UNIVERSITY OF SOUTHAMPTON

Improving the Process of Model Checking through State Space Reductions

by

Edward Nanakorn Turner

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

November 2007

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Edward Nanakorn Turner

Model checking is a technique for finding errors in systems and algorithms. The technique requires a formal definition of the system with a set of correctness conditions, and the use of a tool, the model checker, that searches for model behaviours violating these correctness conditions. The value of existing model checkers depends largely on the complexity of the system being checked. Systems involving complex data structures quickly encounter the problem of state explosion, and checking becomes intractable. Furthermore, auxiliary feedback originally designed to aid the practitioner (e.g., process automata) becomes less useful.

This thesis develops of a set of techniques to address these problems. The main contributions of this thesis are methods that improve model checking in the formal language of B, by reductions in the size of a system's state space. Methods are described that enable a user to view various succinct properties about a system's behaviour through automatic analysis of reached state spaces, and a technique is developed to improve the efficiency of generating state spaces during model checking using algorithms for identifying symmetries via graph isomorphism. Soundness proofs are shown using refinement in B. Each technique has been implemented into the B model checker, called ProB, and is shown to be effective through experimentation and evaluation. This research has stimulated three complementary approaches for improving the generation of state spaces, which are also presented and evaluated. Although this work concerns the context of B and ProB, the techniques could be generalised to verification tools of other languages.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

There are many people who have helped make this thesis happen. First of all, I would like to thank my supervisor, Michael Butler, whose experience and enthusiasm in formal methods and formal verification has made it easy for me to maintain and develop my own interest in the subject. Also, I thank him for all the patient guidance and advice he has given throughout the course of my study. I must thank Michael Leuschel, who originally employed me as a research assistant working with ProB, which initiated my interest in model checking and B. Furthermore, I am very grateful to him for the Ph.D studentship offered to me, which commenced after this period of work – and indeed the numerous interesting discussions we have had since then, concerning the ProB tool-set. I also express gratitude to Corinna Spermann for her contributions to our collaborative work studying McKay's partition refinement algorithm. Finally, and most of all, I would like to thank my family; my mother, father, and two sisters, for their constant support, enthusiasm and entertaining stories, no matter what – and without whom, all of this would not have been possible.

Chapter 1

Introduction

As systems controlled by software increase in size and complexity, the importance of error detection at design time increases. Some estimations suggest that up to 70% of design time is spent performing simulations and tests, to minimise the risk of errors being exposed at a later stage in production, which could require high compensation costs [Schneider, 2003]. This thesis develops a set of *automatic* techniques used for finding errors in systems and algorithms. The techniques concern the subject of model checking; where, given a formal model of a system and its specification, all reachable states of the system are automatically searched for those that violate the specification. The techniques enable a user to view various succinct properties about the states reached, and improve the efficiency of their generation. Each technique has been implemented into the ProB tool-set, a model checker of the formal specification language called B, to produce a model checker with several novel features.

1.1 Model Checking

Model checking [Clarke et al., 1999] is an automatic strategy for finding errors in a system or algorithm, which has gained popularity in recent years. The approach requires the construction of a model of the system and the definition of its correctness conditions, i.e., its specification. The goal of the model checking tool is to answer the question, "Is there a trace/behaviour of the system that violates its specification?". This is answered by an automatic procedure that searches the states in which the system can be, denoted as its *state-space*, and verifying whether or not the given specification holds in each state. If there is a violation, called a *counterexample*, a trace to it is reported to the user. If no counterexample is found, the specification is said to *hold* for the model of this system. Unfortunately, this question is undecidable in general, since a system may have an infinite number of traces. Therefore in practice, the model checker places bounds on the parameters of the system being checked. Often, most, if not all, behaviours of the

full system are checked within these limits, and so model checking can be viewed as a sufficient verification technique; although safety critical systems may also use theorem proving to ensure its correctness. Note that model checking can only show that a specification holds for a model; if either is incorrectly defined, no guarantee can be made for the behaviour of the real system.

Model checking has several advantages over *simulation and testing* and *theorem proving*. Simulation and testing requires the construction, usually manual, of test cases designed to cover certain behaviours of a system – not necessarily all of them; errors could be missed. Model checking, on the other hand, only requires the formal description of the system and some specification, before automatic checking performs a much larger coverage. Theorem proving is a technique where one proves that some conjecture is a logical consequence of a set of axioms and deductive rules, for a system. In comparison to theorem proving, model checking provides less guarantee an error does not exist, since it checks a bounded model of the system. Although, it has the advantage that when an error is found, a trace to it can be presented to the user. Also, far less effort is required from the user, who need not be an experienced practitioner. For example, an interactive theorem proving environment, e.g., Atelier-B [Ste, 1996] or PVS [Owre et al., 1992], requires the user to guide inferences made by the prover, or determine intermediate lemmas, in order to prove some conjecture; whereas model checking only requires the user to specify the model and the specification, before automatic checking can take place.

The correctness conditions analysed during model checking are typically represented as a temporal logic formula, denoted ϕ. They are checked over the set of behaviours of a concurrent system, its *model*, and it is usual to represent this system as a labelled state transition system, called a *Kripke structure*, which is now introduced (as defined in Miller et al. [2006]).

Let V denote the variables of a system and, for each $v \in V$, let $D(v)$ be the domain of v (its possible values). The set of atomic propositions over V is then:

$$AP = \{(v = value) \mid v \in V \text{ and } value \in D(v)\}.$$

Given a set of variables V, a formal definition of a Kripke structure in terms of AP is presented in Definition 1.1:

Definition 1.1. A Kripke structure $\mathcal{M}$ over the set of atomic propositions AP is a four-tuple $\mathcal{M} = (S, S_0, R, L)$ where:

- S is a non-empty, finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a total transition relation, defining the 'steps' of system behaviour from state s to s' such that $s \mapsto s' \in R$, and

- $L : S \rightarrow 2^{AP}$ labels each state with the atomic propositions true in that state.

Given a Kripke structure $\mathcal{M}$, a *path* is defined as an infinite sequence of states $\pi = s_0, s_1, \ldots$ where $s_0 \in S_0$ and $\forall\, i > 0$, $s_{i-1} \mapsto s_i \in R$. A transition sequence is an infinite sequence of transitions. Model checking analyses whether the temporal logic formulas hold for the paths/states of a Kripke structure of a system.

Generally, the temporal logics used is either *CTL** or one of its sublogics; *CTL* (computation tree logic) [Ben-Ari et al., 1983, Clarke and Emerson, 1981], or *LTL* (linear temporal logic) [Pnueli, 1981]. CTL* is defined as a set of state formulas, which may include path formulas. Two quantifiers exist, A and E, denoting *for all* paths or *for some* path respectively. Furthermore, there are five basic temporal operators: X (next time), U (until), R (release – the dual of until), F (eventually) and G (always). In the CTL sublogic, these temporal operators must be immediately preceded by some path quantifier A or E. In LTL, formulas are constrained to be of the form Ap, where p is a path formula in which the only state subformulas permitted are atomic propositions. It is usual to elide the A in an LTL formula, thus the previous path formula would become, p. For a thorough reference to the precise syntax and semantics of CTL* and its sublogics, see [Clarke et al., 1999, Chapter 3].

Regarding the construction of a Kripke structure, concurrent systems are typically defined using a modelling language such as PROMELA (*PR*ocess *ME*ta *LA*nguage) [Holzmann, 1997b], the SMV (*S*ymbolic *M*odel *V*erifier) language [McMillan, 1993], process algebras such as PBC (Petri Box Calculus) [Best and Koutny, 1995], rule-based languages such as Murϕ [Dill et al., 1992], or the formal language of B [Abrial, 1996]. Using this description, most model checkers can induce automatically the Kripke structures (or equivalent) representing the system behaviour.

1.2 Two-Process Mutual Exclusion Problem in Promela

This section presents a two-process mutual exclusion problem to highlight how model checking can be used in practice. Let us consider two processes that require the reading/writing of shared memory, in addition to the condition that this memory can only be accessed one process at a time.

A solution to the problem requires each process, P_i, where $i \in \{0, 1\}$, to only ever be in one of three distinct states: i.) the *non-critical* state, denoted N_i, where the process does not require the critical resources, ii.) the *trying* state, denoted T_i, where the process has made a request for the critical resources (which it currently does not possess), and iii.), the *critical* state, denoted C_i, where the request has been fulfilled and the process is currently accessing the critical resources. In addition, a process must be holding a token to enter the critical state, i.e., whenever a variable `turn` matches the process number.

The behaviour of the system is depicted using a Kripke structure in Figure 1.1 (as it appears in [Miller et al., 2006, Figure 1]). The system has two start states, i.e., where both processes are in the non-critical region, N_0, N_1.

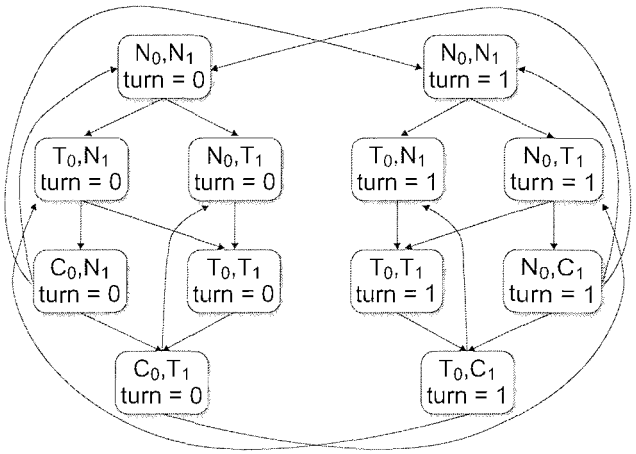


FIGURE 1.1: Kripke structure for the two-process mutual exclusion problem

Peterson's algorithm [Peterson, 1981] provides an implementation of a solution to the problem, which uses a blocking scheme, where a process waits for the critical region to become free (indicated by a certain condition) before entering. Figure 1.2 describes the algorithm using PROMELA – the input language to the SPIN model checker. The encoding comes from the Spin distribution, Spin 4.3.0 [2007].

The PROMELA model makes use of three shared variables. `turn` and `flag[2]` are used to control process blocking, and `ncrit` records the total number of processes in the critical region. Two processes are declared (`active [2]`), whose behaviour is given in

```
bool turn;
bool flag[2];
byte ncrit;

active [2] proctype user()
{
again:
    flag[_pid] = 1;                                  /* Trying */
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);       /* Blocked */

    ncrit++;
    skip;                                            /* Critical */
    ncrit--;

    flag[_pid] = 0;                                  /* Non-critical */
    goto again
}
```

FIGURE 1.2: A PROMELA model for Peterson's mutual-exclusion algorithm

the body of process type, `user()`. `_pid` is a special variable in PROMELA used as a process identifier. The three states of the system (non-critical, trying and critical) are indicated by comments in the code. Note that a process blocks at the location indicated, until the disjunction (`||`) evaluates to `true`.

Correctness conditions of the system may be specified and subsequently verified using SPIN. Therefore, one can verify by model checking the mutual exclusion requirement, "Can we guarantee that two processes are never inside the critical region at the same time?". This can be achieved by specifying the linear time logic formula given below, i.e., "For all computation paths, and all states along them, the value of `ncrit` is at most 1".

```
AG (ncrit <= 1).
```

1.3 State-Space Explosion and Combative Methods

Model checking is made difficult by the state-space explosion problem. This is where, as a formal specification of a system grows linearly in size, its state-space suffers combinatorial growth, quickly becoming too large to feasibly check. Take for example, a system with n variables, each having x possible values. Then, there are possibly x^n states to check. If a new variable with x possibilities is added, the number of states to check increases to x^{n+1}, i.e., an exponential growth.

Much research in model-checking investigates ways to tackle the state-space explosion problem. A review is given of five popular strategies:

1.3.1 Partial Order Reduction

The first method, called *partial order reduction* [Emerson and Sistla, 1997] exploits the independence of concurrent processes. Take for example, two concurrent, independent processes that run from start to finish, each with a set of transitions that preserve the truth value of the property being checked (*invisible* transitions). For checking purposes, only one trace of execution of the composed system must be verified for correctness. However, the standard model checking algorithm analyses every possible trace, and consequently performs a large amount of redundant checking. Partial order reduction aims to exploit such cases, and prevent redundant searches. In general, for each state encountered, such techniques identify an appropriate subset of the enabled transitions to consider, i.e., independent and invisible transitions. Thus, checking explores only a constrained search space, which requires less memory and time for verification.

Note that for certain systems, partial order reduction cannot establish savings in the cost of verification. For example, when no independent transition exists. However, in many cases the technique can be very effective.

One successful approach to partial order reduction is that of the *ample sets* method [Peled, 1997], which has been successfully implemented into the SPIN model checker [Holzmann, 1997b]. Other successful approaches include the *sleep sets* and *persistent sets* methods [Godefroid, 1996], which are applied to the VeriSoft tool [Godefroid, 1997]. In more recent work, [Bhattacharya et al., 2005] presents a reduction approach for rule-based languages such as Murϕ, based on the observation that the necessary independence conditions required for partial order reduction, for bounded systems, can be encoded using boolean propositions and checked using methods for determining their truth values (e.g., using SAT solvers), and [Pradubsuwun et al., 2004] investigates and describes a method for partial order reduction algorithm for timed circuit-based systems.

1.3.2 Symbolic Model Checking

In *symbolic model checking*, one combats the state space explosion problem by reducing the amount of physical memory required to store states and transitions, so that larger systems can be verified. This is achieved through the use of *symbolic* representations of states and transitions, instead of *explicit* representations.

McMillan [1993] presents effective methods for verifying CTL properties of very large hardware systems through the use of the SMV model checker; which is perhaps, currently, the most successful symbolic model checker. There exists a range of other such tools, such as the BEBOP model checker of Ball and Rajamani [2000], which verifies embedded software systems encoded as boolean programs. It is usual for symbolic methods to represent a search space using binary decision diagrams [Bryant, 1986] (BDDs), and indeed, this is the case with SMV and BEBOP. [Hartonas-Garmhausen et al., 1999] integrates symbolic model checking with the notion of probability, to produce the ProbVerus model checker, which allows one to check whether CTL properties hold for some acceptable probability. More recently, [Gunter and Peled, 2003] investigates the application of symbolic model checking to component based verification of software, and [Kahlon et al., 2006] combines symbolic methods with partial order reduction, and transforming the system to be checked into a circuit-based model, which can have LTL properties verified by either a SAT/BDD-based model checker.

1.3.3 On-the-fly Model Checking

On-the-fly model checking is another strategy to reduce the amount of memory required during verification. The premise is that the verification of a property does not always

require the construction of the entire state space. For example, should a counterexample be present, model checking need only generate the part of the state space that leads to the counterexample. Therefore, this method generates sections of the state space only when required.

On-the-fly model checking has been combined with symbolic model checkers with some success [Bouajjani et al., 1997, Ben-David et al., 2003], although it is more commonly used by tools with explicit state/transition representations, since this simplifies the application of state space traversal methods such as depth/breadth first traversal [Vardi and Wolper, 1986, Leuschel and Butler, 2003].

1.3.4 Abstraction

Abstraction is another strategy for reducing the state explosion problem. The general idea is to construct a simplified representation of a system to simplify verification. The abstracted system may not satisfy the same set of properties that hold for the original system, however soundness is usually required: the set of properties that prove for the abstraction also hold for the original system. One common abstraction technique is to analyse a system description, and essentially eliminate those variables that are not referred to by the specification. Consequently, the checked properties are preserved, but the size of the model is reduced. This is called *cone of influence* abstraction. In a second technique, called *data abstraction*, one determines a relationship between system data values and an abstract set of data. By extending this relationship to states and transitions, one may generate an abstraction that is smaller in size, yet sound with respect to properties of the original system. An introduction to both techniques can be found in Clarke et al. [1999].

1.3.5 Symmetry Reduction

Symmetries in systems often arise due to the presence of replicated structures. For example, they can be found in the architecture of certain distributed database systems or within particular data structures exhibiting geometric symmetry. When model checking such systems, exploration must take place over a state space containing replicated (equivalent) regions. *Symmetry reduction* is a technique whose aim is to proactively refuse making such redundant searches. The consequence is that one explores only a constrained state space, the *quotient model*, which should require less memory and time to cover, and therefore, larger systems may be verified.

In a naïve approach to symmetry reduction, one may construct the model of a system, and subsequently search for symmetries. However, there would be little benefit since the full model still must be generated. Therefore, a symmetry reduction strategy generally

finds symmetries of a system without constructing its full model. It is then the identification of symmetries that presents the key challenge for methods of symmetry reduction, especially since the problem is related closely to the graph isomorphism problem, for which there is no known polynomial time algorithm [McKay, 1981].

To date, there are a range of successful strategies for symmetry reduction including the Murϕ verifier [Dill et al., 1992], which introduces a special data type, called a *scalarset* [Ip and Dill, 1993], to indicate symmetries; SMC, the Symmetry based Model Checker [Sistla et al., 2000], which can perform verification of both safety and liveness conditions; the Bogor tool-set, for symmetry reductions when model checking Java programs [Robby et al., 2003]; and the integration of symmetry reduction with symbolic methods in [Emerson and Wahl, 2005], which shows how model checking can exploit a range of symmetry groups. More recently, [Miller et al., 2007] investigates the relationship between symmetry reduction and inductive reasoning with respect to model checking networks of components, which may share symmetric properties. An approach is proposed for developing PROMELA specifications without affecting the natural symmetry of a system, and a method is presented that analyses the process communication structure and system architecture to identify symmetries, which are then exploited during model checking.

The techniques developed in this thesis concern model checking in the formal language of B. In the next section, the B language and its methodology are introduced.

1.4 The B-Method

The B-Method [Abrial, 1996] is a theory and methodology used for the formal specification and development of computer software systems. It includes a concise language, called B, used to describe state-based systems and is suitably abstract for simple systems, yet expressive enough for large safety critical systems. It has been used with notable success in the Météor project for controlling train traffic [Behm et al., 1999], and examining the PCI Protocol [Cansell et al., 2002] and the IEEE 1394 Tree Identify Protocol [Abrial et al., 2003].

It is based on an abstract modelling framework called *abstract machine notation*, which enables systems to be structured in a modular style, similar to the object-oriented approach of some programming languages, allowing larger components to be constructed from a set of smaller components.

The B notation tackles extra complexity introduced by low level programming statements by defining a high level language that forces the designer to use only well understood, concise statements. Systems specified in B are therefore relatively simple. The effect is to shift attention to the *design* of the system.

The notation is based on Zermelo-Fraenkel set theory, with the axiom of choice. The structure of a typical B system (machine) consists of three main parts:

i.) system *data*, using values from sets. This may include constants, which are named in a *CONSTANTS* clause. Constraints on sets and constants are defined using predicate logic in a *PROPERTIES* clause.

ii.) system *state* is defined through variables, given in the *VARIABLES* clause, and an *INVARIANT* clause, which is a safety property expressed using set theory and predicate logic over the variables of the system. For example, the invariant defines the typing information of variables.

iii.) system *behaviour* is defined through operations and their actions (e.g., assignments) in the *INITIALISATION* or *OPERATIONS* clauses. These enable deterministic or nondeterministic assignments to be made on state variables. Operations in B may have input and/or output parameters.

A B specification is also accompanied with a set of mathematical proofs justifying any necessary parts, such as the typing of variables and preservation of the invariant by operations. These proofs also serve to convince the system specifier that the B system is valid.

The concept of refinement [Back, 1981, Abrial, 1996] is the key notion for developing B specifications of hardware/software systems in an incremental manner. The idea is to first define a very abstract specification of the system being developed. Details are progressively added to this through the construction of a sequence, or *chain*, of more concrete specifications. The relationship between neighbouring specifications in this chain is that of refinement. For each such relationship, this means the more concrete refinement machine preserves already proved system properties of its parent machine.

The underlying theory of the B-Method requires operations to satisfy some precondition before any actions take place. However, this is not enforced. Should the caller wish, an operation may be executed outside of its precondition. In such cases, future behaviour of the system is undefined, and one cannot guarantee the preservation of any invariant conditions.

Note that in contrast to the languages checked by existing model checkers such as PROMELA, the B language *does not* include temporal operators. Moreover, the correctness conditions of a B machine is an *invariant*: a set of *safety* conditions expressed using set theory and predicate logic. Instead, it is possible for a notion of liveness to be defined through the refinement of B machines.

A simple example of a B machine of a phonebook is presented in Figure 1.3, which defines four operations that allow one to *add*, *delete*, *lookup*, and request the number of entries in the phonebook (*size_phonebook*).

MACHINE *phonebook*
SETS *Name*; *Code*
VARIABLES *db*, *active*, *activec*
INVARIANT $db \in Name \pfun Code \land$
$active \in \mathbb{P}(Name) \land$
$activec \in \mathbb{P}(Code) \land$
$\mathrm{dom}(db) = active \land$
$\mathrm{ran}(db) = activec$
INITIALISATION $db := \varnothing \parallel active := \varnothing \parallel activec := \varnothing$
OPERATIONS
add(n , c) $\widehat{=}$
PRE
$n \in Name \land c \in Code \land n \notin active$
THEN
$db := db \cup \{n \mapsto c\} \parallel$
$active := active \cup \{n\} \parallel$
$activec := activec \cup \{c\}$
END;

delete(n , c) $\widehat{=}$
PRE
$n \in Name \land c \in Code \land n \mapsto c \in db$
THEN
$db := db \setminus \{n \mapsto c\} \parallel$
$active := active \setminus \{n\} \parallel$
$activec := db[(active \setminus \{n\})]$
END;

$c \leftarrow$ lookup(n) $\widehat{=}$
PRE
$n \in Name \land n \in active$
THEN
$c := db(n)$
END

$s \leftarrow$ size_phonebook $\widehat{=}$
BEGIN
$s := \mathrm{card}(db)$
END
END

FIGURE 1.3: Example B Machine of a Phonebook

A brief explanation is given about the main parts of the *phonebook* machine:

- *MACHINE:* The name of the machine, in this case *phonebook*.
- *SETS:* The sets used in this machine. Names in the phonebook are elements of *Name*, and phone numbers are elements of *Code*.
- *VARIABLES:* The variables required to define the behaviour of this machine. In this case, *db* is the database, or phonebook, that maps *Name*s to *Code*s – modelled as a partial function, denoted by the symbol, $\nrightarrow$. Therefore, each *Name* can have at most one *Code*. *active* is the set of names in the phonebook, and *activec* is the set of numbers in it.
- *INVARIANT:* Contains the system invariant, i.e., a conjunction of conditions that must hold in every state of the machine. It defines the variables' typing, in addition to any extra conditions required. In this machine, *db* is defined as a partial function from *Name*s to *Code*s. The variable, *active* is a subset of *Name*. Since it contains the names in the phonebook, it can be specified as the domain of the phonebook, using dom(*db*). Finally, *activec* is a subset of *Code*s, the range of the phonebook, denoted ran(*db*). Note that we have expressed subsets as elements of powersets, e.g., $active \in \mathbb{P}(Name)$; equivalently, we could use the subset symbol, such that $active \subseteq Name$.
- *INITIALISATION:* Defines an initialisation of the variables that satisfies the invariant. Note that the operator, ||, indicates the parallel evaluation of the three assignment expressions.
- *OPERATIONS:* Defines the system behaviour, through simple statements on the variables. Application of an operation should guarantee the invariant is preserved. This machine has four operations. Firstly, the operation, *lookup*, takes a *Name* as parameter and returns (indicated by $\leftarrow$) the corresponding *Code*. The *add* operation adds entries into the phonebook, specified as *maplets* (ordered pairs) such as $n \mapsto c$, where n is a *Name* : and c is a *Code*. Conversely, the *delete* operation deletes entries. Note that *add* and *delete* make use of set union ($\cup$), set restriction ($\backslash$), and relational image ($db[(active \backslash \{n\})] = \{t \mid s \mapsto t \in db \wedge (active \backslash \{n\})\}$). Finally, *size_phonebook* enables one to query the current number of entries in the phonebook, i.e., its cardinality, denoted card(*db*).

We now consider a modification to the phonebook machine given in Figure 1.3, as an example of an error in a machine. The change involves weakening the precondition of the *add* operation as follows (the rest of the operation remains unchanged):

$$
\begin{array}{l}
\text{add}(\ n\ ,\ c\ ) \mathrel{\widehat{=}} \\
\quad \textbf{PRE} \\
\quad\quad n \in Name \wedge c \in Code \\
\quad \textbf{THEN} \ldots
\end{array}
$$

The *add* operation is now defined over any $n \in Name$ and $c \in Code$: there is no requirement that $n \notin active$. Therefore, it is possible to *add* the same name with two different numbers into the phonebook, *db*. This violates the invariant condition that *db* is a (partial) function. Such errors can be easy to make during the development phase of a machine, however, the machine invariant makes it possible for the presence of these errors to be established, and subseqently resolved. The other type of error in a B machine concerns refinement. Before presenting a refinement error, we first provide a more detailed introduction to refinement in B.

1.4.1 Introducing Refinement in B

Refinement in B involves moving from abstract specifications to more concrete specifications, through a transformation that enables the refining system to simulate the abstract system. Moreover, a refinement machine must guarantee to provide the expectations a user may have about the abstract machine. Given successive refinements of an abstract machine, one can reach concrete systems that conform to their abstract specification, and from which executable system implementations, e.g., C programs, may be derived.

