prove that symmetry reduced model checking preserves LTL properties.

References

1. Clearsy: B4Free tool, Available at `http://www.b4free.com` (2009)
2. Steria, Aix-en-Provence, France: Atelier B, User and Reference Manuals, Available `http://www.atelierb.eu/index-en.php`. (2009)
3. B-Core (UK) Limited: B-Toolkit manuals, Available at `http://www.b-core.com/btoolkit.html` (2002)
4. Abrial, J.R., Butler, M.J., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In Liu, Z., He, J., eds.: ICFEM. Volume 4260 of Lecture Notes in Computer Science., Springer (2006) 588–605
5. Leuschel, M., Butler, M.J.: ProB: A Model Checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME. Volume 2805 of Lecture Notes in Computer Science., Springer (2003) 855–874
6. Leuschel, M., Butler, M.J., Spermann, C., Turner, E.: Symmetry Reduction for B by Permutation Flooding. In Julliand, J., Kouchnarenko, O., eds.: B. Volume 4355 of Lecture Notes in Computer Science., Springer (2007) 79–93
7. Turner, E., Leuschel, M., Spermann, C., Butler, M.J.: Symmetry Reduced Model Checking for B. In: TASE, IEEE Computer Society (2007) 25–34
8. Spermann, C., Leuschel, M.: ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In: TASE, IEEE Computer Society (2008) 15–22
9. Abrial, J.R., Cansell, D.: Click'n Prove: Interactive Proofs within Set Theory. In Basin, D.A., Wolff, B., eds.: TPHOLs. Volume 2758 of Lecture Notes in Computer Science., Springer (2003) 1–24
10. Métayer, C., Abrial, J.R., Voisin, L.: Event-B Language (RODIN, D7) (2005)
11. Leuschel, M., Plagge, D.: Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Ameur, Y.A., Boniol, F., Wiels, V., eds.: ISoLA. Volume RNTI-SM-1 of Revue des Nouvelles Technologies de l'Information., Cépaduès-Éditions (2007) 73–84
12. Abrial, J.R.: The B Book: Assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
14. McKay, B.D.: Practical Graph Isomorphism. Congressus Numerantium **30** (1981) 45–87