A requirement of refinement is that refinement machines must use exactly the same interface as their more abstract parent machine. This includes the same operations, with the same input and output parameters, i.e., the same *signatures*. However, one may introduce new data (e.g., variables or constants) into the more concrete specification, providing that there exist expressions relating data in the refinement machine to data in the abstract machine. Such expressions are called *gluing invariants*.

We present the concept of refinement more formally using an adaptation of the description given in [Leuschel and Butler, 2005], which makes use of the notion of forward simulation [He et al., 1986]. For a reference to refinement in B in terms of guarded commands, see [Abrial, 1996, Part 4]. Let m represent an abstract machine, and let mR represent a specification that refines m. m and mR are comprised of distinct sets of variables that are related via the gluing invariant, such that abstract states relate to concrete states. This correspondence must be satisfied for the initial states of both systems, in addition to the states reached via the application of operations of the systems. Given m and mR, the operations are denoted mOp and $mROp$ respectively; the sets of initial states are denoted mI and mRI respectively; and R is the gluing invariant between m and mR:

1. For every initial concrete state there is a related initial abstract state:

$$sR \in mRI \Rightarrow \exists \cdot s \in mI \wedge sR \mapsto s \in R$$

2. If state sR in mR is related to s in m, then for every execution of $opR \in mROp$ from sR to sR', written $sR \xrightarrow{opR} sR'$, a corresponding execution $s \xrightarrow{op} s'$ exists, where $op \in mOp$ and sR' is related to s':

$$sR \xrightarrow{opR} sR' \wedge sR \mapsto s \in R \Rightarrow \\ \exists \cdot s \xrightarrow{op} s' \wedge s' \mapsto sR' \in R$$

Figure 1.4, presents *phonebookR* as an example refinement of the *phonebook* machine. In this example, one variable called, *sz*, is introduced to store the current size of the phonebook. Hence, there exists the gluing invariant, $\text{card}(db) = sz$, meaning the cardinality of *db* equates to the value of *sz*. Now, by using *sz*, the *size_phonebook* operation need only return *sz* each time a request is made. This is an improvement upon the method used in the abstract machine, which evaluates $\text{card}(db)$, on every request. Accordingly, *sz* is either incremented or decremented each time one *adds* or *deletes* entries respectively.

Also, notice the refinement needs only retain the *db* variable from the abstract specification. The variables *active* and *activec* are discarded since their values can be inferred from either the domain or range of *db* respectively.

We now consider a change to *phonebookR*, which makes it an invalid refinement of *phonebook*. The change is similar to the one introduced into the *phonebook* machine, to provide an example of an invariant violation. On this occasion we weaken the guard of the *delete* operation, as follows:

$$\text{delete}(\ n\ ,\ c\ ) \mathrel{\widehat{=}} \\ \quad \textbf{IF}\ n \in Name \wedge c \in Code \\ \quad \textbf{THEN}\ldots$$

The error is caused by the possibility of the *delete* operation engaging at times other than those specified in the abstract specification of *delete*. That is, the modified *delete* is enabled for any $n \in Name$ and $c \in Code$: there is no longer the restriction of $n \mapsto c \in db$. Therefore, this modified *phonebookR* machine is not a valid refinement of *phonebook*. Given that the body of *delete* remains the same, we can also see that the change does not violate the invariant condition specifying that *db* is a partial function, but does violate the invariant condition, $\text{card}(db) = sz$.

1.5 Formal Verification in B

Formal verification of B specifications can be split into two activities; *consistency checking*, used to show the operations of a machine preserve the invariant, and *refinement checking*, used to show one abstract machine is a valid refinement of another. These activities give rise to a number of *proof obligations*, which guarantee the correctness of

REFINEMENT *phonebookR*
REFINES *phonebook*
VARIABLES db, sz
INVARIANT
$sz \in \mathbb{N} \wedge$
$\text{card}(db) = sz$
INITIALISATION $db := \varnothing \;||\; sz := 0$
OPERATIONS
add(n , c) $\widehat{=}$
IF $n \in Name \setminus \text{dom}(db) \wedge c \in Code$
THEN
$db := db \cup \{n \mapsto c\} \;||$
$sz := sz + 1$
END;

delete(n , c) $\widehat{=}$
IF $n \in Name \wedge c \in Code \wedge n \mapsto c \in db$
THEN
$db := db - \{n \mapsto c\} \;||$
$sz := sz - 1$
END;

$c \leftarrow$ lookup(n) $\widehat{=} \ldots$

$s \leftarrow$ size_phonebook $\widehat{=}$
$s := sz$
END

FIGURE 1.4: A Valid Refinement of the Phonebook Machine

a system description. Such proof obligations are usually discharged using an interactive theorem prover, such as Atelier-B [Ste, 1996] or the B-Toolkit [B-C, 1999].

In on going research, the RODIN project (Rigorous Open Development Environment for Complex Systems) [RODIN], is developing a suite of tools supporting complex system development and verification using the next generation of the B-Method, called Event-B [Abrial and Mussat, 2001, Métayer et al., 2005]. The fundamental difference concerns the method of transitioning between states. That is, operations are replaced by the notion of *events*. When an operation is called in classical B, the caller must ensure its precondition is satisfied, otherwise any state can be reached and the invariant is not ensured. In contrast, an event has a guard and can only engage when the state of the system satisfies the guard. It follows that events can either be observed or not observed, and therefore, proof activities eliminate the potential for erroneous executions of events. In addition, events do not include input or output parameters, unlike operations. Further changes in Event-B include promoting decomposition and refinement of specifications, in an attempt to simplify the sharing of data between specifications. For a thorough

introduction to the differences between classical B and Event-B, see Cansell and Méry [2006].

The deliverables of the RODIN project also include a collection of reusable development templates generated from case studies and a set of guidelines for the rigorous development of complex systems.

1.5.1 Model Checking in B

Currently, only one tool exists for model checking B specifications. This is called ProB [Leuschel and Butler, 2003], and it has been developed, using SICStus Prolog [SICStus]. The tool also enables B specifications to be animated (driven by the user through traces of execution), allowing users to gain confidence in their specifications, and unlike the animator provided by the B-Toolkit, the user need not guess the right values for the operation arguments or choice variables. In its original releases, ProB does not use any strategy in Section 1.3 to combat state space explosion, although it uses a simple *normalisation* scheme to ensure there exists only one representation of each state; so states $s_1 = \langle\{a, b\}\rangle$ and $s_2 = \langle\{b, a\}\rangle$ have the same normal form.

Model checking in ProB makes use of user defined bounds on system types (e.g., maximum integer is 15, minimum integer is -10) to ensure algorithmic termination. The procedure performs an exhaustive search of all states reachable from the initialised state via successive applications of machine operations. Thus, the model is constructed incrementally. The procedure makes use of three key variables, *Queue*, *Visited* and *SGraph*, and is formalised in Algorithm 1. Note that the (safety) properties that can be defined for a system constitute an invariant, denoted ϕ in Algorithm 1. Therefore, model checking in ProB corresponds to invariant checking. Also note that there is only one initial state, denoted *root*, which represents the uninitialised machine.

The variable *Queue*, stores the states yet to be explored and checked, and $Visited$ records states already reached by checking. *SGraph* stores the section of the model explored so far. Model checking proceeds by successively removing the state at the head of *Queue*, checking it (line 4), and (if it is not an error state) then adding any of its successor states to *Queue* not yet encountered. Checking a state s constitutes analysing whether the invariant (set of safety conditions) ϕ holds for the particular values of variables in s (written $s \models \phi$, if this is true), or whether s is deadlocked and so has no enabled operations. If checking encounters such an error state, a trace to the error is returned to the user[1]. The use of the queue, *Queue*, gives control to the type of search that takes place. Placing a new state at the front of it results in a depth-first search, whereas placing it at the back results in a breadth-first search. A combination of these strategies are used

[1]By default, ProB checks for invariant violations and deadlocks. However, user options include controls over whether these conditions are checked.

Algorithm 1 Model Checking in ProB

Require: An abstract machine M and invariant ϕ

1: $Queue := \langle root \rangle$; $Visited := \{root\}$; $SGraph := \{root\}$;
2: **while** $Queue \neq \langle\rangle$ **do**
3: state := $pop(Queue)$;
4: **if** $state \not\models \phi$ or $state$ deadlocks **then**
5: **return** counterexample trace in $SGraph$ from $root$ to $state$
6: **else**
7: **for all** successor $succ$, and operation Op **such that** $state \rightarrow^{M}_{Op} succ$ **do**
8: $SGraph := SGraph \cup \{state \rightarrow_{Op} succ\}$
9: **if** $succ \notin Visited$ **then**
10: **if** random(1) $< \alpha$ **then**
11: add $succ$ to front of $Queue$
12: **else**
13: add $succ$ to end of $Queue$
14: **end if**
15: $Visited := Visited \cup \{succ\}$
16: **end if**
17: **end for**
18: **end if**
19: **end while**
20: **return** ok

during model checking. The choice of strategy is made randomly, whenever a *random* value is less/greater than a user defined value, α (line 10). In practice, this heuristic has proven to be very effective. The breadth-first search is shown to be especially effective at identifying systematic errors in operations, which lead to errors in most states; depth-first search is shown to be effective at finding errors that occur less frequently in a state space [Leuchel and Butler, 2006].

Note that all elements of $Queue$ and $Visited$ have associated hash values. It is therefore usually quite efficient to decide whether $succ \notin Visited$.

ProB also provides a graphical visualisation of certain aspects of the reached state space of a B machine. Visualisation uses the graph drawing package called, *dot* [DOT], which analyses the search space recorded in *SGraph*, in Algorithm 1 (refer to line 8). This alternate form of detailing the model of a B specification can be beneficial to the user, especially since the human mind is adept at visual perception [Ham et al., 2001], such as identifying cycles/symmetries. Original visualisations that ProB offers include the reached state space, the current state, and the shortest trace to the current state. The latter is particularly useful when analysing counterexamples.

Figure 1.5 depicts part of the state space that can be displayed in ProB when model checking the *phonebook* machine in Figure 1.3. The state space is represented as a labelled transition system where system states are denoted by nodes, and operations between states are denoted by labelled edges between nodes. For this example, the

Name and *Code* sets have a cardinality of 2 (specified using an options menu in ProB). Consequently, ProB generates 2 abstract elements for each set for use during animation and checking. So, for the *Name* set ProB generates *Name1* and *Name2*. The current state of the system in Figure 1.5 is indicated by a green octagonal node: the initialised machine. Hence, *db*, *active* and *activec* are all equal to the empty set. The green triangle represents the uninitialised machine, and the red elliptical nodes depict successors of the current state yet to be explored. Note that ProB also has an option to turn on/off self loops in such visualisations. This option is off for Figure 1.3; if it had been on, each state would have a self loop representing the *lookup* and *size_phonebook* operations (which preserves the state of the system).

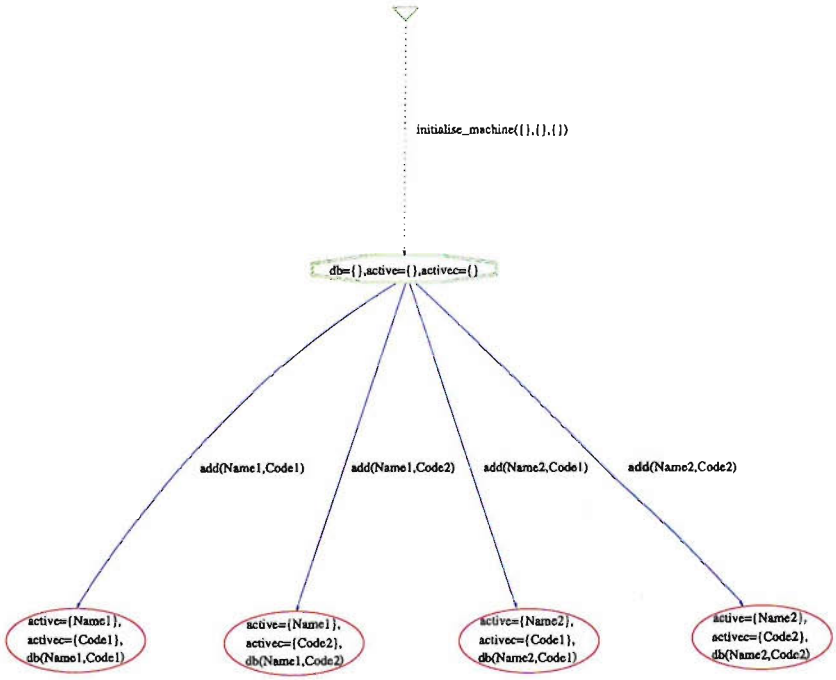


FIGURE 1.5: Part of the state space for the phonebook machine shown in ProB

ProB can also perform *constraint-based checking*, where a state of the system is found from which the application of a single machine operation leads to a state violating the invariant. This is essentially a constraint satisfaction problem, for which there exist mature programming packages that are accessible from ProB (e.g., CLP(FD) in SICStus Prolog). The advantage is that one can find errors of a B system without traversing large state spaces. If an error is found, then it is not possible to prove the B machine correct using the B proof rules. The disadvantage, however, is that a counterexample may not be reachable from a valid initial state; in which case it may be discarded.

In related work, Legeard et al. [2002] present the BZ testing tool (BZTT). In contrast to ProB, this tool does not verify B specifications but instead automatically generates a set of test cases. In addition, it provides an animator for the specification, which

maintains a set of constraints about the variables of the machine, rather than explicitly enumerating possible values as with ProB.

1.6 Contributions of this Thesis

This thesis presents a set of techniques that have been developed to address several of the problems associated with model checking systems in B, which have been introduced in the preceding parts of this chapter. The details of the precise contributions are listed below.

i.) The goal of a model checker is to verify for a finite state model of the system whether all states satisfy some set of correctness conditions. If this is not true, a counterexample should be generated. Some tools provide certain visual feedback to aid the user, such as process automata in Spin. However, there is typically little emphasis on visual feedback. Therefore, this thesis develops a set of techniques that provide useful *visual feedback* on the model of the system, which help the user gain an improved understanding of the system being checked. The premise is that the human mind is adept at processing visual feedback, such as identifying cycles or symmetries, which could be very useful to system designers – regardless of how much experience they have. As a basic example, a Spin user would gain a better understanding of the system in Figure 1.2 if a Kripke structure resembling that shown in Figure 1.1 could be automatically presented.

ii.) During the development of a symmetry reduction strategy, symmetric regions of the state space of a system need to be identified. This task is closely related to the graph isomorphism problem, for which there already exist established programs, such as *nauty* [McKay, 1981]. This thesis documents an extension to the underlying algorithm of nauty that makes it suitable for identifying symmetries in B systems.

iii.) This thesis presents the algorithms that constitute the first strategy for symmetry reduced model checking of systems in B, called symmetry reduction via *canonical labels*, and their integration into the ProB tool-set. The improvements in the tool enable the verification of a larger range of B systems, including more complicated systems, within practical time and memory constraints.

iv.) A detailed evaluation is presented documenting the effectiveness of the symmetry reduction technique developed for B. This includes analytical details that provide insights into ways to improve its performance, in addition to directions for future work.

v.) Symmetry reduction in B via canonical labels has also stimulated works developing three new techniques for symmetry reduction in B. These include the separate work

of *symmetry markers* [Leuschel and Massart, 2007], and the collaborative works of *permutation flooding* [Leuschel et al., 2007] and *canonical labels + symmetry markers*. The techniques are subjected to experimentation to establish the details of their performance, in comparison to the canonical label approach.

vi.) The final contribution of this thesis is a proof of soundness for the symmetry reduction method developed, with respect to standard model checking in ProB. The proof is achieved using refinement in B. Firstly, an abstract B machine is given specifying the goal of model checking. From this, separate chains of refinement are specified for the standard model checking algorithm, and the reduced model checking algorithm. Background research suggests this is novel in the field of B. Furthermore, the use of B and refinement provides an alternate reference for the concepts used during model checking and symmetry reduction.

For the aid of the reader, the chapters of this thesis are organised as follows. Chapter 2 gives a background to the graph isomorphism problem, in addition to an algorithm that can determine whether two graphs are isomorphic. This will be necessary in Chapter 4 later, in which a technique for identifying symmetric B states is described. Chapter 2 also provides a background to symmetry reduction, and a review of several model checkers that use symmetry reduction. The chapters presented next constitute the research contributions of this thesis. Initially, research focussed on improving the visualisation of state spaces via reductions in their size. The methods developed are given in Chapter 3. This work provided the inspiration for the next line of research, where the aim is to develop the first methods for symmetry reduction in B, so that larger B specifications can be model checked. The culminations of this work, presented in Chapter 4, are two techniques for symmetry reduction in B that use a method for identifying isomorphic graphs. Chapter 5 presents an empirical evaluation of the methods for symmetry reduction in ProB and illustrates their effectiveness. Chapter 6 formalises the algorithms of standard model checking and symmetry reduced model checking in B, and shows the correctness of the reduced approach. Finally, Chapter 7 provides conclusions for the research presented in this thesis, and gives directions for future work.

Chapter 2

Graph Isomorphism and Symmetry Reduction

2.1 Introduction

A major component of the B Method's expressivity is the relation, i.e., powersets of ordered pairs. Subsets of relations define different types of frequently used functions, such as partial/total functions, injections, bijections etc. In fact, it is shown later in Chapter 4 that relations can be used to encode any state in B. If it can be detected that two relations are symmetric, it is necessary to store only one; the redundant relation can be discarded. This idea can be used to improve the efficiency of a model checking procedure in terms of memory used and time to find a counterexample, if any, and is the focus of Sections 2.6 – 2.8. First, however, this chapter presents the fundamental ideas behind exploiting such symmetries, which will be used later in Chapter 4 in a technique for symmetry reduced model checking of B systems. Since relations can be represented as directed graphs, this chapter concentrates on finding symmetries between directed graphs.

A naïve approach to detect symmetries between graphs during model checking is to use a function f that can explicitly test two graphs for symmetry on-the-fly (e.g., using an intelligent vertex relabelling scheme). During model checking, relations may constitute any number the state variables; so each relation must first be extracted from the state, and then compared using f with the corresponding relation in every other reached state. This approach would be very wasteful on computing resources since the computations made by f are repeated many times. An improvement is to *label* each relation with some value, such that relations have the same label if and only if they are symmetric. These labels are called *canonical* labels. Then, the test for graph symmetry, and state symmetry in this case, is reduced to a simple test for equality. Techniques that produce

canonical labels for graphs are therefore preferred to other methods, and they are the subject of the first part of this chapter (Sections 2.2 – 2.5).

Section 2.2 introduces the preliminary definitions used throughout this thesis. Sections 2.3 and 2.4 describe the steps used in a successful strategy to compute canonical labels of graphs.

2.2 Preliminaries

First, we shall introduce several concepts of group theory that will be used throughout this thesis. Given a set, G, a group is defined to be G together with a binary operation, $\circ$ acting on G, called the group multiplication. This group, denoted $(G, \circ)$, satisfies certain conditions and is presented formally in Definition 2.1.

Definition 2.1. Given a set G and a binary operation $\circ$, $(G, \circ)$ is a group, where:

1. The multiplication of two elements of G is itself an element of G: G is said to be closed under $\circ$.
2. The multiplication operation is associative, such that for any $g, h, i \in G$, we have $g \circ (h \circ i) = (g \circ h) \circ i$.
3. There exists an identity element, $e \in G$, such that for any $g \in G$, $e \circ g = g \circ e = g$.
4. For each $g \in G$, there exists an inverse element, g^{-1}, where $g \circ g^{-1} = g^{-1} \circ g = e$.

Note that it is usual to omit the binary operation symbol and let G denote the group $(G, \circ)$, when the binary operation is clear from the context. Also, concatenation is often used to denote multiplication. A subset H of G is subgroup of G if H also forms a group under the multiplication operation.

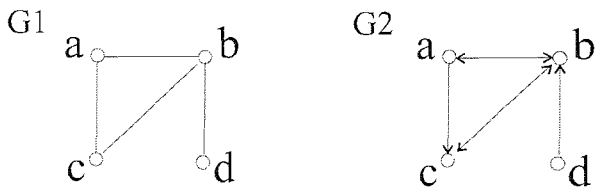
A permutation σ on a finite set A is a bijection over A, denoted $\sigma \in A \rightarrowtail\!\!\!\!\!\twoheadrightarrow A$. The set, $Sym(A)$ is the set of all permutations on A, which forms a group under the composition of functions, and which is sometimes referred to as a *full* symmetric group. A permutation group is defined to be a subgroup of $Sym(A)$.

Having introduced the basic notions used within group theory, we now introduce some key concepts of graph theory, which will also be used throughout this thesis.

The term 'graph', referred to in the previous section, is a node-link diagram where the nodes are called *vertices* and the links are called *edges*. We can distinguish between two types of graphs, *undirected* and *directed* graphs. Preliminary concepts are now given in Definitions 2.2 - 2.8, as found in Gross and Yellen [1999].

Definition 2.2. An *undirected graph* consists of a finite set V of *vertices* and a finite set E of *edges*, such that each edge is a two element subset of vertices. It is customary to write a graph as an ordered pair, (V, E).

Definition 2.3. A *directed graph* is the same as an undirected graph, except that each edge is an ordered pair of vertices - indicating the direction of the edge.



G1 = (V1,E1), G2 = (V2,E2)

FIGURE 2.1: Two graphs: G1 is undirected, G2 is directed.

In Figure 2.1, $G1 = (V1, E1)$, where $V1 = \{a, b, c, d\}$ and $E1 = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}\}$, and $G2 = (V2, E2)$, where $V2 = \{a, b, c, d\}$ and $E2 = \{(a, b), (a, c), (b, a), (b, c), (c, b), (d, b)\}$.

A vertex y is said to be *adjacent* to another vertex x, if x can reach y via some (possibly directed) edge, e.g., c is adjacent to a in both $G1$ and $G2$ in Figure 2.1. An undirected edge $\{i, j\}$ is said to be *incident* with i and j; a directed edge (i, j) is said to be incident from i, and incident to j. The *degree* of a vertex p is the number of edges that are incident with p; for directed graphs, and if there could be confusion, we may refer to the *in-degree* of p as the number of edges incident to p, and the *out-degree* of p as the number of edges incident from p.

Directed graphs naturally model relations; each relational maplet corresponds to a directed edge in the graph. This idea will be used later in Chapter 4. In addition, note that we can represent an undirected graph by a directed graph if every undirected edge $\{u, v\}$ is represented by two directed edges, (u, v) and (v, u). Therefore, for the convenience of this presentation, we refer to directed graphs simply as 'graphs', unless stated otherwise.

A useful representation of a graph is the *adjacency matrix*. Given a list of n graph vertices, an $n \times n$ matrix is constructed whose $[x, y]$ entry is a 1 if $(x, y) \in E$, and 0 otherwise. For example, they can easily be implemented in a C-program with a 2-D array. Table 2.1 shows the adjacency matrix for graph G2 in Figure 2.1.

Note that the vertex ordering used to draw the matrix determines the positions of the 0s and 1s. Two orderings of the vertices of a graph can produce two different matrices, e.g., the matrices corresponding to the orderings (a,b,c,d) and (b,a,c,d) in Table 2.1 are different. However, the graph that they represent remains invariant.

	a	**b**	**c**	**d**
a	0	1	1	0
b	1	0	1	0
c	0	1	0	0
d	0	1	0	0

TABLE 2.1: Adjacency matrix of G2 in Figure 2.1

In many applications, it is useful to know whether two graphs have the same structure, indicating that they may be related in some way, e.g., two graphs are symmetric, therefore the program needs only to store one of them. A structural symmetry is called an *isomorphism*:

Definition 2.4. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a bijection, $f : V_1 \rightarrowtail\!\!\!\!\!\rightarrow V_2$, such that:

$$(f(x), f(y)) \in E_2 \textit{ iff } (x, y) \in E_1$$

The function f is said to be an *isomorphism* between G_1 and G_2.

A special case of an isomorphism, f is when the two graphs are identical, in which case f is a symmetry of the graph, called an *automorphism*. Figure 2.2 shows three graphs. G1 is not isomorphic to any of the other graphs. G2 and G3 are isomorphic, since they have the same structure, regardless of their vertex labels. G1, G2 and G3 all have automorphisms, e.g., (a,b) and (f,g) in G1, (i,j) and (m,n) in G2, (a,b) and (e,f) in G3.

The set of all automorphisms of a graph G is denoted $Aut(G)$, which forms a group under composition of mappings. Given $\pi, \tau \in Aut(G)$, which are permutations (i.e., bijective mappings from the vertices of G onto the same set of vertices), then the result of applying π to a vertex v is denoted v^π. For example, if $\pi = \{(a, b), (b, c)\}$ and $v = b$, then $v^\pi = c$. Permutations are usually composed left to right; $v^{\pi\tau}$ means apply π to v first, and then apply τ to the result. It is usual to say that isomorphic vertices, such as v and v^π where $\pi \in Aut(G)$, belong to the same *orbit*. Furthermore, the *orbit problem* asks whether two vertices belong to the same orbit.

Definition 2.5. Given automorphism group, $Aut(G)$ acting on a set S, then for $s \in S$ the set $\{s^\rho \mid \rho \in Aut(G)\}$ is called the *orbit* of s under $Aut(G)$, and is usually denoted $[s]_{Aut(G)}$, or just $[s]$, when it is clear from the context that it refers to $Aut(G)$.

Finding isomorphisms between graphs is a very well known problem in mathematics, where it is known as the *graph-isomorphism problem*. A polynomial time algorithm is yet to be found that solves it, although many researchers believe that it is not NP-complete [Gross and Yellen, 1999]. However, there is also no proof for this belief.

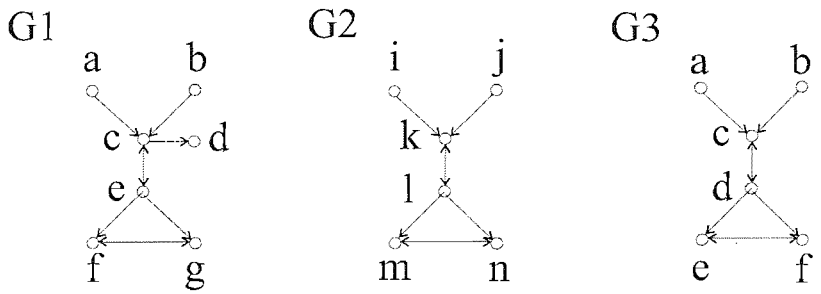


FIGURE 2.2: Three graphs: G1,G2,G3 contain automorphisms, G2,G3 are isomorphic.

Definition 2.6. The *graph-isomorphism problem* is to devise a practical general algorithm that decides whether two graphs are isomorphic.

A step towards finding a strategy to find if two graphs are isomorphic is to compute some property about them that does not depend on how the graphs are presented, called an *invariant*. For example, a list of the degrees of each vertex in a graph is not an invariant because the resulting list depends on the order that vertices are examined. However, if this list is sorted into ascending order, it is an invariant.

Definition 2.7. Let F be a family of graphs, and $G_1, G_2 \in F$. An *invariant* on F is a function ϱ such that $\varrho(G_1) = \varrho(G_2)$ if G_1 is isomorphic to G_2.

If $\varrho(G_1) \neq \varrho(G_2)$, then it is possible to conclude that G_1 and G_2 are non-isomorphic. However, if $\varrho(G_1) = \varrho(G_2)$, no conclusion can be made, e.g., two non-isomorphic graphs can have the same degree sequence. Hence, graph invariants cannot be used directly to solve the graph isomorphism problem. However, they do have the benefit of potentially being computationally inexpensive, e.g., counting vertex degrees and sorting is inexpensive. So, for example, they could be used as a pre-process to a true graph isomorphism algorithm, which can quickly compute whether two graphs are not isomorphic. Indeed, this approach is explored later in Section 2.8.6.

A naïve approach that does solve the graph isomorphism problem may attempt to find an isomorphism function by examining all $n!$ vertex pairs. However this is very inefficient for larger graphs. Another approach is to find an algorithm that labels isomorphic graphs with the same label, called their *canonical* label, and non-isomorphic graphs with different labels. Then two graphs can be tested for isomorphism with an equality test. If a polynomial time algorithm exists for this, then the graph isomorphism problem can be solved in polynomial time. Unfortunately, again, no such algorithm has been found; but, in practice some extremely efficient algorithms exist for most classes of graphs [Kocay, 1996] that may contain several thousands of vertices [Foggia et al., 2001]. A successful method is based on a technique called *partition refinement*.

Definition 2.8. A *canonical* labelling function, *canonical* for a family F of graphs is a function such that for any $G_1, G_2 \in F$, $canonical(G_1) = canonical(G_2)$ if and only if G_1 and G_2 are isomorphic.

The aim of the next two sections is to present to the reader how and why partition refinement can be used as a general graph isomorphism algorithm. Its key issues have been proved by several authors, one of the clearest being that of Kocay [1996]. They are presented again for clarity and completeness.

2.3 Partition Refinement

There is a large amount of literature on graph isomorphism. For an extensive list of relevant work see the bibliographies of Corneil and Kirkpatrick [1980] and Babai and Luks [1983]. Most programs that implement a general purpose graph isomorphism algorithm are based on the method of partition refinement, which is the most efficient method known to date. In the worst case they perform with an exponential complexity, but in practice are extremely efficient for most graphs. Kocay [1996] presents a concise review of the complexity issues associated with this algorithm and others. The most efficient general purpose graph isomorphism program known today is McKay's C-language *nauty*, which is based on partition refinement. Nauty can produce canonical labels for graphs, in addition to their automorphism group. For a thorough reference of the algorithms used, see McKay [1981]. In related work, the *saucy* program [Darga et al., 2004] specialises certain algorithms used by nauty for use on graphs that represent conjunctive normal form (CNF) formulae. In particular, saucy makes optimisations to the partition refinement procedure by exploiting properties, such as the sparsity, of such graphs. Furthermore, *saucy* is shown to outperform nauty by several orders of magnitude for such graphs, when identifying automorphisms. These works indicate the power of partition refinement and highlight the importance of using efficient programming techniques. Unlike nauty, however, saucy has no facility to produce canonical labels. Therefore, we focus on providing an overview of the concepts used by nauty in the next two sections, to show how a canonical label can be identified. First, we present the key methods behind partition refinement and highlight their relevance.

The main structure used in the algorithm is the *partition*. Given a graph $G = (V, E)$, a partition of V is a set of disjoint non-empty subsets of V, whose union is V. For example, a partition of the vertices in $G2$ in Figure 2.1 could be, $\{\{a, b\}, \{c\}, \{d\}\}$. The elements of a partition are typically called its *cells*. A *trivial* cell contains a single vertex, and this vertex is said to be *fixed*. A *discrete* partition contains only trivial cells, and a *unit* partition contains a single cell. An *ordered partition* Π of $G = (V, E)$ is defined as a *list* of cells, $\Pi = [C1, C2, \ldots, Cp]$, where each cell is disjoint, and whose union is V.

A partition Π' is referred to as being *finer* than some other partition Π if and only if each cell in Π' is a subset of some cell of Π. The converse case is defined as being *coarser*.

An example of a ordered discrete partition is $[\{a\}, \{b\}, \{c\}, \{d\}]$ (for the rest of the document ordered discrete partitions will be referred to as just discrete partitions, unless made explicit). The sequence of vertices defined by the discrete partition has a corresponding adjacency matrix, see Table 2.1. The matrix can be viewed as a single binary string comprised of the concatenation of each row of the matrix.

Definition 2.9. Let *compute_label* compute for a given graph and discrete partition the row-by-row binary string of its corresponding matrix.

For example, given the graph, G2 in Figure 2.1 and the discrete partition $d_1 = [\{a\}, \{b\},\{c\},\{d\}]$, then $compute\_label(G, d_1) = 0110101001000100$ (refer to the matrix in Table 2.1).Note that binary strings form an ordered set, if equipped with the lexicographic relation. Therefore it is easy to identify a unique, *canonical* label for a graph, i.e., choose the minimum/maximum element. Continuing from the last example, d_1 could have been ordered differently, e.g., $[\{b\},\{a\},\{c\},\{d\}]$. Indeed, there are 4! different discrete partitions, each of which has a corresponding binary string. Considering the ordered set of binary strings for this graph, its canonical label is taken as the least element (or maximum; although this document uses the least). Use of this method for computing canonical labels is not practical in general though, since the number of different discrete partitions is very large for large graphs: for a graph $G = (V, E)$, this number is $|V|!$.

Partition refinement aims to *restrict* the number of discrete partitions considered while searching for the canonical label. One simple restriction is to consider only the different discrete partitions that preserve the vertex degree sequence, see Definition 2.7. Partition refinement extends this invariant. The extension uses the notion of *equitable/stable* partitions.

Definition 2.10. Given a graph $G = (V, E)$, $v \in V$ and $W \subseteq V$, then the number of elements of W that are adjacent to v in G, is denoted as $d(v, W)$.

Definition 2.11. A partition Π is said to be *equitable/stable* if, for all cells $C_1, C_2 \in \Pi$ and $v_1, v_2 \in C_1$, then $d(v_1, C_2) = d(v_2, C_2)$.

If a partition is not equitable, it can be *refined* to make it so. In practice an efficient algorithm to compute the refinement is required, such as those in McKay [1981] and Kocay [1996]. Let us denote the refinement procedure, *refine*. Observe the following propositions that hold for partitions used with *refine* (see McKay's paper for correctness proofs):

Proposition 2.12. *Partition* $\Pi' = [C1', C2', \ldots, Cp']$ *refines* $\Pi = [C1, C2, \ldots, Cq]$ *(Π' is finer than Π); $i \leq j \Rightarrow \exists\, x, y.Ci' \subseteq Cx \wedge Cj' \subseteq Cy \wedge x \leq y$.*

Proposition 2.13. *Given an unordered partition* Π*, there exists a unique, coarsest, unordered equitable partition* Π_e *that is finer than* Π.

Definition 2.14. The *refine* function maps any partition Π to an equitable partition Π', such that Π' refines Π by Proposition 2.12, and Π' is the unique, coarsest equitable partition of Π that is at least as fine as it.

As an example, consider the two ordered partitions $\Pi_1 = [\{a, b\},\{c, d\}]$ and $\Pi_2 = [\{a\}, \{b\}, \{c, d\}]$. Π_1 has four discrete partitions that are valid refinements, namely, $[\{a\}, \{b\}, \{c\}, \{d\}]$, $[\{b\}, \{a\}, \{c\}, \{d\}]$, $[\{a\}, \{b\}, \{d\}, \{c\}]$, and $[\{b\}, \{a\}, \{d\}, \{c\}]$. Π_2 however, has just two: $[\{a\}, \{b\}, \{c\}, \{d\}]$ and $[\{a\}, \{b\}, \{d\}, \{c\}]$. Note that Π_2 is a valid refinement of Π_1. This illustrates that refining a partition Π restricts the number of discrete partitions that are refinements of Π. If the partition is already stable, refining it produces the original partition and no restriction is gained. However, if it is not stable, the restriction can be substantial. This is a key advantage of partition refinement.

In practice, refinement uses an invariant that orders cells in a partition by ascending degree. Hence, *refine* operates over ordered partitions. A number of ordered partitions correspond to an unordered partition. If $\mathit{refine}(\Pi) = \Pi'$, then Π' is one of possibly many ordered partitions that correspond to the unique, coarsest equitable unordered partition finer than Π. Therefore an implementation of the *refine* function must take care to ensure consistency. For example, given a graph G, an automorphism $\pi \in Aut(G)$, and two partitions for G, Π_α and Π_β, where $\Pi_\alpha = (\Pi_\beta)^\pi$, then the refinement of these partitions should preserve the permutation between the resulting partitions, and their ordering. That is, refinement should guarantee, $\mathit{refine}(\Pi_\alpha) = (\mathit{refine}(\Pi_\beta))^\pi$.

The procedure for partition refinement is presented in Algorithm 2, as in McKay [1981]. For a thorough reference, including correctness proofs and optimisations of this technique, refer to McKay's paper.

Observe that for an equitable partition produced by partition refinement, and vertices v_1 and v_2 in different cells, there exists a cell, C, such that $d(v_1, C) \neq d(v_2, C)$, see line 6 of Algorithm 2. Therefore, in terms of the automorphisms of a graph, v_1 and v_2 cannot be symmetric, i.e., $\forall \pi \in Aut(G) \cdot v_1 \neq v_2^\pi$.

2.3.1 Automorphisms of a Graph

This section introduces the properties of the automorphisms of a graph, required for the sections ahead.

Recall from Section 2.2 that an automorphism on a graph $G = (V, E)$ is a permutation $\pi \in Aut(G)$ that leaves V and E invariant. A cell C of a partition Π is said to be *fixed* by $Aut(G)$ if $C^\pi = C$ for all $\pi \in Aut(G)$. If $\Pi = [C_1, C_2, \ldots, C_p]$, then

Algorithm 2 *refine*(π, G): Refining a partition

Require: Graph G and partition $\pi = [V_1, \ldots, V_n]$
1: $\tilde{\pi} := \pi$;
2: $\alpha = [V_1, \ldots, V_n]$;
3: **while** $\tilde{\pi}$ is not discrete and α is not empty **do**
4: Remove an element W from α;
5: **for all** $k \in 1 \ldots n$ **do**
6: Compute ordered partition $[Y_1, \cdots, Y_s]$ from V_k, where $\forall\, i, j, x, y \cdot 1 \leq i, j \leq s \wedge x \in Y_i \wedge y \in Y_j \Rightarrow i < j \Leftrightarrow d(x, W) < d(y, W)$
7: **if** $s > 1$ **then**
8: update $\tilde{\pi}$ by replacing the cell V_k with the cells $Y_1, \ldots, Y_s$;
9: $\alpha = concatenate(\alpha, [Y_1, \ldots, Y_s])$;
10: **end if**
11: **end for**
12: **end while**
13: **return** Equitable partition, $\tilde{\pi}$;

Π^π is denoted $[C_1^\pi, C_2^\pi, \ldots, C_p^\pi]$. Kocay [1996] proves the following proposition, which connects partition refinement and the automorphism group.

Proposition 2.15. *Let* Π_0 *be a partition of* $V(G)$ *such that* $Aut(G)$ *fixes every cell of* Π_0. *Let* Π *a refinement of* Π_0. *Then* $Aut(G)$ *fixes every cell of* Π.

2.4 Stabilising Vertices

In order to find a canonical label for a graph (V, E), an efficient algorithm searches a minimum number of vertex orderings of V. The last section describes one of the major steps that can be used to reach this goal. However, it is not sufficient alone; ideally, refining *any* partition should generate a partition with more cells (a more discretised partition) – but this is not true when stabilising a partition that is already stable, in which case the same partition is generated. Hence an extra technique, called *splitting* is used to manipulate a stable partition into a valid refinement of itself, in an attempt to make it non-stable, so that partition refinement can be applied once again. Thus, we have a method for progressively making a partition finer, until discrete partitions are obtained, and from which we can identify a canonical label. This section describes splitting and several important properties of the techniques described so far.

***Splitting Cells*:** Given a non-discrete ordered partition Π, pick a cell C from it and let $u \in C$. Also let Γ be a subgroup of $Aut(G)$ such that Γ fixes Π. Concentrate on orderings where u is further left than any other element of C. Split C into two cells, $\{u\}$ and $C - \{u\}$. u is said to have been *fixed*. Call the modified partition Π'_u. By Proposition 2.12, Π'_u is a valid refinement of Π. Note that although Γ no longer fixes Π'_u, the subgroup $\Pi_u = \{\gamma \in \Gamma \mid u^\gamma = u\}$ that fixes u also fixes $C - \{u\}$ and all other

cells of Π'_u. This idea will be used later. As an example of a partition that has been fixed, Π could be $[\{u, v, w\}, \{x, y, z\}]$ and Π'_u be $[\{u\}, \{v, w\}, \{x, y, z\}]$.

Definition 2.16. Let Γ be a permutation group acting on a set V. The subgroup Γ_u containing permutations that fix u is called a *stabiliser subgroup.*

No discrete partition that is a valid refinement of a stable partition Π can be discarded during the search for a canonical label. Therefore Π must be split $|C|$ times, so that all $u \in C$ are fixed eventually. Continuing the previous example, Π must also be split into $[\{v\}, \{u, w\}, \{x, y, z\}]$ and $[\{w\}, \{u, v\}, \{x, y, z\}]$.

Vertex Stabilisation: Stabilisation can be applied post-splitting since the result may not be stable. The process of fixing a vertex u in a partition Π to get Π'_u, and refining Π'_u to an equitable partition Π_u is called *vertex stabilisation.* It usually produces a refinement of Π that is far finer, from which considerably fewer discrete partitions can be generated.

Pseudocode for the recursive algorithm that uses these ideas to find the smallest adjacency matrix associated with an ordering is given in Algorithm 3:

Algorithm 3 *stabilise*(Π, G): Finding a canonical label of a graph

Require: Unlabelled, undirected graph, G

```
1: Π_e = refine(Π); // refine Π to an equitable partition
2: if Π_e is discrete then
3:    // compare with smallest label so far
4:    v = compute_label(G, Π_e);
5:    if v < best then
6:       best = v; // update label
7:    end if
8: else
9:    // Π_e is not discrete
10:   C = first non-discrete cell in Π_e;
11:   for all u ∈ C do
12:      make a copy Π_u of Π_e in which C is split into u and C − {u}
13:      stabilise(Π_u, G);
14:   end for
15: end if
```

Algorithm 3 shows that splitting takes place on the *first* non-discrete cell of a non-discrete ordered partition. The choice could have been made using a more complicated rule that may reduce the size of the search tree, but the computation overhead generally outweighs this benefit due to the high frequency of splitting in the algorithm. Hence the algorithm opts for the simple choice.

Given a graph $G = (V, E)$, *stabilise* can be called with the unit partition containing V to find the canonical label for G. The algorithm generates a tree structure where the root is the unit partition and the leaves are discrete partitions. As an example,

given the graph in Figure 2.3, Figure 2.4 then shows its corresponding search tree. The canonical label is taken to be the lexicographically least adjacency matrix of the two discrete partitions. The worked example can be found in Appendix A. Note that it is possible to identify any discrete partition in a search tree by a list of fixed vertices that trace its location in the tree from the root. For example, in Figure 2.4 the discrete node, [{b},{e},{c},{d},{a}] can be identified by the list [*fix(e)*]. The root node of the tree corresponds to the empty sequence [], where no vertices have been fixed.

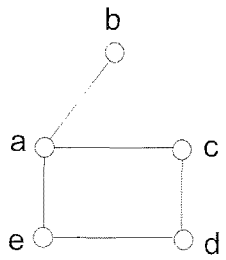


FIGURE 2.3: A simple graph

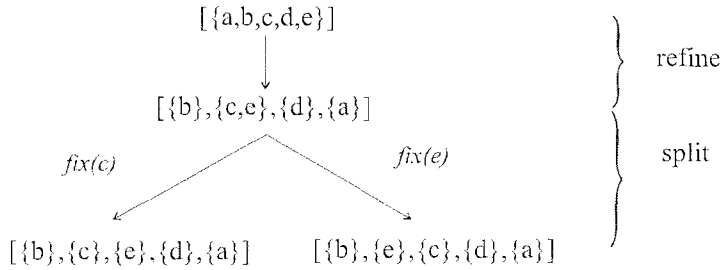


FIGURE 2.4: An example search tree generated by *stabilise*([{a, b, c, d, e}]), for the graph in Figure 2.3.

***Canonical labels are the same for isomorphic graphs*:** Consider two graphs $G = (V, E)$ and $H = (V^\pi, E^\pi)$, where π is the vertex transformation between the G and H. Now consider the search trees for G and H generated by the *stabilise* algorithm using the unit partitions as the initial partition. The unit partition Π_a of G corresponds to the unit partition Π_a^π of H. Let Π_b be the equitable partition of Π_a produced by partition refinement. Then Π_b^π is the stabilisation of Π_a^π. Apply this idea recursively to the whole tree. This means the set D_G of discrete partitions in G correspond to the set D_G^π in H. Since the two graphs have the same structure, and π is the transformation between their vertices, the adjacency matrices that correspond to D_G are the same as those for D_G^π. This illustrates that the canonical labels for both graphs are the same.

***Non-isomorphic graphs have different canonical labels*:** Let the *stabilise* algorithm generate the same canonical labels for two non-isomorphic graphs. This is impossible since the label is derived from the adjacency matrix, which determines the structure

of the graph; and these graphs have different structures. Therefore non-isomorphic graphs have different canonical labels.

Reducing the size of the search tree: Consider a run of the search procedure given in Algorithm 3, which encounters an initial discrete partition with the label '01...' (found using *compute_label* in Definition 2.9). Lines 5 and 6 guarantee that the canonical label found will begin with either '01' or '00'. However, it is still possible to generate discrete partitions greater than this, beginning with either '10' or '11'. The first optimisation, called *lexicographic pruning*, exploits such redundant searches by observing that partitions are progressively discretised from left to right (line 10). Therefore, the *stabilise* procedure does not need application to any partition encountered whose discretised left hand cells determine a partial adjacency matrix beginning with either '10' or '11'. An appropriate change to Algorithm 3 simply requires the computation of a partial adjacency matrix, and its comparison to the lowest label found so far, before executing line 13. The effect of this improvement can be a considerable reduction in the size of the search tree, and a reduction in the time required to find the canonical label.

The second optimisation exploits the occurrence of multiple discrete partitions corresponding to the same adjacency matrix. Once more, the effect is to prune the search tree so that it takes less time to identify a canonical label. The strategy involves the use of automorphisms of the original graph that are inferred during the search. Therefore, this optimisation is called *automorphism pruning.*

Consider the search tree for a graph G. Let $\pi \in Aut(G)$. Let Π_1 be a node in the tree, and O_1 be the set of nodes that descends from Π_1. Also, let Π_2 be another node in the tree, such that $\Pi_2 = \Pi_1^\pi$. This means the set of nodes O_2, below Π_2, is O_1^π. Moreover, the labels associated with the discrete partitions in O_2, are exactly those of O_1; so O_2 does not need to be searched if O_1 has already been searched.

An automorphism is found whenever two discrete partitions are found to have the same adjacency matrix, as shown in Definition 2.17, e.g., if two discrete partitions α and β have the same adjacency matrices, the transformation from α to β, and vice-versa is in $Aut(G)$.

Proposition 2.17. *If π is a permutation of V, A^π shall denote the adjacency matrix obtained from A by permuting the rows and columns by π. Two orderings π_1 and π_2 are equivalent if $A^{\pi_1} = A^{\pi_2}$. So $A^{\pi_1 {\pi_2}^{-1}} = A$. Therefore $\pi_1 {\pi_2}^{-1} \in Aut(G)$.*

A naïve method to make use of automorphisms is as follows. Whenever a partition is produced in the search tree, find the transformations between itself and every other partition in the tree. If one of these transformations is an automorphism, the search below this partition can be omitted. This approach is computationally expensive, especially for graphs with large search trees and large automorphism groups. It can be improved

and is discussed next. First, some definitions need to be given (as they appear in Kocay [1996]).

Let Γ be a permutation group acting on V, $\pi \in \Gamma$, $u \in V$ and $v = u^\pi$.

Definition 2.18. The *conjugate* of a permutation $\gamma \in \Gamma$ by π is $\pi^{-1}\gamma\pi$, denoted by γ^π. If γ maps u to v, then γ^π maps u^π to v^π.

Definition 2.19. The conjugate of a subgroup Γ_u is $\pi^{-1}\Gamma_u\pi = \Gamma_u^\pi = \Gamma_v$. The conjugate of the stabiliser of u by π is the stabiliser of $v = u^\pi$.

Definition 2.20. If Γ_u fixes a partition Π, then Γ_u^π fixes Γ^π.

Proposition 2.21. *Take Γ to be a subgroup of $Aut(G)$ that fixes Π, and Π_u is obtained by partition refinement. Let $\pi \in \Gamma$ and $v = u^\pi$. Then $\Gamma_v = \Gamma_u^\pi$.*

Let $\Pi_0, \Pi_1, \ldots, \Pi_k$ be the sequence of partitions occurring in a path to a leaf node in the search tree. Π_0 is the initial equitable partition that is obtained by stabilising the unit partition. Vertex u_0 in Π_0 is fixed, and after stabilisation, Π_1 is obtained. Then u_1 in Π_1 is fixed etc. The branch of the search tree rooted at Π_0 on the edge labelled u_0 gets searched by *stabilise*(Π_1). If however, u_0^π is fixed in Π_0 instead, to get Π_1^π, then the branch descending from u_0^π is isomorphic to the branch just searched. Having fixed u_0, no other points u_0^π need to be fixed where $\pi \in Aut(G)$. Now apply this idea more generally. Let $\Gamma_0 = Aut(G)$ for some graph G, and Γ_1 be the stabiliser subgroup of u_0 in Γ_0, such that Γ_1 fixes Π_1. Let Γ_i be the subgroup of Γ_{i-1} that is obtained by fixing vertex u_{i-1}. Then Γ_i fixes Π_i. Now, *stabilise* chooses a vertex u_i in the first non-trivial cell of Π_i and fixes it. If $\pi \in \Gamma_i$, the nodes in the search tree descending from Γ_i on the edge labelled u_i is isomorphic to the one descending on the edge labelled u_i^π. So having chosen to fix u_i in the first non-trivial cell of Π_i, no other points u_i^π need to be fixed where $\pi \in \Gamma_i$. That is, to find the canonical label, only one point needs to be fixed from each orbit of Γ_i on the first non-trivial cell of Π_i.

2.5 Discussion

This chapter, so far, describes the most efficient general purpose graph isomorphism algorithm known to date, namely partition refinement, and the key ideas that make it successful. In particular, it uncovers a method to canonically label a graph, so that isomorphic graphs can be detected easily. In practice, these graphs may represent certain structures used in the formal specification of a system. Such isomorphism algorithms then make it possible to identify equivalent structures of a system. We make use of this premise later in Chapter 4, where we describe the how to identify symmetric states of B systems.

The remaining sections of this chapter focus on *how* techniques that identify symmetries inherent in a system can be exploited to produce a more efficient model checking strategy.

In Sections 2.6 and 2.7 symmetry reduction is introduced in more detail and a description is given for a classic implementation technique. This is followed by a literature review of 5 model checkers that use symmetry reduction, in Section 2.8.

2.6 Symmetry Reduction in Model Checking

Standard model checking performs an *exhaustive search* of the state space of a system to determine whether some specification holds, and has been presented for ProB in Algorithm 1. The size of this state space can grow exponentially as more components are added to the specification, e.g., a new variable. As a result, the verification of large systems becomes intractable; requiring too much time/memory for the search to terminate. Symmetry reduction is one of several techniques (amongst others, see Section 1.3) that can be employed by model checkers to try and alleviate this problem.

The premise behind symmetry reduction is to identify symmetries in the system being model checked, such that symmetric states may be interchanged with no effect on the behaviour of the system. States encountered during checking, that are symmetric to others already reached, can then be discarded to avoid redundant searches. The result of exploiting symmetries is to search a smaller, equivalent state space. The major problem associated with this scheme is finding a computation to identify symmetries, whose time saved by a constrained search is not outweighed by its computational inefficiency.

Considerable progress has been made over the past 10 years in advancing symmetry reduction in model checking. One of the first descriptions is by Huber et al. [1984] for describing high-level Petri Nets, which is developed by Starke [1991] for deadlock and liveness checking. Following this, Clarke et al. [1993] and Emerson and Sistla [1993] discovered ways to exploit symmetry in temporal logic model checking. Expanded descriptions of these papers, amongst others, can be found in the journal article by Emerson and Sistla [1996].

Although there has been a wide range of research into this area, so far no literature exists on symmetry reduction for model checking systems specified in the *B language* – despite its increasing popularity. Hence, a primary goal of this research is to develop existing techniques for symmetry reduced model checking, for application to the B language and the B model checker, ProB.

2.7 Classical Technique

The most frequently used technique for symmetry reduction is based on exploiting symmetries induced by the transition relation of a finite-state concurrent system, i.e., the behaviour of the system. Symmetry can be found, for example, in memories, caches, network protocols, a software specification of a phonebook – anything with replicated structure. Regarding the Kripke structure of a system, $\mathcal{M} = (S, R, L, S_0)$, these symmetries imply the existence of automorphisms of the form $\rho \in S \rightarrowtail\!\!\!\!\!\rightarrow S$, which preserve the transition relation, R, such that for every transition $a \mapsto b \in R$, we have $\rho(a) \mapsto \rho(b) \in R$.

The group of all automorphisms for a Kripke structure $\mathcal{M}$ is denoted $Aut(\mathcal{M})$. Given a subgroup of these automorphisms, denoted G, one can partition the set of states, S, into disjoint orbits (Definition 2.5), which may be used to construct a *quotient* Kripke structure for the system, denoted $\mathcal{M}_G$. Generally, $\mathcal{M}_G$ is smaller in size than M (when there exist automorphisms other than $id(S)$), but is never larger. Therefore, model checking such structures usually requires less memory and time for verification. Given now is a formal definition of $\mathcal{M}_G$:

Definition 2.22. Given Kripke structure $\mathcal{M} = (S, R, L, S_0)$, with a group of automorphisms G, there exists a quotient Kripke structure, $\mathcal{M}_G = (S_G, R_G, L_G, S_G^0)$, where:

- $S_G^0 = \{[s]_G \mid s \in S_0\}$,
- $S_G = \{[s]_G \mid s \in S\}$,
- $R_G = \{[s]_G \mapsto [t]_G \mid s \mapsto t \in R\}$,
- $L_G([s]_G) = L(rep([s]_G))$, where $rep([s]_G)$ finds a unique representative of $[s]_G$

For quotient structures to be of use in model checking, one must ensure that any violation of the correctness conditions in the original Kripke structure corresponds to a violation in the quotient structure. In previous work [Emerson and Sistla, 1993], it has been proved that for every symmetric CTL* formula ψ and every state $s \in S$, then $\mathcal{M}_G, [s]_G \models \psi \Leftrightarrow \mathcal{M}, s \models \psi$. Indeed, this ensures that checking a property for a system over a quotient Kripke structure is sound with respect to checking the property over the original Kripke structure. A CTL* formula, ψ, is *symmetric* with respect to a group of automorphisms, G, if for each maximal propositional subformula, υ, that appears in ψ, then for every $\rho \in G, \mathcal{M}, s \models \upsilon \Leftrightarrow \mathcal{M}, \rho(s) \models \upsilon$.

The quotient model given in Definition 2.22 uses orbits as states, e.g., S_G^0 in Definition 2.22. The standard approach, however, is to make use of *representatives* of orbits – computed by some function, *rep* that maps a state to its representative state. Let us define formally the *rep* function, and modify the definition of $\mathcal{M}_G$ accordingly:

Definition 2.23. Given a group of automorphisms, G, let $rep \in S \rightarrow Rep$ be the function mapping a state in the original model, to its orbit representative state in the quotient model, such that for every $s, s' \in S \cdot (rep(s) = rep(s') \Leftrightarrow [s]_G = [s']_G)$.

Definition 2.24. Given Kripke structure $\mathcal{M} = (S, R, L, S_0)$, with a group of automorphisms G, there exists a quotient Kripke structure, $\mathcal{M}_G = (S_G, R_G, L_G, S_G^0)$, using rep in Definition 2.23, where:

- $S_G^0 = \{s \mid \exists\, s' \cdot (s' \mapsto s \in rep \wedge s' \in S_0)\}$,
- $S_G = Rep$,
- $R_G = \{s \mapsto s' \mid s, s' \in Rep \wedge \exists\, r \in S \cdot (s \mapsto r \in R \wedge r \mapsto s' \in rep)\}$,
- $L_G(s) = L(rep(s))$

To illustrate the use of unique representatives from each orbit of a state, Figure 2.5 presents again the Kripke structure for the two-process mutual exclusion problem, introduced in Section 1.2; this time, however, states within the same orbit are indicated by identical shapes in their upper left-hand corners. Figure 2.6 then presents the quotient model for the system: once again, this illustration is has been adapted from [Miller et al., 2006, Figure 2]). Observe that the application of the transposition, $\pi = (0\,1)$, in the original structure generates an identical structure. Thus, π is an automorphism of the system. The automorphism then makes it possible to analyse orbits of states and determine unique representatives for use in the quotient model, as in Figure 2.6.

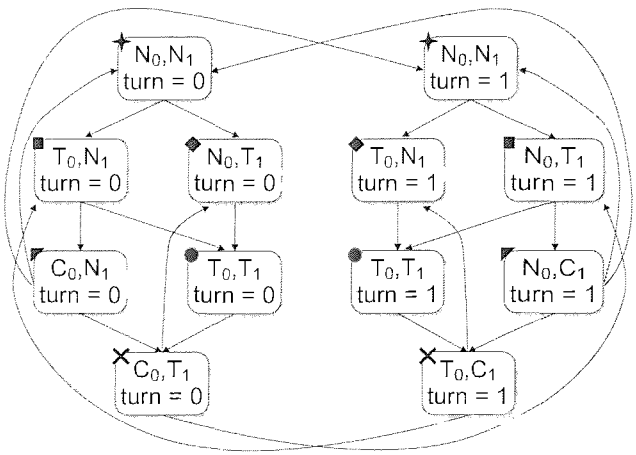


FIGURE 2.5: Kripke structure for the two-process mutual exclusion problem

Procedures that compute representatives can be computationally expensive. Jha [1996] shows that the problem of testing the equivalence of two states, known as the *orbit problem*, is harder than the graph isomorphism problem; therefore techniques may choose not to approach the orbit problem directly. One method is use a more efficient computation that maps each orbit to a subset of representative states – as opposed to a unique representative. For this case, one defines a representative *relation*, as in Definition 2.25:

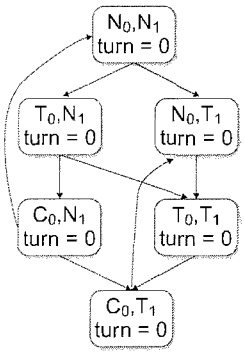


FIGURE 2.6: Symmetry reduced Kripke structure for the two-process mutual exclusion problem

Definition 2.25. Given a group of automorphisms, G, let $rep \in \mathbb{P}(Rep \times S)$ be the representative relation if, for every $s, s' \in S \cdot s \mapsto s' \in rep \Leftrightarrow s \in Rep \wedge [s]_G = [s']_G$.

A formal description of a corresponding quotient structure is now given:

Definition 2.26. Given Kripke structure $\mathcal{M} = (S, R, L, S_0)$, with a group of automorphisms G, there exists a quotient Kripke structure, $\mathcal{M}_G = (S_G, R_G, L_G, S_G^0)$, using rep in Definition 2.25, where:

- $S_G^0 = \{s \mid \exists\, s' \in S_0 \cdot (s \mapsto s' \in rep)\}$,
- $S_G = Rep$,
- $R_G = rep^{-1}\,;\ R\,;\ rep$ (where ';' denotes relational composition),
- $L_G(s) = L(rep(s))$

2.7.1 The Modified Model Checking Algorithm

The integration of a representative computation (Definitions 2.23 and 2.25) into a standard model checking algorithm, such as Algorithm 1, requires only a small change. Recall that in the original procedure, for each transition to some new state, s, one adds s to the set of states that will be checked in the future. Symmetry reduced checking differs by adding the representative of s, namely $rep(s)$, to this set to be checked. Thus, for several symmetric states with the same representative, only a single representative needs checking. We now formalise the modified search procedure. For convenience, we also present a generalised version of Algorithm 1. The procedures are given in Algorithms 4 and 5, and have been adapted from Ip and Dill [1993] and Bosnacki et al. [2002].

The success of a technique for symmetry reduction depends largely on the efficiency of the identification of symmetries, using such *rep* computations, in addition to the overall reduction in model size. One should note that, if a system exhibits very few symmetries, the symmetry reduction observed will be limited.

Algorithm 4 Standard Model Checking Exploration

Require: System description with transition relation R, and specification ϕ
  reached := *unexpanded* := $\{s \mid s \in S_0\}$;
  while *unexpanded* $\neq \varnothing$ **do**
    remove a state s from *unexpanded*;
    for all $s \mapsto s' \in R$ **do**
      if $s' \not\models \phi$ **then**
        stop and return trace from an initial state to error;
      end if
      if $s' \notin$ *reached* **then**
        add s' to *reached* and *unexpanded*;
      end if
    end for
  end while

Algorithm 5 Model Checking with Symmetry Reduction

Require: System description with transition relation R, and specification ϕ
  reached := *unexpanded* := $\{rep(s) \mid s \in S_0)\}$;
  while *unexpanded* $\neq \varnothing$ **do**
    remove a state s from *unexpanded*;
    for all $s \mapsto s' \in R$ **do**
      if $s' \not\models \phi$ **then**
        stop and return trace from an initial state to error;
      end if
      if $rep(s') \notin$ *reached* **then**
        add $rep(s')$ to *reached* and *unexpanded*;
      end if
    end for
  end while

In the following section a review of five model checkers that use symmetry reduction is presented.

2.8 Related work

Having provided the motivation for symmetry reduction in model checking, and a classical technique, let us now introduce five popular model checking verification systems, and examine their strategies for symmetry reduction.

2.8.1 Murϕ and Scalarsets

The Murϕ protocol verification system [Dill et al., 1992] consists of a description language and an explicit state model checker, which has been developed gradually over the past decade. System descriptions in the Murϕ language were inspired by the Unity language [Chandy and Misra, 1988], and consist of a set of iterated guarded commands.

(A guarded command is a condition followed by a sequence of actions.) Properties of a system can be specified using *assert* statements, defined in-line with guarded commands, and/or via a system *invariant*. However, the verifier does not support the specification of any other temporal properties. Model checking then consists of enumerating through the reachable states of the system, and checking whether the assertions and invariant conditions are violated in some state, or if there exist any deadlocks – any counterexamples are reported to the user.

The Murϕ verifier also makes use of a symmetry reduction [Ip and Dill, 1993] strategy that conforms to Algorithm 5, so that quotient structures are progressively constructed as the search space is explored. This makes it possible to verify systems that would be intractable using standard checking alone. Symmetries are exploited using a special construct in the description language called a *scalarset*, which represents an integer subrange of symmetric elements. This symmetry is defined such that program behaviour is invariant under arbitrary permutation of elements of a scalarset – known as *full* symmetries. As an example, a system that uses an array of processors might declare the process identifiers used to index the array as belonging to such a scalarset. A possible reason why a specifier may choose this could be because they are not interested in processor i processing job j; instead they may only be concerned that *some* processor is processing j. Consider Figure 2.7, showing two states in a system with one variable, an array of scalarset type; state A and B could be represented by two different states, but are in fact symmetric; this is because each array tells us that it contains 3 elements of the same symmetric type.

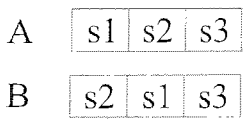


FIGURE 2.7: Two equivalent states, each containing one array of scalarset type.

To preserve the symmetric properties of scalarsets, they must only be subject to a restricted set of operations. For example, scalarset variables can only be compared for equality; it would make no sense to apply the binary operator, '<', to scalarset elements, since they have no order. Therefore, prior to model checking, a Murϕ system description is analysed during a compilation phase to ensure the correct use of scalarsets. For more details, see Ip and Dill [1993].

The Murϕ verifier provides two methods of selecting representative states, both of which utilise a lexicographic ordering on the states of the system. Let the term, *state vector*, denote the sequence of variables and their values in some state. The first approach uses a *canonicalisation* function that applies all permutations to a state vector, and records the lexicographically least as the representative. This problem is inherently complex and is at least as hard as testing for graph isomorphism, for which there is no known

polynomial time algorithm. An alternate strategy therefore enables one to increase the speed of finding (possibly multiple) representatives, by compromising the amount of symmetry exploited. In this approach, a state vector is split into two parts: either a most/least significant part, see Figure 2.8. Permutations are then applied to its most significant section to generate the lexicographically least value. There may be several permutations, λ, that produce this value. Subsequently, a permutation from λ is used to *normalise* the least significant section of the state vector. The representative is taken to be the concatenation of both parts of the vector. By altering the location of a split point, symmetry reduction may be tuned with regard to speed, or the amount of symmetry exploited. In the extreme case, where the most significant part includes the whole state, observe that a unique representative is found per orbit so that more symmetries are exploited. However, the computation is also more expensive. On the other hand, if the split point is close to the start of the vector, e.g., $n = 2$ in Figure 2.8, representatives are computed relatively quickly, although fewer symmetries are exploited.

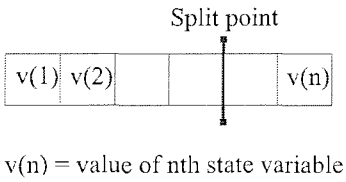


FIGURE 2.8: An example of a state-vector and it's split point.

As mentioned previously, scalarsets exploit full symmetries in a system. The result is a potential reduction in the size of a state space of $N!$, where N is the number of elements in a scalarset. This is because for such a scalarset, the maximum size of an equivalence class is $N!$, and the (canonicalisation) reduction strategy will select a unique state from each. Experimental results presented in Dill et al. [1992] indicate that it is possible to approach such savings. Systems exhibiting more general symmetries, such as rotational symmetry (e.g., ring topologies), will not be exploited by the current strategy of scalarsets, and are discussed in Ip [1996]. Potential savings for such symmetries would be a factor of N.

A disadvantage of using scalarsets is the overhead of introducing a new data type into the specification language. In addition, it is the responsibility of the specifier to identify and encode the presence of symmetries within a system. Consequently, their *ability* becomes a major factor influencing the symmetry reductions achieved.

2.8.2 SymmSpin

SPIN is a popular, on-the-fly, explicit-state model checker [Holzmann, 1997b], targeted at verifying systems specified in PROMELA, which is a language with a syntax similar to C, based on the guarded command language of Dijkstra [1976]. Through the use

of breadth/depth-first exploration, the verification system is capable of checking LTL properties.

SPIN benefits from a range of optimisations to its method of verification. Strategies for state space reduction currently include the partial order reductions of Holzmann and Peled [1994] and Holzmann [1999]. In addition, more savings in memory consumption during verification are gained through the use of state storage techniques similar to BDDs [Visser and Barringer, 1996], state compression [Holzmann, 1997a], bitstate hashing [Holzmann, 1998] and hash-compact procedures [Wolper and Leroy, 1993].

Symmetry reduction is integrated into SPIN through the SymmSpin package [Bosnacki et al., 2002]. The technique extends the work of Ip and Dill [1993] on scalarsets by adding heuristics to the computation of representatives, resulting in four new strategies. The scalarset type remains to be fully integrated into the PROMELA language, which would require an extension to the PROMELA parser. Instead, all symmetry information is supplied in an accompanying file. Accordingly, the state space exploration algorithm of SPIN is modified so that model checking progressively constructs only a quotient model.

The devised heuristics exploit the ordering of variables in the state vector, on which the lexicographical ordering is based. Moreover, by moving certain variables towards the most significant (left-hand) end of the state vector, one reduces the number of permutations to be considered when determining a representative. In the *sorted* strategy, an array indexed by a scalarset type is identified within the state vector and moved (conceptually) to the extreme left of this vector. Such an array is called a *main array*. Let us denote this main array, m_1. Subsequently, m_1 is sorted to generate m_2. Using one of the permutations between m_1 and m_2 (sorting may induce more than one permutation), the least significant part of the state vector is normalised to generate the representative. The advantage over the canonicalisation procedure, as used by Murϕ, is that sorting identifies efficiently the lexicographically least state vector.

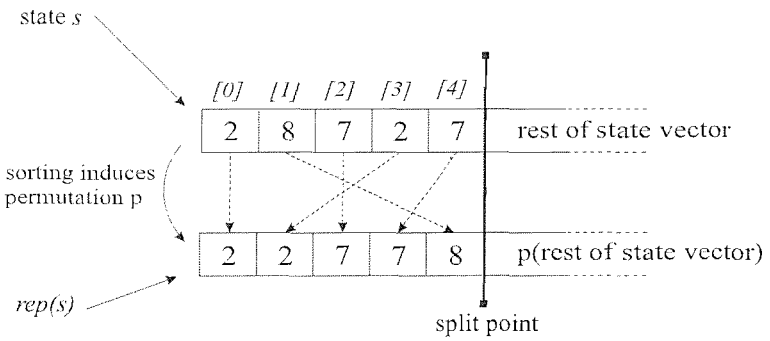


FIGURE 2.9: An example of sorting the state-vector.

An example is given in Figure 2.9 (adapted from [Bosnacki et al., 2002]), where state s, has already had a main array moved left of the split point. In this case, the main array is indexed $0 \ldots 4$. Sorting then induces a permutation, $p = \{0 \mapsto 0, 3 \mapsto 1, 2 \mapsto 2, 4 \mapsto$

$3, 1 \mapsto 4\}$, which produces the representative, $rep(s)$, when applied to the right-hand part of s. Note that this method is not canonical; it may find multiple representatives in the same orbit, e.g., there are actually 4 permutations between the main arrays in Figure 2.9, which could be used to generate a representative.

The second approach, called the *segmented* strategy, shares the underlying idea of the first approach, and modifies it to form a canonical technique. Instead of normalising the rest of the state using one permutation between main array m_1 and m_2, apply all permutations between them and select the lexicographically least state vector as the representative. This approach is guaranteed to identify a unique representative per orbit, and is therefore more memory efficient in comparison with the sorted strategy, which may identify multiple representatives. Note however, that the segmented approach is computationally more expensive than the sorted approach.

The final two strategies enable the automatic application of the sorted and segmented strategies to concurrent systems. Given the processes in a system, their program counters are placed in an array, which is then treated exactly as a main array. The approaches are called *pc-sorted* and *pc-segmented* accordingly. If a concurrent system also makes use of a standard main array, then a combination of the techniques may be applied.

Empirical results have been obtained from applying all four strategies to a range of models, including a Process Initialisation Protocol [Holzmann, 1991], and a Base Station Telecommunication Protocol [Dravapoulos et al., 1997]. As a control, a canonicalisation function is also applied, which considers all permutations of a state vector when identifying representatives (as used by the Murϕ verifier). This is called the *full* strategy. In summary, it is shown that for each technique it is possible to approach the limit for reduction [Bosnacki et al., 2002, Section 4.4.4]. Furthermore, it is the form of the program that dictates the effectiveness of any reduction. Generally, as the size of the standard main array increases, the sorted/segmented strategies outperform the corresponding pc-sorted/segmented alternatives. This is because the number of program counters has remained constant. However, if the latter is then increased, the opposite effect is observed. As already mentioned, there is also a trade off between the sorted and segmented strategies. Although the sorted strategy is more efficient computationally, it may find multiple representatives, and therefore is more memory intensive. Conversely, the segmented strategy is computationally more expensive, but generally requires less memory.

2.8.3 Symmetry in RuleBase: A Symbolic Model Checker

A symbolic representation of a search space during symbolic model checking, introduced in Section 1.3.2, usually requires less memory when compared to explicit state techniques. Ideally, adding a method for symmetry reduction to a symbolic model checker would

be only as complicated as that for explicit state model checkers. Unfortunately this is not the case. The problems associated with symmetry reduction (Section 2.7) become more complex when using symbolic representations, e.g., dealing effectively with the orbit problem.

Jha [1996] proves that the size of the BDD for the orbit relation increases exponentially with the number of BDD variables, and shows that multiple representatives per orbit can be used to reduce the size of the BDD for the orbit relation. This leaves the problem of *selecting* representatives from an orbit, since different choices affect the size of the BDDs encoding the quotient model. One approach is to avoid the need to compute representatives in advance, and simply choose as a representative the first state encountered from an orbit. This technique is proposed by Gyuris and Sistla [1999] for explicit state model checking using a depth-first search. However, the method does not apply well to symbolic model checking since such search algorithms are inefficient with BDDs.

The symbolic model checker, *RuleBase* [Beer et al., 1996] extends the CTL verification engine used by SMV to handle a new language, called Sugar. The goal is to simplify the definition of CTL specifications and make it more accessible to non-experts. Furthermore, RuleBase accepts as input the industry standard hardware description languages of Verilog [Thomas and Moorby, 2002] and VHDL [Lipsett et al., 1989]. The culmination of these efforts, among others [Beer et al., 1996], is a verification tool suitable for industrial hardware design.

Barner and Grumberg [2002] present a method for integrating symmetry reduction into RuleBase. Their approach utilises a technique for *under-approximation*, where for each newly encountered state, only a subset of the successors are explored. The use of under-approximation does not guarantee the discovery of *all* violations of the correctness conditions. However, it is effective at *falsification*, i.e., finding some violation. Therefore, to provide verification, the model checker may accept 'hints' from the user in the form of a set of generators for the system symmetries, which are invariant for the correctness conditions. These can be used to exploit symmetries during model checking in the usual way, in addition to ensuring at least one state from each orbit is explored. The choice of representative is directed by certain BDD criteria, such that multiple representatives per orbit may be found. Note that, as with the scalarset approaches of Murϕ and SymmSpin, the effectiveness of symmetry reduction in RuleBase depends on information supplied by the user and is therefore prone to errors.

Experimentation indicates this symmetry reduction performs well when applied to liveness properties, in comparison to standard checking in RuleBase. For instance, results for a Futurebus (IEEE 896) example show that verification time can be reduced by a

factor of approximately 18, and memory consumption can be reduced by around 7. However, results also indicate that a saving in memory consumption is not always observed, as illustrated by an Arbiter example.

2.8.4 Alloy Analyser

The ALLOY tool-set [Jackson, 2006a] is a software design framework consisting of a first-order, declarative language called ALLOY, for specifying abstract software systems, and the ALLOY ANALYSER, which provides a fully automatic analysis of constraints over ALLOY systems. Originally inspired by model checking, the analysis tool benefits from symmetry reductions via an approach that differs to the other methods described so far in this Section, e.g., scalarsets. Symmetries are inferred automatically from the types in the specification, after which they are exploited (the user is not required to indicate any symmetry information).

The analysis engine performs constraint-based evaluation using propositional satisfiability (SAT) solvers, including Chaff [Moskewicz et al., 2001] and BerkMin [Goldberg and Novikov, 2002]. A key difference with standard model checkers is the absence of a notion of state machines. Instead, analysis is achieved by translating a system description and set of constraints, including bounds on types, into a large boolean formula in conjunctive normal form (CNF), which is passed to a SAT solver. This attempts to find an assignment of variables for the formula, called an *interpretation*, that evaluates to `true`. If this is successful, the interpretation is called an *instance* of the model. The analyser is said to perform *simulation* if the instance denotes states or executions that satisfy a given property. Conversely, if the instance violates a given property (a counterexample), *checking* is performed. Given an instance, the formula is converted back into the tool's native language and reported to the user. Despite the lack of a state machine idiom, it is possible to check LTL properties by modelling traces explicitly. However, this is shown to be effective typically for types with small bounds [Jackson, 2006a, Section 6.1.3 - 6.1.5].

During analysis, all types are represented internally as relations. Formulas involving scalars or sets are translated into an equivalent one involving only relations, using a method inspired by Schmidt and Ströhlein [1993]. Elements of these relations remain uninterpreted throughout, and therefore share similarities with the elements of scalarsets; an injective relabelling of the atoms has no effect on formula evaluation. However, unlike the methods that use scalarsets, where two states are tested directly for symmetry, this strategy seeks to ensure symmetric interpretations are prevented from being generated (before an equivalence check could take place). This is accomplished by adding *symmetry-breaking predicates* [Crawford et al., 1996] to the formula passed to the SAT solver, which are constructed using symmetry information inferred from properties of data structures used in a system.

The identification of *perfect* symmetry-breaking predicates, where exactly one interpretation is generated per isomorphism class, is an NP-complete problem [McKay, 1998]. Furthermore, such predicates would be large and would be likely to compromise the performance of analysis. Therefore, a non-perfect strategy is employed, which simplifies computation. *Partial symmetry-breaking predicates* are used that find *at least one* interpretation from each isomorphism class. These predicates must be carefully chosen so that a large fraction of interpretations are eliminated, and yet they must also be compact to ensure interpretations are efficiently computed. Shlyakhter [2007] describes techniques for finding partial symmetry-breaking predicates for the specific data types used by ALLOY that eliminate over 99% of symmetric interpretations, and Khurshid et al. [2003] shows how they can be applied during analysis. The precise details of this are not given here; instead, an example borrowed from Crawford et al. [1996] is presented to show how symmetry-breaking predicates may be used.

Consider the CNF boolean formula $(a \vee \overline{c}) \wedge (b \vee \overline{c}) \wedge (a \vee b \vee c) \wedge (\overline{a} \vee \overline{b})$, for which a SAT solver will try to find instances. Let t denote `true` and f denote `false`. The only two models are: $(a = t,\ b = f,\ c = f)$, and $(a = f,\ b = t,\ c = f)$. There is one automorphism of the formula; swap a for b. The application of an automorphism to an interpretation generates an interpretation with the same truth value. Then, to break the symmetry defined by the automorphism, the automorphism itself is added to the original formula. In this case, the automorphism can be defined by a implies b, or $a \rightarrow b$. The formula extended with the symmetry-breaking predicate is then, $(a \rightarrow b) \wedge (a \vee \overline{c}) \wedge (b \vee \overline{c}) \wedge (a \vee b \vee c) \wedge (\overline{a} \vee \overline{b})$. Considering this automorphism, the new formula retains the semantics of the original formula, but there is now only *one* model: $(a = f,\ b = t,\ c = f)$. That is, the symmetric model has been broken.

The use of symmetry-breaking predicates in the ALLOY tool-set is an effective alternate to classical symmetry reduction, as demonstrated by numerous successful case studies, including checking Microsoft's Common Object Modelling interface for intercommunication [Box, 1998], the Intentional Naming System for resource discovery in mobile networks [Adjie-Winoto et al., 2000], and avionics systems [Dennis, 2003], and the design of certain cancer therapy machines [Jackson and Jackson, 2006]. In addition, there appears to be a large scope for further improvements, especially since the techniques can be used in conjunction with independent, state of the art SAT solvers, for which there is a separate abundance of research.

2.8.5 Recent Development in PROB, I: Permutation Flooding

The following two sections present two recent approaches to symmetry reduction in PROB, whose development has been stimulated by the research documented in Chapter 4 of this thesis.

The first technique, which is called *permutation flooding* [Leuschel, Butler, Spermann, and Turner, 2007], uses an alternate strategy for addressing the orbit problem when compared with classical symmetry reduction. The method works by permuting states encountered during model checking, and adding them to the state space. The premise is that the permuted states would eventually be reached by model checking, and yet are relatively easy to compute for B's data structures.

The permutations used by this approach are defined by a permutation function that permutes certain abstract elements in a B machine, giving rise to symmetries in data structures used within the system[1]. These abstract elements belong to a standard type of set in B, called the *deferred set*. The application of the permutation function to a B state will generate a symmetric state. Therefore, one can draw similarities between symmetries that arise from deferred sets, and those induced by scalarsets [Ip and Dill, 1993, Bosnacki et al., 2002], as used in other symmetry reduction schemes.

Permutation flooding optimises the standard exhaustive checking algorithm used in ProB through the use of symmetric state permutations. Each time a new state is encountered, all permutations/symmetries are computed and added to the state space. The key advantages of using this methodology is:

- Computation of permutations requires only the permutation of deferred set elements and is therefore relatively simple; even for complicated B data structures.
- The orbit problem is addressed for any state s by adding to the model all states symmetric to s, denoted by $[s]$, where $s \in [s]$, and then marking all states in $[s]\backslash\{s\}$ as being 'already visited'. Future checking then will not compute successors of $[s]\backslash\{s\}$, but does so for s, and thus we have s as the representative.
- One drawback of the approach is that it may generate many permutations ('flood' the state space). However, standard exhaustive checking will eventually reach all permutation states and so there is nothing to lose. In fact, the algorithm has the benefit that one need not check the invariant for all symmetric states, nor compute the enabled operations and their effects.

Let us now examine a simple B machine of a stack to illustrate the effect of symmetry reduction by permutation flooding in ProB. The machine has two operations: *push*, to add data to the top of the stack, and *pop*, to remove the data on the top of the stack. The encoding of the machine is given in Figure 2.10.

The machine uses a single deferred set, called *DATA*, which models the data that can be stored on the stack. The constant, *length*, indicates the maximum number of data elements the stack can store. For the convenience of the presentation, this is set to a low number ($length = 2$). In addition, there is a single variable, *contents*, which is a sequence of *DATA*, which models the stack.

[1]This permutation function forms one of the research contributions of this thesis, and therefore is presented in detail later in Section 4.2.

MACHINE
Stack
SETS
DATA
CONSTANTS
length
PROPERTIES
$length \in \mathbb{N} \wedge$
$length = 2$
VARIABLES
contents
INVARIANT
$contents \in seq(DATA)$
INITIALISATION
$contents := []$
OPERATIONS
$push(d) \mathrel{\widehat{=}}$
PRE
$d \in DATA \wedge size(contents) < length$
THEN
$contents := d \rightarrow contents$
END;
$d \leftarrow pop \mathrel{\widehat{=}}$
PRE
$size(contents) > 0$
THEN
$contents := tail(contents) \;||$
$d \leftarrow first(contents)$
END;
END

FIGURE 2.10: A Stack machine

Standard model checking of this machine in PROB generates the state space presented in Figure 2.11. Note that the cardinality of deferred sets is set to 2, to keep the state space small and aid description. As can be seen, the state space has 8 distinct states, including the triangular node representing the uninitialised machine[2]. The label of each state represents the current value of *contents*. For clarity, the parameter of the *push* operation is hidden, and the *pop* operation is not depicted (this does not affect the set of reachable states).

[2]The state space PROB visualises actually includes an extra node immediately after the triangular node in Figure 2.11, to show the values of machine constants.

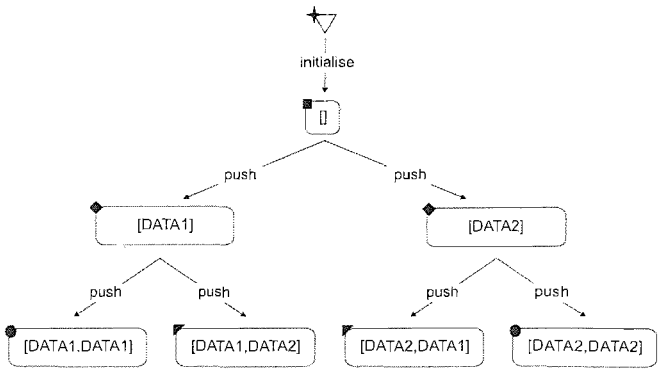


FIGURE 2.11: State Space of Stack machine (only push operation shown for clarity)

The use of the deferred set, *DATA*, gives rise to symmetric states. Symmetric states in Figure 2.11 are indicated by having the same black shape in their upper left corner. The permutation function used in permutation flooding is able to identify these symmetries. Figure 2.12 shows the state space reached by permutation flooding, where representative states are highlighted by dark shadows. Observe that, after the initialisation of the machine, model checking adds an element of data to the stack, i.e., *contents* = $[DATA1]$, and subsequently permutes the abstract elements of the deferred set to generate a symmetric state (*contents* = $[DATA2]$), which is also added to the state space. Next, the procedure computes the two non-symmetric successors of $[DATA1]$, which are both taken as representatives, and their corresponding symmetric states are added to the state space. To the end, model checking reaches and checks the invariant of five states, compared to eight states in Figure 2.11, and will have generated three permutation states.

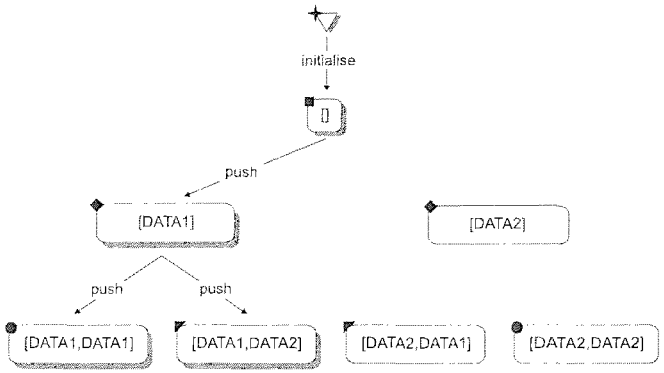


FIGURE 2.12: State Space of Stack machine using Permutation Flooding (only push operation shown for clarity)

Empirical results presented in Leuschel et al. [2007] highlight the effectiveness of permutation flooding, where speedups over standard model checking in PROB can exceed an

order of magnitude. Further experimental data can be found in Chapter 5, which provides a comparison of permutation flooding with the techniques for symmetry reduction described in Chapter 4 of this thesis.

2.8.6 Recent Development in ProB, II: Symmetry Markers

Since the development of the technique for symmetry reduction in B, presented later in Chapter 4, another method to exploit symmetries in B has been devised, called *symmetry markers* [Leuschel and Massart, 2007]. The technique involves efficient *approximate* verification of B models and is inspired by the success of Spin's bitstate hashing approach, presented in Holzmann [1998]. The idea is to efficiently compute 'markers' for states, such that symmetric states are guaranteed to have the same marker (hence, *symmetry marker*). However, this property may not hold in the contrary situation; non-symmetric states may have the same symmetry marker. Hence, the strategy is *approximate*: it may fail to discover a state that violates the invariant, or deadlocks.

As with permutation flooding, the symmetry marker approach is described with reference to a permutation function over deferred sets, which identifies symmetric states (described in detail in Section 4.2 of this thesis). Leuschel and Massart [2007] also describe the necessary conditions of a B machine, in which the technique achieves verification. These conditions may, however, imply a system exhibits less symmetry, and so there is less symmetry to exploit[3]. Experimentation indicates that symmetry reduction via symmetry markers may observe speedups in verification time over standard checking in ProB, exceeding two orders of magnitude.

To compute the symmetry marker for a given state, one treats states as graphs and makes use of the concept of *graph invariants*, see Definition 2.7. Such invariant conditions will hold for two graphs if they are symmetric, but may also hold for non-symmetric graphs. As an example, one can compute for some graph a list of vertex degrees in ascending order. In practice, there is a trade-off between the number of graphs distinguished by some invariant condition ϕ (its *precision*), and the complexity of computing ϕ. The symmetry marker approach makes use of conditions that are easy to compute, yet can be very precise.

An informal description is now given for the computation of a symmetry marker, for a B state, s. (A formalisation of the procedure can be found in [Leuschel and Massart, 2007, Pages 7-9].)

- First, analyse the graph g, that represents the state s, and compute structural information for each element d of a deferred set in s, which is invariant under

[3]For example, Leuschel and Massart [2007] describe that verification can be achieved by replacing deferred sets (containing abstract elements) with enumerated sets (the elements of which are given a distinct name, and are not abstract). However, in doing so, symmetries induced by deferred set elements can no longer be exploited.

symmetry. Achieve this by computing the multiset of *paths* in g, which lead to an occurrence of d.

- Second, for the values of a state, replace deferred set elements by the structural information computed above.

Let us now examine a simple example to illustrate the use of symmetry markers. Consider a B machine that uses one deferred set, $DS1$, and contains three variables $v_1 \in DS1$, $v_2 \in \mathbb{P}(DS1)$ and $v_3 \in DS1 \leftrightarrow DS1$. Let state s_1 have the value $v_1 = d_1$, $v_2 = \{d_1, d_2\}$, and $v_3 = \{d_1 \mapsto d_3\}$. Let us also assume $d_1, d_2, d_3 \in DS1$. Then, the symmetry marker for s_1 is depicted in the rounded box on the right hand side of Figure 2.13. The left hand side shows the graph of *paths*, representing the values used in the state. (The braces, $\{| \ldots |\}$, indicate the use of multisets, and 'el' is used to indicate the constituent elements of values.)

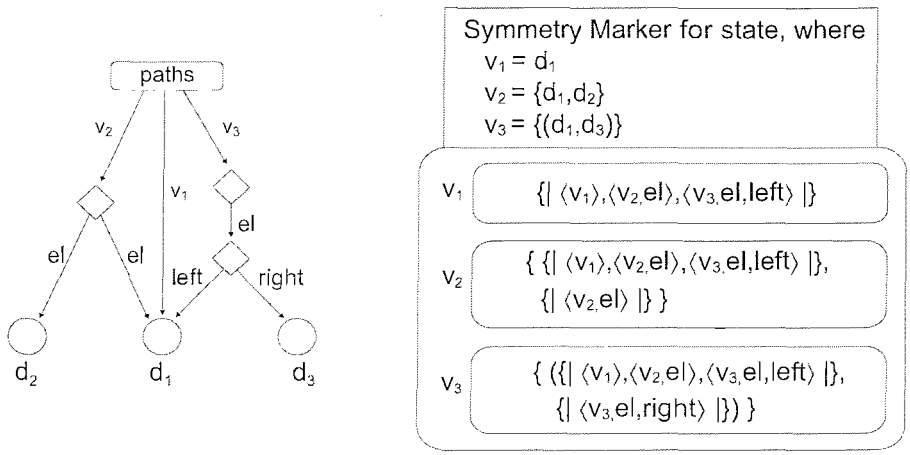


FIGURE 2.13: The symmetry marker for state s_1

As can be seen, the graph representing the state contains the deferred set elements d_1, d_2 and d_3 – each of which is identified (reached) by a set of paths from the initial *paths* node in Figure 2.13. Accordingly, d_1 can be identified by the paths $\langle v_1 \rangle$, or $\langle v_2, el \rangle$, or $\langle v_3, el, left \rangle$. Let us denote this set of paths, p_1. Given that $v_1 = d_1$, symmetry marking then represents the value of v_1, as p_1. Continuing in this way, the paths p_2, reaching d_2, and the paths p_3, reaching d_3, enable one to find the symmetry marker for the entire state, such that $v_1 = p_1$, $v_2 = \{p_1, p_2\}$ and $v_3 = \{(p_1, p_3)\}$.

Observe that by replacing deferred set elements by the structural information identifying them, symmetry marking can exploit symmetries. For instance, a state symmetric to s_1, where $v_1 = d_2$, $v_2 = \{d_2, d_1\}$, and $v_3 = \{(d_2 \mapsto d_3)\}$ (i.e., given s_1, permute d_1 with d_2), has the same symmetry marker as s_1. Thus, model checking need not check it, or any state reachable from it.

There are cases, however, where symmetry markers fail to distinguish between non-symmetric states. For instance, this may occur when a B machine uses a relation over a single deferred set, e.g., given a machine containing a single relation, $r \in DS \leftrightarrow DS$, and $ds_1, ds_2, ds_3, ds_4 \in DS$, then a state, where $r = \langle\{ds_1 \mapsto ds_2, ds_2 \mapsto ds_3, ds_3 \mapsto ds_4, ds_4 \mapsto ds_1\}\rangle$, has the same symmetry marker as a different state, where $r = \langle\{ds_1 \mapsto ds_2, ds_2 \mapsto ds_1, ds_3 \mapsto ds_4, ds_4 \mapsto ds_3\}\rangle$. However, it should be noted that the discovery of any invariant violation or deadlocked state is guaranteed to exist in the original state space. Therefore, an effective use of symmetry marking is that of falsification.

Continuing from the example in the previous section, let us apply symmetry markers to the Stack machine given in Figure 2.10. Recall that for the convenience of the presentation, both the length of the stack, and the size of deferred sets is set to 2. Figure 2.14 depicts the symmetry marker for a state that models a stack containing two different elements of data.

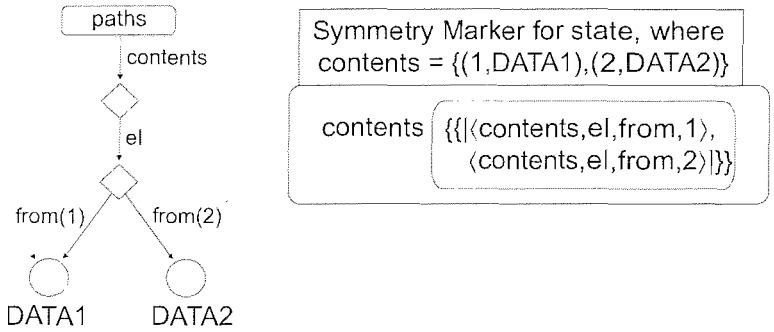


FIGURE 2.14: The symmetry marker for a state in the Stack machine

The labels, $from(1)$ and $from(2)$, indicate the value of the natural number from which a deferred set element is reached. As would be desired, the symmetry marker illustrated is the same as that for the symmetric state where $contents = \{(1, DATA2), (2, DATA1)\}$, but differs from the markers for states where $contents = \{(1, DATA1), (2, DATA1)\}$ or $contents = \{(1, DATA2), (2, DATA2)\}$ (since only 1 $DATA$ element is used in these states; a fact that is recorded in multisets). The state space obtained by model checking in ProB with symmetry markers, is given in Figure 2.15. Unlike permutation flooding, this technique identifies five states only, each of which is a representative.

The next chapter of this thesis presents the initial research contributions made towards improving the process of model checking via state space reductions. The techniques involve reducing the size of state spaces, obtained after model checking, while retaining certain useful information. The reduced state spaces are then visualised. The premise is that few model checking tools focus on improving such aspects, despite evidence that visual feedback can constitute an important part in the understanding of a system.

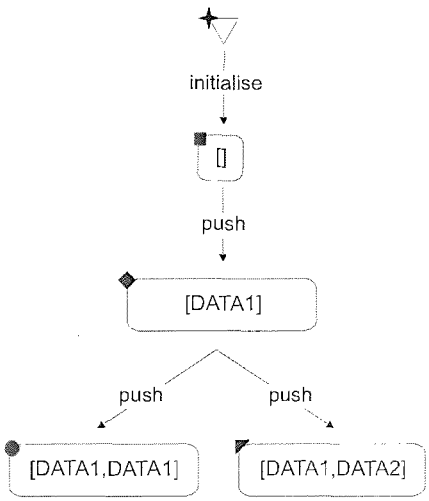


FIGURE 2.15: State space of Stack machine using Symmetry Markers (only push operation shown for clarity)

Chapter 3

Visual State Space Reduction

3.1 Introduction and Motivation

Some model checkers provide a graphical view of the state space the model checker has already explored; ProB does so with the use of a graph drawing package called, *dot* [DOT]. This feedback can be very beneficial to the understanding of the specification since human perception is good at identifying structural similarities and symmetries [Ham et al., 2001]. This feature works well for small state spaces, but in practice specifications under analysis can often consume thousands of states, which limits the usefulness of the graph.

Consider the *phonebook* machine presented in Section 1.4 (Figure 1.3), with the modification of removing the *size_phonebook* operation, for simplicity. Therefore, it specifies three operations that allow one to *add*, *delete* and *lookup* entries in the phonebook. The full state space of this example (with *Name* and *Code* set to cardinality 3) has 65 states and 433 transitions.

As can be seen in Figure 3.1, the visualisation of the state space in ProB is possible, but is difficult to grasp by humans and certain other useful aspects of the state space are not easily identified in the visualisation. For example, it is difficult to spot individual operations or that one can perform at most three consecutive calls to the *add* operation.

It turns out that there are very few tools and techniques that address the problem of rendering aspects of a system state space in ways more suitable for human understanding; it is typical to only provide direct visualisations of an encoded system, e.g., an interpretation tree in the Alloy Analyser [Jackson, 2006a]. Therefore, this chapter investigates various ways to improve visualisations of state space of B machines. A selection of the results in this chapter appear in [Leuschel and Turner, 2005].

This chapter is organised as follows. The techniques that have been developed and implemented into the ProB tool-set are described in Sections 3.3 and 3.4. An empirical

FIGURE 3.1: Phonebook machine - Original State Space.

analysis of their performance is given in Section 3.5, followed by a description of complementary extensions to the methods, in Section 3.6. Finally, a review of future work is presented in Section 3.7.

3.2 Background

A background to visual state space reduction is now presented. First, Section 3.2.1 presents several successful ways of encoding a state space and reducing its size, and discusses the associated issues. Subsequently, Section 3.2.2 reviews the various aspects of a visualisation that make it more understandable to the human eye.

3.2.1 Reducing the Size of a State Graph

A standard structure used to depict a state space, representing system behaviour, is the Kripke structure – introduced in Definition 1.1. An alternate formation is a Finite State Automaton [Cassandras and Lafortune, 1999], which enables system transitions (e.g., B operations) to be encoded naturally via labelled edges, and system states encoded as states of the automaton. In addition, these structures have a range of techniques for reducing their size. A key difference between Kripke structures and Finite State

Automata is that the latter lacks the notion of atomic propositions. A formal description of Finite State Automata is given in Definition 3.1, which is followed by a selection of approaches for reducing their size.

Definition 3.1. A finite state automaton is a four-tuple $M = (Q, A, q_0, \delta, F)$, where Q is the set of states, A is the set of transition labels (the alphabet), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times (A \cup \varepsilon) \times Q$ is the transition relation.

- $L(M)$ is the *language* recognised by M, defined as the set of all sequences of letters of the alphabet that define a trace from q_0 to a final state.
- Two automata, M_1 and M_2, are *equivalent* iff $L(M_1) = L(M_2)$.
- An automaton (Q, A, q_0, δ, F) is *deterministic* (DFA), if δ is a partial function over $Q \times A$. If $\delta \subseteq Q \times A \times Q$, then it is *nondeterministic* (NFA). Finally, it is called *nondeterministic with ε-transitions* (εNFA), if there are no extra restrictions on δ.

DFA Minimisation: Given a DFA or NFA, there exists a smallest DFA accepting the same language. Regarding a state graph, which can be encoded naturally as an NFA (operations in B may exhibit nondeterminism), this can be achieved via two steps. First, it must first be converted into a DFA using a well known algorithm often known as *subset construction*. Then, it can be *minimised* using another well known algorithm, which works by finding indistinguishable states that can be merged into a single state; starting from the fact that all non-final states are distinguishable from all final states. Both algorithms can be found in Aho et al. [1988].

εNFA Minimisation: NFA can be smaller than a DFA by an exponential factor [Meyer and Fischer, 1971], and so an ideal strategy would directly minimise the NFA state graph of a B specification. One such algorithm [John, 2004] is based on minimizing εNFA. Unfortunately, this performs best when the automaton contains a large number of ε-transitions – which are not present in graphs of B systems. Therefore, this technique seems not to be very relevant to the goal.

Minimizing Finite State Automata Is Computationally Hard: NFAs and DFAs accept the same languages, i.e., the set of regular languages. However, an NFA can be exponentially smaller than an equivalent DFA. This suggests a greater visual state space reduction would be gained through the use of NFAs. Unfortunately, this is not so simple since the NFA must be minimised prior to visualisation, for which there is no known polynomial time algorithm. Malcher [2004] investigates this problem by looking at the amount of nondeterminism required to preserve the efficiency of minimizing a finite state automaton. It finds that by restricting each accepting path of NFAs to a fixed number of nondeterministic transitions (e.g. one or two), the time complexity of certain decidability problems can be reduced from non-polynomial to polynomial time. This provided hope that similar improvements in time complexity would be gained for minimizing restricted NFAs. However, it is proven to be NP-complete. Therefore, NFA minimisation was eliminated as a tractable option for reducing the size of state space graphs.

An alternate approach to minimisation is a non-minimal reduction in size, while retaining language equivalence, as demonstrated in Ilie and Yu [2002]. This method is based on the idea of merging indistinguishable states, such as in DFA minimisation. Essentially, an equivalence relation is computed on the states, such that non-equivalent states are made distinguishable, and equivalent states may be indistinguishable. The reduced automaton therefore may not be minimal, but may be significantly smaller than the original NFA.

Non-Equivalent Reduction: The previous techniques all generate smaller automatons that recognise the same language. However, a useful visual reduction need not preserve language equivalence; the user may not be interested in the entire state graph, but only certain properties of it. For instance, a user may only want to know that all states in an automaton have a certain outgoing transition. Figure 3.2 shows an example of this: the left hand side shows the original automaton and the right side is the reduced automaton.

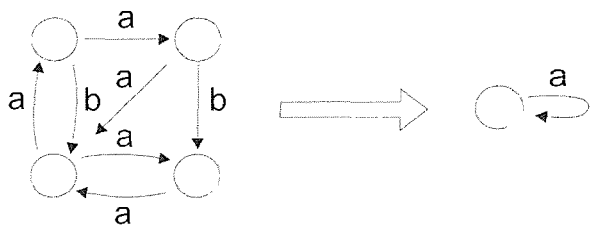


FIGURE 3.2: Non-equivalent reduction; all states have an '**a**' transition.

Non-equivalent reduction has the advantage of being capable of reducing the size of the original automaton by a greater amount than the techniques described previously, e.g. by order(s) of magnitude; all that matters is what information a user finds useful. The next section complements this by reviewing aspects of a visualisation many users deem useful.

3.2.2 Features of a Good Visualisation

Effective visualisations should also consider aspects of visualisations in general, which can complement an algorithm that produces smaller graphs. Several of these that Dulac et al. [2002] recognise as key are now presented.

Redundant Recoding: This is a principle taken from programming language design research, where the same information is specified in more than one way. Used effectively, this technique highlights only the useful sets of information in a representation.

Elision: This refers to the ability to temporarily hide certain information that is not of immediate interest, for instance by rendering it in a different color.

Layout: The layout is important since it can highlight certain elements of the representation. For example, it may draw attention to symmetries within a state graph. The graph visualisation survey by Herman et al. [2000], emphasizes two important aspects

that affect layouting, the first of which concerns the notion of predictability; given two similar graphs, a run of a layout algorithm on each should not lead to radically different representations. It is desirable to preserve the *mental map* of the user.

The second aspect regards the size of a graph. Few layouting techniques can claim to deal effectively with thousands of nodes even though graphs of this size appear frequently in application domains, including model checking. The size of a graph can make a normally good layout algorithm completely unusable. Therefore many visualisation techniques attempt to reduce the size of the graph to display. An overview of the important techniques are documented in Herman et al. [2000], one of which appears to be more relevant to this research; that is, *clustering*. This method generally assigns nodes of a graph that satisfy some condition, e.g., defined by an equivalence relation, to the same *cluster*. Edges are then made between clusters to represent the relation between the nodes of one cluster with those of another. No tool has been found that clusters states of a state space similar to one generated by model checking, and it appears further research into the area could uncover some interesting results.

The next two sections introduce two new algorithms devised to improve the visualisation of the state space of a B machine, generated by ProB. Improvements entail reductions in the size of a state space, while utilising background information presented in the preceding section.

3.3 The DFA-Abstraction Algorithm

This is the first of two algorithms developed for use in the ProB model checker. It uses another notion of transition systems; the labelled transition system (LTS) [Keller, 1976]. The state space generated by ProB can be viewed as non-deterministic labelled transition system, where the edges are labelled with terms of the form $op(a_1, \ldots, a_n)$ where op is the name of the operation that has been applied and $a_1, \ldots, a_n$ are the arguments of the operation. The formal definition of an LTS is now given:

Definition 3.2. An LTS is a 4-tuple (Q, Σ, q_0, δ) where Q is the set of states, Σ the alphabet for labelling the transitions, q_0 the initial state and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

Here, $q \rightarrow_a q'$ denotes that $(q, a, q') \in Q$. This is extended to sequences of transitions so that $q \rightarrow_{a_1,\ldots,a_k} q'$ denotes the fact that there exists a sequence of states $q_0, \ldots, q_k$ such that $q_0 = q$, $q_k = q'$ and $q_i \rightarrow_{a_i} q_{i+1}$. The set of *reachable states* of an automaton is defined to be the set $\{q \in Q \mid q_0 \rightarrow_\gamma q \text{ for some sequence of states } \gamma\}$. Finally, the *traces* of an automaton L is the set of sequences $traces(L) = \{\gamma \mid q_0 \rightarrow_\gamma q \text{ for some } q \in Q\}$.

This algorithm reduces the complexity of an LTS by abstracting away certain details of the labelling function. For example, the user may not be interested in seeing (all) the arguments of (all) the operations. The algorithm is defined in the following:

Definition 3.3. An *abstraction function* for an LTS (Q, Σ, q_0, δ) is a function α from Σ to some new alphabet Σ'. Then, the α-abstraction of the LTS is defined as a new LTS $(Q, \Sigma', q_0, \delta')$ where $\delta' = \{(q, \alpha(a), q') \mid (q, a, q') \in \delta\}$.

The α-abstraction on its own is not yet very useful, as it does not diminish the number of states (even though the number of transitions may have reduced). The first thing that comes to mind in that respect is the classical determinisation and minimisation algorithms for finite automata. Moreover, a finite LTS, such as those produced by model checking, can be viewed as a NFA simply by marking all states as final states. Therefore, the original NFA can be converted into a DFA. Given that the resulting automaton contains only finite states, it is already minimal and need not be applied to classical DFA minimisation. This is exactly what has been implemented in the *DFA-Abstraction* Algorithm, which has been integrated into the ProB tool-set. In summary, the *DFA-Abstraction* computes:

i.) the α-Abstraction of an LTS

ii.) then determinises the resulting LTS

The algorithm is shown on a small example in Figure 3.3.

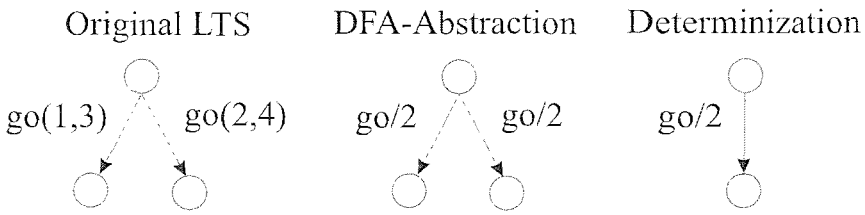


FIGURE 3.3: Illustrating the DFA-Abstraction Algorithm.

The algorithm was mainly used as a control: something to which other algorithms could be compared. However, it was also known that it had the potential to collapse symmetrical subgraphs. Testing shows it is very useful in some cases, while in other cases it increases the size of the graph (a known observation of the NFA to DFA conversion, but which is rarely observed in practice).

DFA-abstraction of the phonebook: An example of its effect is shown in Figure 3.4 (the full state space is given in Figure 3.1). Clearly there is a large reduction whose result is a graph far clearer to an observer. One can see that only three phonebook entries are permitted, and also that *add* and *delete* operations change the state.

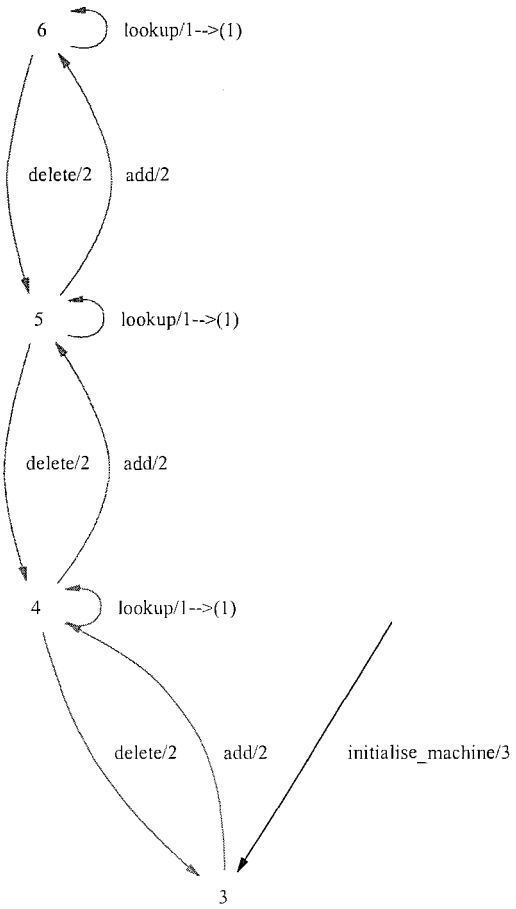


FIGURE 3.4: Phonebook machine - DFA-Abstraction.

In general, every node in the reduced graph corresponds to a set of states of the animated B machine. Note that although operation argument information is lost during α-abstraction, the DFA conversion algorithm preserves possible trace equivalences. This, and the fact that determinisation may merge multiple B states, means that if a node in the DFA-abstracted graph has an outgoing edge corresponding to some operation, it does not guarantee this operation can be applied in *all* B states covered by this node. Therefore, this is indicated in the implementation through the style of the edge; if all covered B states can perform the operation, the edge is solid, otherwise the edge is dashed.

3.4 Merge States with same Outgoing Transitions

This technique was devised after studying a collection of graphs produced by ProB. It works by merging all states with the same outgoing transitions and so it may produce an automaton that is not equivalent (with respect to traces of operations) to the original one. However, the technique can achieve a big reduction in the size of the automaton while preserving information about which B operations are enabled in a particular state (i.e. the traces of length 1). Before giving its definition, the signature of a state is first presented, which represents the operations (i.e., transition labels) that can be performed in that state.

Definition 3.4. Let (Q, Σ, q_0, δ) be an LTS. Then define the signature of a node $q \in Q$, denoted by $signature(q)$, as: $signature(q) = \{a \mid q \rightarrow_a q' \text{ for some } q' \in Q\}$.

Definition 3.5. Let (Q, Σ, q_0, δ) be an LTS. The *Signature-Merge* of the LTS is defined to be a new LTS $(Q^s, \Sigma, q_0^s, \delta')$ where $Q^s = \{signature(q) \mid q \in Q\}$, $q_0^s = signature(q_0)$, and $\delta^s = \{(signature(q), a, signature(q')) \mid (q, a, q') \in \delta\}$.

The effect of a signature-merge is to merge all states which have a common signature. This ensures that at least for traces of length 1 there is no loss of precision. This alone may not seem so useful, however some more properties add to its benefit. Let there be some original LTS L, and the signature-merge LTS of L, called L_S. Then, if one state in L_S has no out going transitions, corresponding to a machine that deadlocks, then L also deadlocks. Also, the set of traces defined by L are a subset of those of L_S – which means any trace that cannot be performed on the reduced LTS cannot be performed in the original LTS.

Similar to the last algorithm, the implementation first computes the α-abstraction before applying the signature-merge.

Signature-Merge of the phonebook: The method is applied to the phonebook example and is shown in Figure 3.5. As can be seen, there is a large reduction that is far more digestible to a human. Also, note that modifying the machine so that it can store more phone numbers produces no change in the reduced graph, since this generates no new signatures. So, in principle, one can even visualise the machine for an unbounded set *Name*. This is not the case for the DFA (where if we allow 100 entries the DFA will have 100 nodes). However, some of the precision of the DFA visualisation is lost: one can no longer spot how many entries can be added; all that can be seen is that one can add at least two entries, but not exactly how many. Still, the signature based approach has managed to keep relevant information. For example, it is still obvious from the graph that entries can only be looked up or deleted after adding an entry, and that it is possible to reach a state where it is no longer possible to add entries.

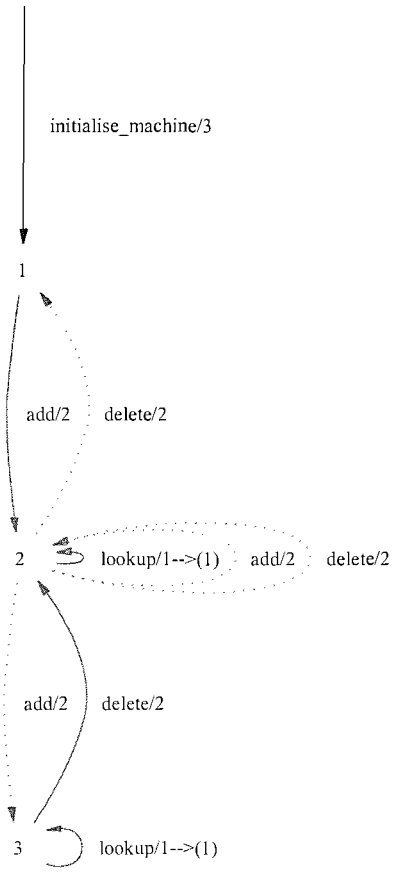


FIGURE 3.5: Phonebook machine - Signature-Merge

Although merging states with the same signature can be valuable, the visualisation may become overly complicated. For instance, consider two states each with one, identical outgoing transition; they have the same signature. If these transitions lead to successor states with different signatures, then both transitions must be shown in the reduced graph. Therefore, this work improves the interpretation of the graph using edges with different styles. That is, if *all* states with the same signature (represented by one state in the reduced graph) have an operation that leads to a set of next states, each sharing the same signature, then this operation is said to be *definite*; indicated by a solid edge in the graph. If not, then the edge is dashed. A concrete example of this can be seen in the example in Figure 3.5; state **2** has dotted *delete* edges. This is because a state with the same signature as **2** in the original graph can either *delete* a phone number to result in a state with:

- the same signature as **2**, e.g., when the phonebook contains two numbers, or
- a different signature, i.e., the empty phonebook.

3.5 Empirical Evaluation

The evaluation in the last sections suggest that the algorithms are often efficient at deriving informative graphs. Some more comprehensive examples can be found in Leuschel and Turner [2005]. An empirical evaluation of the algorithms is now presented, to show concrete numbers on the size reductions achieved.

Tables 3.1 and 3.2 show key statistics obtained after applying both the *Signature-Merge* and the *DFA-Abstraction* algorithms on 47 arbitrary state spaces that had been previously model checked with the ProB model checker: Table 3.1 shows percentages of states and transitions compared to the original state space[1] and Table 3.2 shows the overall statistics. The exact numbers of states and transitions for each state space, are given in Appendix B.

Signature-Merge produced the best results, reducing the number of states by at least 85% and the number of transitions by at least 87% in half of the state spaces tested. Moreover, 80% of the graphs had at least 43% fewer states and 59% fewer transitions than the original. The best case produced a graph with approximately 99% fewer states and transitions. The *DFA-Abstraction* technique also gave good results; half of the graphs having at least 40% fewer states and at least 64% fewer transitions, and the best case again reduced the number of states and transitions by 99%. The worst case didn't follow the trend of producing a reduction, but in fact increased the size of the original graph by approximately ten times. A result like this should not be unexpected since, after all, it is possible for a DFA to be exponentially greater in size than an equivalent NFA. However, only a small proportion of the applications of this technique had this effect; approximately 80% of the tests produced a reduction.

TABLE 3.1: Sizes as percentage of original state space

	Sig. Merge		DFA-Abstr.	
Machine Name	States	Transitions	States	Transitions
Ambulances	0.24	0.02	0.86	0.10
Baskets	6.33	2.02	21.52	8.59
B_Clavier_code	100.00	42.11	133.33	42.11

[1]Some of the machine names in Table 3.1 appear more than once, however their implementations differ.

Table 3.1 Sizes as percentage of original state space (continued)

Machine Name	Sig. Merge		DFA-Abstr.	
	States	**Transitions**	**States**	**Transitions**
bibliotheque	73.33	58.49	93.33	75.47
B_Site_central	60.00	12.50	80.00	12.50
CarlaTravelAgency	9.09	30.09	60.61	78.76
CarlaTravelAgencyErr	13.33	43.17	67.5	71.22
countdown	0.13	0.10	7.97	7.77
Cruise	29.54	18.19	1203.97	901.35
CSM	83.12	86.60	101.30	100.00
DAB	40.00	4.88	80.00	7.32
dfa	75.00	57.14	150.00	100.00
dijkstra	42.86	33.33	100.00	66.67
DSP0	12.24	10.61	16.33	12.12
Fermat	11.76	3.70	58.82	20.99
FinalTravelAgency	0.93	0.57	7.69	6.12
FunLaws	1.95	0.63	14.79	6.49
FunLaws	4.28	2.45	20.23	15.39
GAME	8.97	5.30	32.79	20.45
GSM_revue	36.36	28.57	63.64	50.00
Inscription	25.93	16.03	33.33	19.08
inst_adapted	1.07	0.41	17.17	6.68
Jukebox	15.00	4.53	1225.00	616.83
Level0	0.26	0.03	1.43	0.16
m0	100.00	99.98	150.77	150.29
Main	100.00	100.00	150.00	100.00
mm0	3.55	2.44	43.65	40.52
monitor	9.88	3.59	39.51	18.90
phonebook	6.15	1.62	9.23	2.31
Queues	42.86	22.22	57.14	22.22
Results	66.67	45.45	83.33	45.45
Rubik2	0.09	0.10	100.03	100.00
RussianPostalPuzzle	2.04	1.71	27.21	22.33
rw	90.00	94.59	105.00	100.00

Table 3.1 Sizes as percentage of original state space (continued)

Machine Name	Sig. Merge		DFA-Abstr.	
	States	**Transitions**	**States**	**Transitions**
scheduler	22.22	14.05	33.33	20.66
SensorNode	60.00	18.18	80.00	18.18
SeqLaws	15.79	22.41	71.05	101.72
SetLaws	1.23	0.72	17.40	11.78
station	25.00	14.61	28.57	14.61
Teletext	16.00	5.71	48.00	35.71
Teletext	21.43	9.84	107.14	100.00
TheSystem	14.04	43.09	72.81	69.92
TransactionsSimple	16.79	33.33	76.34	83.01
TravelAgency	9.09	34.55	59.66	62.83
TravelAgency_trace_check	0.33	0.93	38.75	40.92
TravelProB	0.80	0.26	3.83	0.95
UndefinedFunctions	29.41	13.99	70.59	37.82

Statistic	Sig. Merge		DFA-Abstr.	
	States	**Transitions**	**States**	**Transitions**
Minimum	0.09	0.02	0.86	0.10
Maximum	100.00	100.00	1225.00	901.35
Median	15.00	12.50	59.66	35.71
Average	27.77	22.23	107.76	73.33
80th Percentile	**56.57**	**40.6**	**100.02**	**96.6**
Std. Dev.	31.05	27.95	241.50	155.05

TABLE 3.2: Statistics from results in Table 3.1

3.6 Complementary extensions

Four new complementary extensions have been added to further improve the visualisations, and are described here.

3.6.1 Diminishing the Abstraction function

Since the work of Leuschel and Turner [2005], the algorithms have been made more precise by diminishing the α-abstraction such that it only abstracts away certain arguments. This is guided by the user, to give more control over the reductions possible.

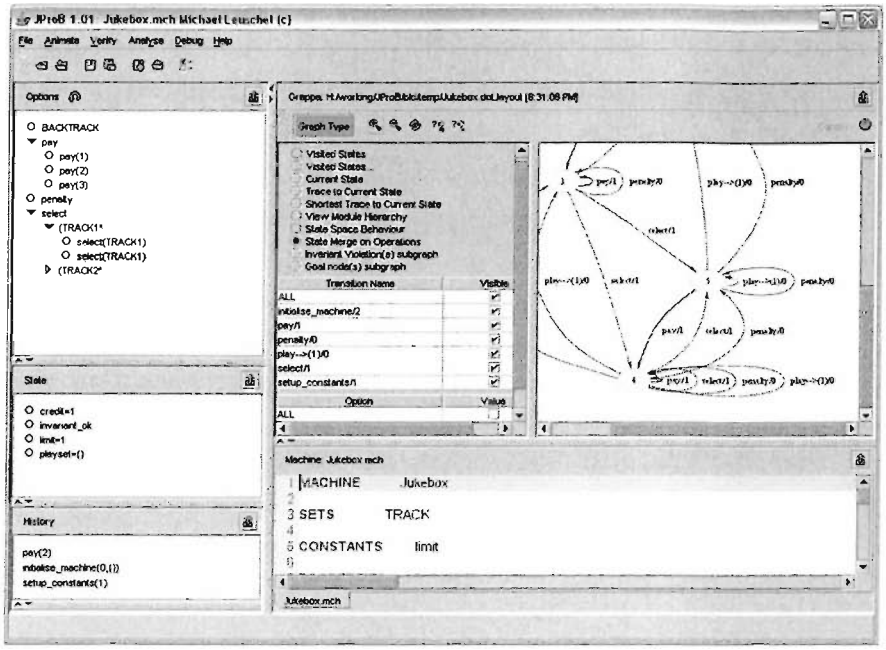


FIGURE 3.6: Screenshot of Java version of ProB

3.6.2 Integrated Java/Swing Visualiser

Figure 3.6 shows a version of PROB that has been developed using Java to take advantage of its cross platform compatibility and rich graphical user interface library. Various panes in the main window present the user with information relating to the current state; including the variables and values of the current state, a history of operations executed, a hierarchical expansion of enabled operations (top left pane) and a state space visualisation. There is also an integrated specification editor to facilitate any changes necessary. As can be seen in the central pane of the screenshot, the user has several choices of visualisation to choose from, including DFA-Abstraction and Signature Merging. Certain visualisations also allow operations to be selectively removed from the visualisation, e.g., to remove self loops and improve clarity.

3.6.3 User Defined Constraints

Through previous experience gained with model checkers, it was proposed that a better understanding of the system might be gained if the user were able to directly query the state space. Therefore the model checking tool was extended by enabling the user to define constraints on system variables and on values of operation arguments, and to subsequently view a graph of all states in which these hold, and the relationship between them, if any. This is generally useful when the user is interested in exposing some subtle aspect of the state space, which a more general algorithm would be unlikely

to reveal without user intervention. It should be noted that the effectiveness of reducing state spaces using this technique depends largely on the user's literacy in the specification language and their understanding of the system; however its potential makes it a feature worth keeping and extending in the future. As mentioned, it is also possible for the user to selectively turn off visible operations in the visualisation, to further reduce the size of the graph: see tick boxes in Figure 3.6.

3.6.3.1 Subgraphs

Another method of reducing the size of the graph is to show only part of it: a subgraph. Hence, the system also has the option to view the subgraph that connects one or more states (determined either interactively via the visualisation, or via an options menu). This is particularly useful when one wants to view all paths from the initial state that lead to a state that violates the invariant of a B specification.

3.7 Summary and Future Work

Tables 3.1 and 3.2 show some encouraging results. The often considerable reduction of the original state space by the DFA-Abstraction algorithm can be explained by its nature of finding regular behaviour amongst abstracted transitions, and collapsing duplicated instances of it into a single path (most of the time). A good example of this is shown in the original Phonebook example (Figure 3.1) and the DFA reduced Phonebook example (Figure 3.4). The graph generated remains very useful since all of the behaviour of the original state space is captured in a definite finite automaton.

The Signature-Merge algorithm gives better reductions than the DFA -Abstraction method, producing non-equivalent graphs to the original that do not show explicitly the exact behaviour. However, they remain useful since they can still be used to check many properties, such as checking if a trace may exist in the full state space. This is because a system's behaviour is a subset of the behaviours described in its Signature-Merge graph.

Future Work: The signature of a node can be viewed as all traces of length 1 that can be performed from that node. An improvement can be made by extending the notion of a signature and comparing all traces of length 2,3,..., where the choice is guided by the user. This enables more detailed trace information to be read from graphs generated by the signature-merge method.

A converse improvement to that of diminishing α-abstraction (Section 3.6.1), and therefore increasing precision in graphs, is to make it α *less* precise to achieve more reduction

in several ways. For instance, it could be made more aggressive, mapping several operations together, which maybe the user is not interested in. Another option is to modify Signature-Merge so that instead of merging nodes that have exactly the same signature, they could be merged if the signatures are sufficiently similar. On the other hand, it may be useful to combine both of these approaches: the user could type a certain number as a target for the ideal number of nodes and then the graph is progressively made less or more precise to approach that number.

The visual reduction strategies presented in this section take as input a state space produced from model checking a B machine. The work inspired the line of research presented in the next section, which involves reducing the size of a state space *during* its construction, when model checking a B machine; namely, the technique of *symmetry reduction.*

Chapter 4

Symmetry Reduction in ProB

4.1 Introduction

The B-method, introduced in Section 1.4, is a theory and methodology used for the formal development of computer programs. It includes a concise language based on set theory and predicate logic, called B, and is used by industries in a range of critical domains, notably in railway control.

Proof activities in B are usually carried out using the interactive theorem provers, Atelier-B [Ste, 1996] or the B-toolkit [B-C, 1999]. Model checking is a useful, complementary approach that can perform these tasks *automatically*, if bounds are placed on system types; as with the combined B-animation/model checker, ProB.

Recall that a major challenge facing model checking is the problem of *state space explosion*. This is where a linear increase in the size of a specification leads to an exponential increase in the number of states, which the model checker must explore. Thus, checking larger specifications becomes intractable. Much research in model checking focuses on methods to combat this problem, including partial order reduction [Holzmann and Peled, 1994] and abstraction [Clarke et al., 1994]. Symmetry reduction is another approach, which exploits symmetries inherent in the system [Clarke et al., 1999] by constraining the search to representatives of symmetric states; often resulting in significant savings in memory and time. A successful technique relies on a special data type, called a *scalarset*, being introduced into the language of the model checker, to indicate symmetric structures (e.g., the Murϕ Verifier [Ip and Dill, 1993] and SymmSpin [Bosnacki et al., 2002]). This requires the user to indicate symmetries of the model, and is therefore error prone, and compromises the automation of model checking.

This chapter presents an *automatic* method for exploiting symmetries caused by a key component of the B language, the *deferred set*. The culmination of the techniques it uses is a method called *canonical labels*. A description of this technique can also be

found in Turner et al. [2007]. The work uses a very different technique to an alternate strategy called *permutation flooding*, which is introduced in Section 2.8.5. For a thorough reference, see Leuschel, Butler, Spermann, and Turner [2007].

The remaining parts of this chapter are organised as follows. The introduction continues in Section 4.1.1 by describing a component of the B language that gives rise to symmetries, which our techniques exploit. Section 4.1.2 elaborates on the types of symmetries exploited using a detailed example. Presented next, in Section 4.2, are correctness proofs showing that the types of symmetries we exploit are sound. Following this is a description of how the symmetries are exploited, using a graph isomorphism algorithm based on *nauty*. Section 4.3 details how states in B can be represented as graphs. We relate the identification of symmetric states to the identification of isomorphic graphs, in Section 4.4, and present the algorithm for finding isomorphic graphs in Section 4.5. Section 4.6 shows how these techniques are integrated into model checking in PROB, to give the new method for symmetry reduction. Section 4.7 introduces complementary work for symmetry reduction in PROB, called the *canonical labels + symmetry markers* approach, which combines the techniques of canonical labels and symmetry markers (Section 2.8.6), in an attempt to gain the benefits of both methods. Finally, Section 4.8 presents how the new techniques for model checking fit within the architecture of the ProB model checker.

4.1.1 Deferred Sets in B give rise to Full Symmetries

In B there are two ways to introduce sets into a B machine: either as a parameter of the machine, or via the SETS clause. Sets introduced in the SETS clause are called *given sets*. Given sets whose elements are explicitly enumerated are called *enumerated sets*; the other types of sets are called *deferred sets*. Figure 4.1 shows an example of a SETS clause in a B machine, which uses both kinds of given sets.

```
...
SETS
  ExitMsg = {success,fail}; // an enumerated set
  Proc // a deferred set
...
```

FIGURE 4.1: Examples of given sets: *ExitMsg* (*enumerated set*) and *Proc* (*deferred set*)

Deferred sets consist of abstract elements. For instance, *Proc* is a set of abstract processors. The only information known about a given element of such a set is the *identity* of this deferred set. It follows that the permutation of one abstract element for another will have no effect on semantics; no information is gained or lost. Extending this idea, one finds a set containing n elements of *Proc* to be symmetric to another set containing n elements of *Proc*. Indeed, the use of *abstract* elements in a B specification gives rise

to symmetries in data structures used within the system, which our scheme can exploit. This is similar to the symmetries induced by scalarsets [Ip and Dill, 1993, Bosnacki et al., 2002], which are sets of permutable scalar values. As a consequence, our scheme only exploits *full* symmetries of a system and not other types of symmetry such as rotational or reflective symmetries: exploiting these would require new reduction strategies. In related work, [Ip, 1996] describes adaptations of the scalarset type that handles rotational symmetries, and [Donaldson et al., 2005] presents a method for identifying general symmetries within a message passing system, described in PROMELA, through the analysis of a graphical representation of the communication structure of the system, called a *static channel diagram*.

4.1.2 Motivation

Let us indicate the type of symmetry our method exploits by considering a simple B machine of a phone book (a simplified version of Figure 1.3), which is presented below. The machine has three operations: *add*, to add entries into the phone book; *delete*, to remove entries; and *lookup*, to query a person's phone number.

MACHINE
phonebook

SETS
Name;
Code

VARIABLES
db

INVARIANT
$db \in Name \pfun Code$

INITIALISATION
$db := \emptyset$

OPERATIONS
add(n , c) $\widehat{=}$
PRE
$n \in Name \wedge c \in Code \wedge n \notin \mathrm{dom}(db)$

THEN
$db := db \cup \{n \mapsto c\}$
END;

delete(n , c) $\widehat{=}$
PRE
$n \in Name \wedge c \in Code \wedge n \mapsto c \in db$
THEN
$db := db \setminus \{n \mapsto c\}$
END;

$c \leftarrow$ lookup(n) $\widehat{=}$
PRE
$n \in Name \wedge n \in \mathrm{dom}(db)$
THEN
$c := db(n)$
END
END

The machine has two deferred sets, *Name* and *Code*, modelling sets of names and phone numbers respectively. A single variable, *db*, which is a partial function, stores the contents of the phone book; hence, a *Name* can have at most one *Code* associated with it.

An exhaustive search of the state space of this machine requires bounds to be placed on the types used [Leuschel and Butler, 2003]. So, by setting the cardinality of the deferred sets to 2, the full state space has 10 distinct states. Figure 4.2 shows the state space, where the label of each state is the current value of *db* (e.g., {(Name1,Code1)}). For clarity, the parameters of the *add* operation are hidden, and the *delete* and *lookup* operations are not depicted (this does not affect the set of reachable states).

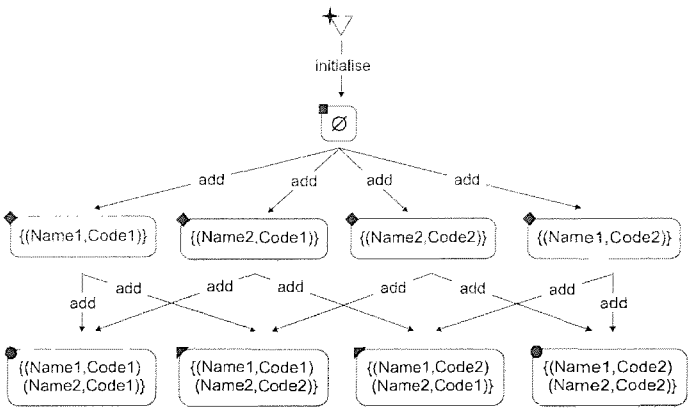


FIGURE 4.2: State Space of Phonebook (only add operation shown for clarity)

The use of deferred sets give rise to symmetries among the states. Informally, let us define two states as symmetric if the invariant has the same truth value in both states, and if there is a permutation between the two states that permutes values of deferred sets. One also requires this permutation to respect the typing, e.g., a *Name* can only be permuted with another *Name*. In Figure 4.2, the state $db = \{(Name1, Code1)\}$ is symmetric to $db = \{(Name1, Code2)\}$ since both are functions and there is the permutation, $\{(Code1, Code2)\}$ between them. Symmetric states in Figure 4.2 have been indicated by identical black shapes in their top left hand corner. As can be seen, three other states are symmetric to $db = \{(Name1, Code1)\}$.

Let us now highlight that permutations that preserve the equivalence of two states must take care with values that are elements of enumerated types (including Booleans and integers). These values may not imply the existence of symmetries as described above. For example, an alternate phonebook machine might specify *Code* as an enumerated set, $Code = \{Code1, Code2, \text{'999'}\}$, where '999' is used to represent the phone number of the emergency services. To ensure this number is reserved, one could strengthen the precondition of the *add* operation with $n \neq$ '999'. Supposing $db = \{(Name1, Code1)\}$ and $db = \{(Name1,\text{'999'})\}$ are two reachable states for this new machine, they should *not* be found to be symmetric, since '999' now has a special meaning, with respect to the *add* operation.

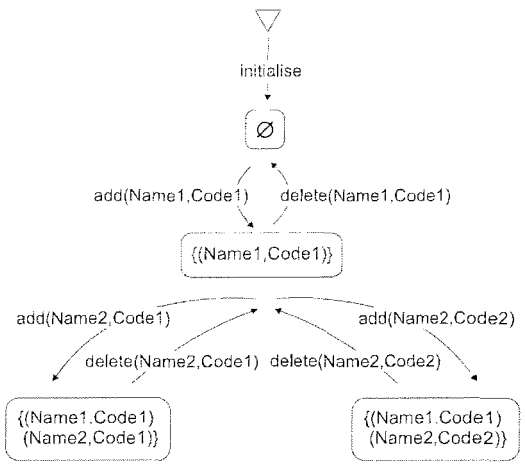


FIGURE 4.3: Reduced State Space of Phonebook

Our approach to symmetry reduction checks only one (unique) representative per symmetry class, using an algorithm for graph isomorphism that permits the permutations described. Figure 4.3, illustrates the symmetry reduced state space for the phonebook machine. The technique exploits symmetries caused by deferred sets, so it is likely to significantly reduce the time and memory required to model check many B specifications, since such sets are commonly used in B. For instance, deferred sets often occur near the top of stepwise refinement chains, in the abstract specifications. In the next

section, the symmetries we exploit are presented formally, in addition to a proof that their exploitation is sound.

4.2 Soundness of State Symmetries

The values of free variables in B expressions and predicates are either elements of given sets (including Boolean values and integers), pairs of values, or sets of values. Let us now describe more formally the symmetry between B states, introduced in the previous section, which involves the existence of a permutation function over the deferred sets of a machine, which respects the typing (it only permutes within the same deferred set). Let us call this function, f. The description has been adapted from collaborative work presented in, [Leuschel, Butler, Spermann, and Turner, 2007, Definitions 1 and 2].

Definition 4.1. Let DS be the set of disjoint deferred sets in a machine M. A *permutation f over DS* is a bijection from $\cup_{S \in DS} S$ to $\cup_{S \in DS} S$ such that $\forall S \in DS$ we have $\{f(s) \mid s \in S\} = S$. f is a fixpoint for enumerated types, including Boolean values and integers. Since values in B can be expressed as elements of given sets, sets of values, or pairs of values, it is possible to lift f to states, using the following rules:

i.) $f(\{val_0, \ldots, val_n\}) = \{f(val_0), \ldots, f(val_n)\}$ (sets)

ii.) $f((val_0, val_n)) = (f(val_0), f(val_n))$ (pairs)

In addition, it is possible to recursively lift f to states of the form $\langle val_0, \ldots, val_n \rangle$, using:

iii.) $f(\langle val_0, \ldots, val_n \rangle) = \langle f(val_0), \ldots, f(val_n) \rangle$

We shall now proceed to justify the use of f to identify symmetric states. First of all, some notational conveniences are presented. We shall represent a B state as a substitution of the form $[v1, \ldots, vn := c1, \ldots, cn]$, where $v1, \ldots, vn$ (denoted V) are the variables in any B expression/predicate and $c1, \ldots, cn$ (denoted C) are the values. Such variables include state variables, machine constants, quantified variables and local operations variables.

For expression E, we write $E[V := C]$ to denote the value of E in state $[V := C]$. This value will be an element of some type constructed from the given sets of a machine. Similarly for predicate P, we write $P[V := C]$ to denote the boolean truth value of P in state $[V := C]$. Most B set operators are defined in terms of other more basic operators and/or set comprehension[1]. This means we can focus on the core predicate and expression syntax as defined in Abrial [1996]. This core syntax is shown in Figures 4.4 and 4.5[2].

[1] For example, $S \subseteq T \Leftrightarrow \forall x.(x \in S \Rightarrow x \in T)$

[2] To simplify the presentation we ignore integer and boolean expressions. These will never be permuted by f. However an integer expression may contain a subexpression of the form $max(S)$ or $card(S)$, where

$$
\begin{array}{rcl}
E & ::= & Var \\
& | & Enum \\
& | & (E, E) \\
& | & E \times E \\
& | & \mathbb{P}(E) \\
& | & \{x \mid x \in S \wedge P\} \\
& | & E(E)
\end{array}
$$

FIGURE 4.4: Core syntax for expressions

$$
\begin{array}{rcl}
P & ::= & P \wedge P \\
& | & \neg P \\
& | & E = E \\
& | & \forall x.(x \in S \Rightarrow P) \\
& | & E \in E
\end{array}
$$

FIGURE 4.5: Core syntax for predicates

The goal is now to prove that permutation function f preserves the evaluation of *any expression or predicate*. This is expressed in Theorem 4.2.

Theorem 4.2. *For any expression E, predicate P, state $[V := C]$ and permutation function f:*

$$
\begin{array}{rcl}
f(E[V := C]) & = & E[V := f(C)] \\
P[V := C] & \Leftrightarrow & P[V := f(C)]
\end{array}
$$

The theorem can be proved by structural induction over expression and predicate terms. The induction is mutual since expressions may contain predicates and vice versa. We now consider several interesting cases of the structural induction. Firstly, we consider the base case where E is an enumerated value ev:

$$
\begin{array}{rll}
 & f(ev[V := C]) & \\
= & f(ev) & ev \text{ has no free variables} \\
= & ev & f(ev)=ev \\
= & ev[V := f(C)] & ev \text{ has no free variables}
\end{array}
$$

The case of an equality predicate makes use of the injectivity of f:

$$
\begin{array}{rll}
 & (E1 = E2)[V := f(C)] & \\
\Leftrightarrow & E1[V := f(C)] = E2[V := f(C)] & \text{substitution distributes} \\
\Leftrightarrow & f(E1[V := C]) = f(E2[V := C]) & \text{induction hypothesis} \\
\Leftrightarrow & E1[V := C] = E2[V := C] & f \text{ is injective} \\
\Leftrightarrow & (E1 = E2)[V := C] &
\end{array}
$$

The case of a membership predicate is similar to the case for equality:

S is a set. The set S in $max(S)$ must be a set of integers and therefore will never be permuted. The set S in $card(S)$ can be any finite set and therefore could be permuted. Such permutation is sound since the injectivity of f means that for any set S, $card(S) = card(f(S))$. This is currently not supported in our definition of f.

$$
\begin{array}{lll}
 & (E1 : E2)[V := f(C)] & \\
\Leftrightarrow & E1[V := f(C)] : E2[V := f(C)] & \textit{substitution distributes} \\
\Leftrightarrow & f(E1[V := C]) : f(E2[V := C]) & \textit{induction hypothesis} \\
\Leftrightarrow & E1[V := C] : E2[V := C] & \textit{f is injective} \\
\Leftrightarrow & (E1 : E2)[V := C] &
\end{array}
$$

For pairs of expressions, we have:

$$
\begin{array}{lll}
 & f((E1 \times E2)[V := C]) & \\
= & f(E1[V := C] \times E2[V := C]) & \textit{substitution distributes} \\
= & E1[V := C] \times E2[V := C] & \textit{induction hypothesis} \\
= & (E1 \times E1)[V := C] &
\end{array}
$$

For the case of set comprehension, we have:

$$
\begin{array}{lll}
 & \{x \mid x \in S \wedge P\}[V := C] & \\
= & \{x \mid x \in S[V := C] \wedge P[V := C]\} & \textit{substitution distributes} \\
= & \{x \mid x \in S[V := f(C)] \wedge P[V := f(C)]\} & \textit{induction hypothesis} \\
= & \{x \mid x \in S \wedge P\}[V := f(C)] & \textit{substitution distributes} \\
= & f(\{x \mid x \in S \wedge P\}[V := C]) & \textit{induction hypothesis}
\end{array}
$$

The case for a universal quantification predicate:

$$
\begin{array}{lll}
 & (\forall x \cdot (x \in S \Rightarrow P))[V := f(C)] & \\
\Leftrightarrow & \forall x \cdot (x \in S[V := f(C)] \Rightarrow P[V := f(C)]) & \textit{substitution distributes} \\
\Leftrightarrow & \forall x \cdot (x \in S[V := C] \Rightarrow P[V := C]) & \textit{induction hypothesis} \\
\Leftrightarrow & (\forall x \cdot (x \in S \Rightarrow P))[V := C] &
\end{array}
$$

The remaining cases (e.g., for a conjunction of two predicates, $P \wedge P$) follow similar patterns of proof to the above. Note that for the cases of set comprehension and universal quantification, x is a free variable and is independent of V and C in both.

From Theorem 4.2, it follows that for any B machine, the symmetries induced by f preserve the truth value of the system invariant (which is a predicate). Another interesting corollary is obtained by characterising B operations as predicates, as described in Leuschel and Butler [2005]. This involves representing machine constants and variables by a vector v, and defining a predicate $P(x, v, v', y)$ that holds when an operation $x \leftarrow Op(y)$, operates on input variables, y, and output variables x, to generate the new value of v, denoted v'. Applying the idea to states of a machine M, a state s relates to s' by operation $i \leftarrow Op(j)$ (write $s \rightarrow^{M}_{op.i.j} s'$), when predicate $P(i, s, s', j)$ holds. Then, by Theorem 4.2, we can conclude that $P(i, s, s', j)$ holds *iff* $P(f(i), f(s), f(s'), f(j))$ holds. That is, f preserves the transition relation of a B machine[3].

Corollary 4.3. *From Theorem 4.2 if follows that for every state permutation f, for B machine M, with invariant I, we have:*

[3]In [Clarke et al., 1999, Section 14.1], the symmetries induced by f are referred to as automorphisms of the system.

- $\forall s \in S : s \models I$ *iff* $f(s) \models I$
- $\forall s \in S, \forall s' \in S: \quad s \rightarrow^{M}_{op.i.j} s' \quad \Leftrightarrow \quad f(s) \rightarrow^{M}_{op.f(i).f(j)} f(s')$.

Using induction with Corollary 4.3, it also follows that a trace of execution between states, s and s' exists *iff* there exists a corresponding trace between $f(s)$ and $f(s')$. A symmetry reduction algorithm that identifies symmetries defined by f is then guaranteed to encounter all reachable orbits of states.

Having identified and justified the types of symmetries that are exploited by our symmetry reduction technique, we can now describe *how* this is achieved. In the next section we justify and present our method for representing B states. In later sections, we describe how we use these representations to identify symmetric states.

4.3 Representing a State as a Graph

Recall that values in B are either elements of sets (including Boolean values and integers), pairs of values, or sets of values. These rules can be used to construct any (arbitrarily nested) valid B value. Indeed, this is how ProB encodes values. A natural representation of the value of a B state using such rules is the graph, where each vertex corresponds to a single rule. Our research of formulating a method to exploit symmetries between states in B uses this natural representation of states. The challenge of detecting symmetric states therefore becomes one of identifying graph isomorphism. Our decision to use graphical forms for states is supported by the large amount of research into the graph isomorphism problem. Moreover, although the problem has no known polynomial time solution, in practice some extremely efficient algorithms exist for most classes of graphs [Kocay, 1996], which may contain several thousands of vertices [Foggia et al., 2001]. Given that our graphical representation for states uses one vertex per rule, this suggests that our approach to symmetry reduction has the potential to scale even for B specifications containing large amounts of state information. Later in this chapter we present the use of identifying graph isomorphism in two symmetry reduction techniques for ProB. However, as a first step we describe our procedure for constructing a graphical representation of any B state.

Let us consider an example of a graph that represents a state, which we will refer to later. Figure 4.6 shows the state graph of the state, $db = \{(Name1, Code1), (Name2, Code2)\}$ in Figure 4.2.

In this graph, the value of the relation, db is represented by edges that indicate specific ordered pairs, whose edge labels denote the variable they encode. A special 'root' vertex is also shown, as are vertex colours; these will be explained later in this section[4].

[4]The 'root' vertex in this case refers to the root of the state graph. This differs from the root node occurring in state spaces traversed by ProB, which represent the uninitialised B machine (introduced in Section 1.5.1).

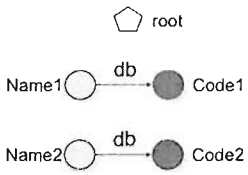


FIGURE 4.6: A phonebook state as a graph

We use the fact that B values are either elements of sets, pairs of values, or sets of values as follows. Each element of a set will correspond to a unique vertex in the final graphical representation. Extra vertices are also introduced to handle nested values. However, let us first consider the simple cases where we ignore nested values in B. Then, we can use four simple rules to translate a value to its concrete graphical representation. For an element of a set (an *atom)*, $v \in S$, where $v = s_0$, we have the graph in Figure 4.7. The graph of a set, $v \in \mathbb{P}(S)$, where $v = \{s_0, \ldots, s_n\}$ is shown in Figure 4.8. For a pair, $v \in S \times T$, where $v = (s_0, t_0)$, we construct a graph as in Figure 4.9. Finally, a relation, $v \in S \leftrightarrow T$, where $v = \{(s_0, t_0), \ldots, (s_n, t_m)\}$ is depicted in Figure 4.10. Also, although our graph representation does not distinguish $v = \{s_0\}$ from $v = s_o$, or $v = \{(s_0, t_0)\}$ from $v = (s_o, t_0)$, the B type system does and we only work with well-typed machines (typing is decidable in B).

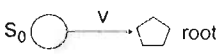


FIGURE 4.7: Graph for an atom

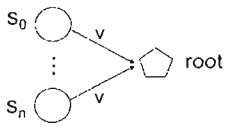


FIGURE 4.8: Graph for a set

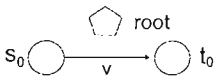


FIGURE 4.9: Graph for a pair

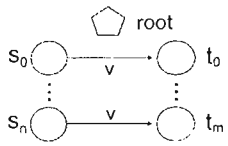


FIGURE 4.10: Graph for pairs

We extend this idea for nested data structures, such as sets of sets, through the introduction of a set of special vertices, X, which contains an element $x \in X$ for each nested value, *val*. Informally, for nested sets, $v = \{val_0, \ldots, val_n\}$, we create $n + 1$ special vertices, and we translate the set $\{x_0, \ldots, x_n\}$ to a graph, as in Figure 4.8. Then, we recurse on each nested value val_i, $0 \leq i \leq n$ and draw the corresponding graph with x_i as the new 'root'.

Similarly, for nested relations, $v = \{(val_0, val_1), \ldots, (val_{n-1}, val_n)\}$, we have $n+1$ special vertices, and we translate the relation $\{(x_0, x_1), \ldots, (x_{n-1}, x_n)\}$ to a graph, as in Figure 4.10. Then, we recurse on each nested value val_i, $0 \leq i \leq n$ and draw its corresponding graph with x_i as the new 'root'. This idea also applies to any variable whose value is a nested pair, in which case there will be just two vertices from X.

As an example, Figure 4.11 shows the graph for a variable, $v_1 \in \mathbb{P}(\mathbb{P}(S))$, whose value is $v_1 = \{\{s_0\}, \{s_1\}\}$. Note the two vertices, x_0, x_1 used to represent nested values.

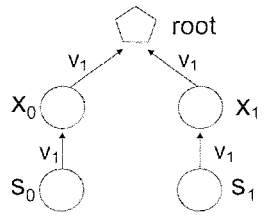


FIGURE 4.11: The graph for variable, $v_1 = \{\{s_0\}, \{s_1\}\}$

It is also convenient to encode typing information of a B state into such graphs, which can be indicated to a canonical labelling procedure such as nauty. The assignment of labels to vertices is the method we adopt. By convention, vertex labels are called *colours*, and a vertex labelled graph is called a *coloured* graph. So, more precisely, we choose to assign the same colour to vertices *iff* the symmetric permutation function f (Definition 4.1) permutes their corresponding state values. A colouring scheme is depicted in Figure 4.6, which shows a state of a phonebook. As can be seen, *Names* and *Codes* have different colours since they are two distinct deferred sets; whereas the individual elements of *Names* or *Codes* have the same colour.

The coloured graph representing a B state is constructed by composing the individual graphs representing the values of each state variable. This is called the *state graph* of a state. Formal descriptions of the recursive procedures for constructing state graphs are presented in Algorithms 6-11. The top level procedure is *state_graph* (Algorithm 6), which takes as parameter a state of a B machine. Firstly, it sets up the colours to be assigned to graph vertices, using global variables. The injection, *dcol*, provides different colours for each deferred set used by the machine (*dsets*). *x_col* is a colour, not already used in *dcol*, which is used for vertices of X required to represent nested values (see Algorithm 6, line 5). *esets* is the set of enumerated sets used by the machine, and *used_col* is the set of colours already in use by the state graph. Subsequently, this algorithm draws coloured graphs (Algorithm 6, line 9) representing each variable value in the state – each of which has the same 'root' vertex, for consistency within the final state graph.

The assignment of colours to vertices is performed by Algorithm 7, *assign_colour*. The use of *dcol*, on line 2 of this procedure, ensures vertices corresponding to values in

the same deferred set have the same colour. Similarly, *xcol* (line 8) ensures vertices corresponding to nested values have the same colour. On the other hand, for enumerated values (line 4), *vertex* is assigned any unused colour, c, which is then used to update the set of used colours, *used_col*. Therefore, a unique colour is assigned to every vertex that corresponds to an enumerated element in the B machine. Moreover, note that vertices are assigned the same colour *iff* their corresponding values are permuted by f.

In Algorithm 8, the *var_graph* routine establishes the type of B value (set, atom, relation or pair) of a variable, and applies the corresponding procedure. Subsequently, the *set*, *atom* and *relation* algorithms create the necessary vertices and draws the edges required. Note that we do not include a separate algorithm for drawing the graph of a pair, since this is the same as a relation of size 1. Instead, we convert the pair to a singleton relation, and apply the *relation* algorithm: see line 9 of Algorithm 8. In Algorithms 9-11, vertices are created for each value analysed. Those that correspond to nested values are elements of X, otherwise they correspond to a distinct element of a set in the original B state; see Algorithm 9, line 2, Algorithm 10, line 1, and Algorithm 11, line 2. Observe that, for each nested value, we apply it to *var_graph*, to construct the graph rooted at the current vertex. This is only possible in the *set* or *relation* routines (i.e., Algorithm 9, line 6, and Algorithm 11, lines 10 and 13), since atoms are indivisible. Also note that edges are not drawn to the root vertex, for relations (Algorithm 11, line 7). The precise effects of this choice, and other properties of the structure of state graphs, are not studied in this research; our focus is to implement a working concept. Given that the structure of state graphs could affect the time required for an algorithm to determine isomorphism, for such graphs, the issue may constitute future work. For inspiration, we can look to the *saucy* program [Darga et al., 2004], which optimises nauty for graphs that represent CNF formulae.

Algorithm 6 *state_graph*(*state*)

Require: State value, *state*

1: // Setup vertex colours using global variables
2: **global** $dsets := \{DS_0, \ldots, DS_n\}$; // deferred sets used in machine
3: **global** $esets := \{ES_0, \ldots, ES_m\}$; // enumerated sets used in machine
4: **global** $dcol :=$ an injection from $\{0, \ldots, n\}$ to a set of colours, *Colours*;
5: **global** $xcol :=$ any element from $Colours - ran(dcol)$;
6: **global** $used\_col := ran(dcol) \cup \{xcol\}$;
7: // Draw graph
8: **for all** variable-value pairs, $\langle v, var \rangle$ in *state* **do**
9: $\quad var\_graph(\text{'}root\text{'}, v, val)$;
10: **end for**

Let us now illustrate the use of *state_graph*, by applying it to the B state, $\langle v_1 = \{(\{s_0\}, \{s_1\})\}, v_2 = \{\{s_2\}\}\rangle$, where $s_0, s_1 \in DS_0$, $s_2 \in DS_1$, and $DS_0, DS_1 \subseteq dsets$. The action of computing $state\_graph(\langle v_1 = \{(\{s_0\}, \{s_1\})\}, v_2 = \{\{s_2\}\}\rangle)$ results in the graph depicted in Figure 4.12. Note that the same colour is given to vertices x_0, x_1 and x_2, which belong to the set of vertices, X.

Algorithm 7 *assign_colour(val, vertex)*

Require: B value, *val*, and corresponding vertex, *vertex*

1: **if** $\exists S_p \cdot S_p \in dsets \wedge val \in S_p$ **then**
2: assign *vertex* with colour, $dcol(p)$;
3: **else if** $\exists S_p \cdot S_p \in esets \wedge val \in S_p$ **then**
4: assign *vertex* with colour, $c \in Colours - used\_col$;
5: $used\_col := used\_col \cup \{c\}$;
6: **else**
7: // *val* is nested, so use colour for vertices in X
8: assign *vertex* with colour, *xcol*;
9: **end if**

Algorithm 8 *var_graph*(V_{parent}, v, *val*)

Require: Vertex V_{parent}, and variable v, with value *val*

1: **if** *val* is a set **then**
2: $set(V_{parent}, v, val)$;
3: **else if** *val* is an atom **then**
4: $atom(V_{parent}, v, val)$;
5: **else if** *val* is a relation **then**
6: $relation(V_{parent}, v, val)$;
7: **else**
8: // *val* is a pair
9: $val := \{val\}$; // new value has same graph
10: $relation(V_{parent}, v, val)$;
11: **end if**

Algorithm 9 *set*(V_{parent}, v, *val*)

Require: Vertex V_{parent}, and variable v, with value $val = \{val_0, \ldots, val_n\}$

1: **for all** $0 \leq i \leq n$ **do**
2: Create vertex V_{val_i}, such that if V_{val_i} is not an atom, then $V_{val_i} \in X$;
3: $assign\_colour(val_i, V_{val_i})$;
4: Draw edge, $V_{val_i} \overset{v}{\mapsto} V_{parent}$;
5: **if** val_i is not an atom **then**
6: $var\_graph(V_{val_i}, v, val_i)$;
7: **end if**
8: **end for**

Algorithm 10 *atom*(V_{parent}, v, *val*)

Require: Vertex V_{parent}, and variable v, with value $val = val_0$

1: Create vertex V_{val_0};
2: Draw edge, $V_{val_0} \overset{v}{\mapsto} V_{parent}$;

Algorithm 11 *relation*(V_{parent}, v, val)

Require: Vertex V_{parent}, variable v, with value $val = \{(val_0, val_1), \ldots, (val_{n-1}, val_n)\}$

1: **for all** $(val_i, val_j) \in val$ **do**
2: Create vertices V_{val_i}, V_{val_j}. If these are not atoms, then V_{val_i}, $V_{val_j} \in X$;
3: *assign_colour*(val_i, V_{val_i});
4: *assign_colour*(val_j, V_{val_j});
5: Draw edge, $V_{val_i} \overset{v}{\mapsto} V_{val_j}$;
6: **if** V_{parent} is not '*root*' **then**
7: Draw edges, $V_{val_i} \overset{v}{\mapsto} V_{parent}$, $V_{val_j} \overset{v}{\mapsto} V_{parent}$;
8: **end if**
9: **if** val_i is not an atom **then**
10: *var_graph*(V_{val_i}, v, val_i);
11: **end if**
12: **if** val_j is not an atom **then**
13: *var_graph*(V_{val_j}, v, val_j);
14: **end if**
15: **end for**

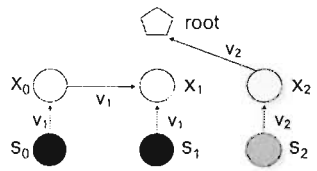


FIGURE 4.12: The graph computed by *state_graph*($\langle v_1 = \{(\{s_0\}, \{s_1\})\}, v_2 = \{\{s_2\}\}\rangle$)

4.4 Relating Graph Isomorphism to State Equivalence

In this section we aim to elucidate the relationship of symmetries between states, as described in Section 4.2, and the identification of isomorphic state graphs, using the concepts introduced in Chapter 2. (Note that by referring to isomorphic state graphs, we mean graphs that have the same 'shape', and vertex colouring.)

In Section 4.2, it has been shown that the application of permutation function, f, to a state will preserve both its structure in terms of the core syntax used to encode the state (see Figure 4.5), and the elements of enumerated/deferred sets its uses, e.g., given $x \in S$, then $f(x) \in S$. As we have proved, permutation states, s and $f(s)$, then satisfy the same predicates, and induce automorphisms on the transition relation of the B machine. Therefore, we say s and $f(s)$ are equivalent, or symmetric.

Our method for identifying symmetric B states is to use a canonical labelling algorithm that establishes whether their corresponding state graphs are isomorphic. To relate state symmetries to the algorithms used for identifying graph isomorphism, let us recall that canonical labelling functions rely on the permutation of graph vertices to find for some graph a unique canonical label that is the same for all isomorphic graphs. In the

terminology of the symmetries we have defined over states, the action of the permutation function, f, corresponds to the permutations/relabellings of graph vertices applied during the identification of a canonical label of a state graph, e.g., during partition refinement (refer to Section 2.3). More precisely, the relabelling must guarantee that permutations only occur over vertices of the same colour, since vertices have the same colour *iff* f permutes the corresponding state values (see Algorithm 7, *assign_colour*). We require this condition to be satisfied because canonical labels depend on the vertex permutations involved in their computation; if the condition is not satisfied, we cannot guarantee that isomorphic state graphs have the same canonical label. We now define our canonical labelling function, which can be used to identify isomorphic state graphs.

Definition 4.4. The function, *canon*, computes for a state graph its canonical label, by permuting vertices with the same colour, such that for two graphs, g_1 and g_2, $canon(g_1) = canon(g_2) \Leftrightarrow g_1$ is isomorphic to g_2.

To justify the use of *canon* to identify symmetric B states, we must also guarantee that isomorphic state graphs are only found when their corresponding states are symmetric. This is expressed by the following theorem:

Theorem 4.5. *For any two states, s and s', s is symmetric to s' iff $state\_graph(s)$ is isomorphic to $state\_graph(s')$.*

We do not present a full proof of Theorem 4.5, however, we describe several steps to show that it is intuitive. For these steps, we formally define *state_graph* as an injective function from B states to state graphs. This is reasonable given that the procedure works by analysing the structure of a state, so that different states generate different state graphs.

	State, s is symmetric to state, s'	
$\Leftrightarrow$	$\exists f \cdot f(s) = s'$	
$\Leftrightarrow$	$\exists f \cdot state\_graph(f(s)) = state\_graph(s')$	*state_graph is injective*
$\Leftrightarrow$	$\exists f \cdot f(state\_graph(s)) = state\_graph(s')$	*f commutes through state_graph*
$\Leftrightarrow$	$state\_graph(s)$ isomorphic to $state\_graph(s')$	

The key step is the commutability of f through *state_graph*. Although we have not formally defined f over state graphs, we highlight the validity of this claim by noting that f preserves the structure of a B state and permutes only elements of deferred sets. Then, given that *state_graph* analyses the structure of a state to find its state graph, it follows that the application of f to a state graph preserves the structure of the graph, and only affects the vertices that correspond to elements of deferred sets. In the final step, the test for isomorphism between state graphs can be carried out by the *canon* function, given in Definition 4.4.

Our goal is now to develop a canonicalisation function for state graphs, which can identify the presence of symmetry between the original B states. The premise is to base

the function on the underlying algorithm used by nauty, which is shown to be effective and efficient on large graphs that may contain several thousands of vertices [Foggia et al., 2001]. However, state graphs cannot be applied immediately to nauty, which works on graphs without edge labels[5]. In the next section we show how the underlying algorithm of nauty, presented in Section 2.3, can be extended to work directly with state graphs.

4.5 Computing Canonical Labels for Labelled, Directed Graphs

In order to represent the values of individual variables and constants, as well as to faithfully represent more complicated B data structures as graphs, we have to use directed, labelled and coloured graphs. Our algorithm to compute canonical labels for such graphs has been developed and implemented using SICStus Prolog, to allow integration into the ProB tool-set.

In contrast to adjacency matrices for undirected graphs, those for directed graphs are not necessarily triangular. Therefore, the move to directed graphs is straightforward and consists of using full adjacency matrices in the procedure for computing a canonical label, Algorithm 3.

Similarly, treating coloured graphs is also not difficult; we simply define an order on the vertex colours used, and then start off with an initial partition where the vertices have already been partitioned according to the various colours (see also McKay [1981]). For example, ordering colours dark to light, the initial partition for the graph in Figure 4.12, would be, $[\{s_0, s_1\}, \{s_2\}, \{x_0, x_1, x_2\}, \{root\}]$. As a consequence of this partitioning, the initial partition will be finer than the unit partition, providing the original B machine uses more than one type. This affects the size of the search tree explored when computing the canonical label, and therefore may affect the time required for computation, i.e., the finer an initial partition is, the less time required to compute its canonical label. However, also note that finer initial partitions will generally correspond to states with smaller orbits.

The move to graphs with labelled edges is less obvious. We now describe our adaptation of the partition refinement procedure to handle such graphs. The main algorithm for computing the canonical label of a coloured graph with directed and labelled edges is the same as Algorithm 3, except for the *compute_label* procedure, given in Definition 2.9, which determines a matrix for a given discrete partition. The change consists of placing an ordering on the set of labels, L (the variable names), so that labelled graphs can

[5]This was true at the time we developed our algorithm. In the most recent version of *nauty*, version 2.4 (Beta), released 11 July 2007, graphs with edge labels are handled. However, such graphs need transformation to a graph which nauty can accept, e.g., a graph containing n vertices and k edge labels is transformed to one containing $O(n \log k)$ vertices. Our approach works directly with state graphs.

be encoded as a single matrix, where each entry is a binary string of size $\mid L \mid$. For directed edges, we ensure *compute_label* takes the row-by-row binary string for the full matrix. As an example, consider Figure 4.12 and variable ordering v_1, v_2. Then, the matrix entry at index $[x_0, x_1]$ would be '1, 0', since between x_0 and x_1 there exists only the single edge labelled v_1. Regarding the implementation in Prolog, this matrix entry consists of a term of the form, `mentry(x_0,x_1)`, since this gives constant time access to the value of `x_0` or `x_1`: a task performed frequently when analysing the order of matrices[6]. The current distribution of SICStus Prolog limits the number of arguments of `mentry` terms to 256. Therefore, this approach only handles B specifications containing up to 256 variables[7]. To handle more than 256 variables, we also implement an alternate method that uses AVL-Trees [Knuth, 1973], which have logarithmic access times.

We now present the changes to the partition refinement procedure. Since we work on graphs with directed and labelled edges, we must first adapt the function $d(v, W)$ and the definition of equitable.

Definition 4.6. Let G be a graph with directed, labelled edges and set of vertices V, $v \in V$, $W \subseteq V$ and $L := (l_1, \cdots, l_l)$ the labels on the edges. Then $d_{in}(v, W, l_\nu)$ is the number of elements in W, that have an edge with the label $l_\nu \in L$ leading to v, and $d_{out}(v, W, l_\nu)$ is the number of elements in W, that have an edge with the label l_ν coming from v.

Definition 4.7. Let G be a graph with directed, labelled edges and set of vertices V and $L := (l_1, \cdots, l_l)$ the labels on the edges. An ordered partition π of V is called *label equitable* if, for all cells V_1, V_2 in π, $v_1, v_2 \in V_1$ and label $l_\nu \in L$, we have:

$$d_{in}(v_1, V_2, l_\nu) = d_{in}(v_2, V_2, l_\nu) \text{ and}$$
$$d_{out}(v_1, V_2, l_\nu) = d_{out}(v_2, V_2, l_\nu).$$

The original procedure for refinement of a partition (Algorithm 2) can now be adapted to handle directed, labelled graphs, and is given in Algorithm 12.

Example 4.1. *We integrate the methods described in Algorithm 3 and Sections 4.3-4.5, to show how to compute the canonical label of an example B state, whose state graph G_x is given in Figure 4.13.*

The example state makes use of two deferred sets, DS_1 and DS_2, and uses two variables, $v_1 \in \mathbb{P}(\mathbb{P}(D_1))$ and $v_2 \in \mathbb{P}(D_1 \leftrightarrow D_2)$. Let us assume $s_0, s_1 \in DS_1$, $s_2 \in DS_2$, and the variables values $v_1 = \{\{s_0\}\}$ and $v_2 = \{\{(s_1, s_2)\}\}$. Note, Figure 4.13 also depicts the special vertices $x_0, x_1 \in X$ for nested values, and the orders of variables and colours. Therefore, we start with the initial partition $\pi = [\{s_0, s_1\}, \{s_2\}, \{x_0, x_1\}, \{root\}]$, such

[6] An alternate structure with constant time access is the array. However, these are not available in SICStus Prolog.

[7] In practice, it is not often that we deal with B machines containing more than 256 variables. However, it may be required by large scale projects, e.g., industrial projects.

Algorithm 12 *refine*(π, G): Extended partition refinement

Require: Directed, labelled graph G, $\pi = [V_1, \cdots, V_n]$, $L = (l_1, \cdots, l_l)$

1: $\tilde{\pi} := \pi$;
2: $\alpha = [V_1, \cdots, V_n]$;
3: **while** $\tilde{\pi}$ is not discrete and α is not empty **do**
4: Remove an element W from α;
5: **for all** $\nu \in 1 \ldots l$ **do**
6: **for all** $k \in 1 \ldots n$ **do**
7: Compute ordered partition $[Y_1, \cdots, Y_s]$ from V_k, where $\forall i, j, x, y \cdot 1 \leq i, j \leq s \wedge x \in Y_i \wedge y \in Y_j \Rightarrow i < j \Leftrightarrow d_{in}(x, W, l_\nu) < d_{in}(y, W, l_\nu)$
8: **if** $s > 1$ **then**
9: update $\tilde{\pi}$ by replacing the cell V_k with the cells $Y_1, \cdots, Y_s$;
10: $\alpha = concatenate(\alpha, [Y_1, Y_2, \cdots, Y_s])$;
11: **end if**
12: **end for**
13: **end for**
14: Repeat one time lines 5 - 13, but use alternate condition $d_{out}(x, W, l_\nu) < d_{out}(y, W, l_\nu)$ on line 7.
15: **end while**
16: **return** Label equitable partition, $\tilde{\pi}$;

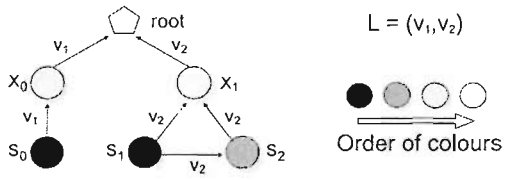


FIGURE 4.13: Example graph

that vertices in the same cell have the same colour, and the cells are ordered by their colours.

Initially, line 1 of Algorithm 3 requests the partition refinement of π. *We then enter Algorithm 12 with* π, *and* $\alpha = [\{s_0, s_1\}, \{s_2\}, \{x_0, x_1\}, \{root\}]$. *In the first traversal of the* **while** *loop,* $W = \{s_0, s_1\}$. *The algorithm considers only edges with label* v_1 *in the first cycle of the outer* **for***-loop. For the first cell of* π, $V_1 = \{s_0, s_1\}$, *no edges labelled* v_1 *originate from* W *and lead to an element of* V_1; *so* $d_{in}(s_0, W, v_1) = d_{in}(s_1, W, v_1) = 0$ *and* π *remains unchanged. The second cell of* π, $V_2 = \{s_2\}$, *is trivial and cannot be split further, so the algorithm continues. For the third cell,* $V_3 = \{x_0, x_1\}$, *we have* $d_{in}(x_0, W, v_1) = 1 > 0 = d_{in}(x_1, W, v_1)$. *So,* V_3 *is split into two cells,* $\{x_1\}, \{x_0\}$; *which must be ordered by their values for* d_{in}. *The algorithm updates* π *and* α, *where* $\pi := [\{s_0, s_1\}, \{s_2\}, \{x_1\}, \{x_0\}, \{root\}]$ *and* $\alpha := [\{s_2\}, \{x_1, x_0\}, \{root\}, \{x_1\}, \{x_0\}]$. *Since the last cell* $V_4 = \{root\}$ *is trivial, the algorithm progresses and begins considering edges labelled,* v_2 *(line 5, second iteration).*

Splitting next occurs when d_{out} *is analysed (execution of line 14), for* $W = \{s_2\}$ *and*

the edge label, v_2. When, $V_1 = \{s_0, s_1\}$, we have, $d_{out}(s_0, W, v_2) = 0 < 1 = d_{out}(s_1, W, v_2)$ and so partition π is updated to $\pi := [\{s_0\}, \{s_1\}, \{s_2\}, \{x_1\}, \{x_0\}, \{root\}]$, which is now discrete. Further splitting is not possible, so Algorithm 12 terminates with this discrete, label equitable partition[8]. With execution returning to line 1 of Algorithm 3, π_e is discrete and the procedure terminates with the canonical label, compute_label(G_x, π_e). Figure 4.14 shows the first two rows of the adjacency matrix of G, which constitute the twelve most significant bits of the canonical label.

	s_0	s_1	s_2	x_1	x_0	*root*
s_0	00	00	00	00	10	00
s_1	00	00	01	01	00	00
⋮		⋮		⋮		⋮

FIGURE 4.14: Adjacency matrix corresponding to the canonical label of G_x

Our implementation also integrates the optimisations of automorphism pruning and lexicographic pruning into the search (see Section 2.4), with an option to turn them on/off. However, it should be noted that our canonicalisation procedure *does not* use several intricate programming optimisations used in nauty. For example, nauty represents branches of the search tree using two array variables. This improves upon the memory efficiency of our procedure, which constructs branches via recursion (Prolog has no loop structures), and so a branch comprises a number of stackframes in memory.

4.6 Symmetry Reduced Model Checking Algorithm

In this section, we formalise the integration of the canonicalisation function, developed in Sections 4.2-4.5, into model checking in PROB. The resulting symmetry reduction scheme for B machines is presented in Algorithm 13.

The algorithm behaves much like the standard procedure for model checking in PROB, presented in Algorithm 1[9]. Although now, for each state encountered, we compute its canonical label, so that we can determine whether a state symmetric to it has already been encountered. This computation takes place in a two-phase process that first computes the state graph of the state, and then applies the canonicalisation function to the resulting graph: see line 8.

[8]Note that a label equitable partition may not be discrete; see Definition 4.7.

[9]For the clarity of the presentation, we omit from Algorithm 13 the *SGraph* variable present in Algorithm 1, which is used to represent the reached state space – for visualisation purposes. However, such visualisations are still possible in the implementation of Algorithm 13.

Algorithm 13 Symmetry Reduced Model Checking in ProB

Require: An abstract machine M and invariant ϕ

```
1:  Queue := ⟨root⟩ ; Visited := {canon(state_graph(root))};
2:  while Queue ≠ ⟨⟩ do
3:    state := pop(Queue);
4:    if state ⊭ φ or state deadlocks then
5:      return counterexample trace from root to state
6:    else
7:      for all successor succ, and operation Op such that state →^M_Op succ do
8:        sr := canon(state_graph(succ))
9:        if sr ∉ Visited then
10:         if random(1) < α then
11:           add succ to front of Queue
12:         else
13:           add succ to end of Queue
14:         end if
15:         Visited := Visited ∪ {sr}
16:       end if
17:     end for
18:   end if
19: end while
20: return ok
```

In a change from Algorithm 1, the $Visited$ variable now records the *canonical labels* of the states already reached by checking, as opposed to storing the actual states. Despite this difference, the approach still provides the same use: to identify whether a state (or symmetric state) has already been visited. For example, let us consider the case where the reduced strategy has checked a state s_0, so that $canon(state\_graph(s_0)) \in Visited$. If model checking then encounters a new state s_1, where $s_1 = f(s_0)$ (i.e., s_0 and s_1 are symmetric), we have $canon(state\_graph(s_0)) = canon(state\_graph(s_1))$. As a consequence, the condition on line 9 fails, since $canon(state\_graph(s_1)) \in Visited$, and the symmetric state s_1 is not added to the $Queue$ variable for further analysis.

Let us highlight that our method for symmetry reduction, given in Algorithm 13, takes the *first* state encountered in each orbit as the representative state. The canonical label is not the same as the representative state – it is a binary string (an example of which is given in Figure 4.14), which is computed for a state, and which is the same for all states in an orbit. Therefore, by storing the canonical label in *Visited*, it is possible to determine whether future states encountered are symmetric to it by computing their canonical labels, and testing for membership in *Visited* – as described above.

Empirical results from experimentation of this symmetry reduction technique are provided later in Chapter 5. The line of research presented in this chapter has also stimulated the development of two new techniques that achieve symmetry reductions in ProB – each of which exploit symmetries induced by the elements of deferred sets in B machines. The culmination of this research is given as related work in Sections 2.8.5

and 2.8.6; namely the methods of *permutation flooding* and *symmetry markers*. The next section details collaborative work that has developed another symmetry reduction technique for PROB, which aims to improve upon certain aspects of the classical approach given in Algorithm 13. This new method makes use of both the canonicalisation function, *canon*, for state graphs of B states, in addition to symmetry markers.

4.7 Collaborative Work: Canonical Labels + Symmetry Markers

A natural extension to the techniques presented so far is to integrate *efficient* approximate verification by symmetry markers with full (bounded) verification performed by canonicalisation labels; since, such a strategy could gain the benefits of both methods. This is the premise for the technique described in this section.

Informally speaking, the new procedure aims to verify B machines using symmetry markers alone. If two states are found to have different symmetry markers, there is no need to compute their canonical labels. However, if two states have the same symmetry marker, one determines the presence of any symmetry by computing a canonical label for each and comparing labels in the usual way. Thus, unlike the symmetry marker approach, this technique achieves verification. We can draw similarities between our approach with that used during model checking in SMC [Sistla et al., 2000], which also uses a procedure with two-phases to determine the presence of symmetry. Each state encountered in SMC has an associated integer, called a *checksum*, which is computed from the values of variables in that state. Checksums are computed such that two different checksums for two states guarantee the states are not symmetric, whereas identical checksums *do not* guarantee symmetry between the states. Hence, checksums achieve the same effect as our symmetry markers. If model checking encounters two states with identical checksums, SMC attempts to establish the presence of symmetry using a randomised algorithm that analyses whether a permutation between the states is a symmetry of the system. In contrast to our technique, this approach may falsely indicate non-symmetry, and therefore produce a non-minimal quotient model: however, the approach is still safe. The precise algorithm we use to combine symmetry markers with canonical labels in PROB is given in Algorithm 14.

Notice that the procedure is very similar to that of Algorithm 13, for model checking via canonical labels; the differences involve the initial computation of a symmetry marker (line 9), and the routine for subsequently computing a canonical label to determine symmetry (lines 18-32) when a symmetry marker is encountered more than once.

The algorithm uses the variables, *VisitedM* and *VisitedC*, to store the visited symmetry markers and visited canonical labels, respectively. The *Queue* variable is used as it is

Algorithm 14 Canonical Labels and Symmetry-Markers in ProB

Require: An abstract machine M and invariant ϕ

1: $VisitedM := \{m(root)\}$; $VisitedC := \{canon(state\_graph(root))\}$;
2: $Queue := \langle root \rangle$; $one\_mk\_reached := \varnothing$;
3: **while** $Queue \neq \langle\rangle$ **do**
4: $state := pop(Queue)$;
5: **if** $state \not\models \phi$ or $state$ deadlocks **then**
6: **return** counterexample trace from $root$ to $state$
7: **else**
8: **for all** successor $succ$, and operation Op **such that** $state \rightarrow^{M}_{Op} succ$ **do**
9: $succ_m := m(succ)$;
10: **if** $succ_m \notin VisitedM$ **then**
11: **if** random(1) $< \alpha$ **then**
12: add $succ$ to front of $Queue$;
13: **else**
14: add $succ$ to end of $Queue$;
15: **end if**
16: $VisitedM := VisitedM \cup \{succ_m\}$;
17: $one\_mk\_reached := one\_mk\_reached \cup \{succ_m \mapsto succ\}$;
18: **else**
19: **if** $\exists s \cdot succ_m \mapsto s \in one\_mk\_reached$ **then**
20: $s_c := canon(state\_graph(s))$; $VisitedC := VisitedC \cup \{s_c\}$;
21: $one\_mk\_reached := one\_mk\_reached \setminus succ_m \mapsto s$;
22: **end if**
23: $succ_c := canon(state\_graph(succ))$;
24: **if** $succ_c \notin VisitedC$ **then**
25: **if** random(1) $< \alpha$ **then**
26: add $succ$ to front of $Queue$;
27: **else**
28: add $succ$ to end of $Queue$;
29: **end if**
30: $VisitedC := VisitedC \cup \{succ_c\}$;
31: **end if**
32: **end if**
33: **end for**
34: **end if**
35: **end while**
36: **return** ok

in Algorithms 1 and 13, to indicate the next state to be checked. *one_mk_reached*, is a partial injection from symmetry markers to corresponding states, and is introduced to indicate that a particular symmetry marker has only been computed once[10].

The technique ensures only one state per symmetry class is checked, i.e., the first state encountered in its orbit. Let us illustrate this by examining a simple flow of execution for the process. Assume the algorithm first encounters state $s1$, which is added to *Queue*

[10]In practice, the function served by *one_mk_reached* is modelled by querying whether there exists a single instance of a particular symmetry marker in the Prolog database.

(so it will be checked), and the symmetry marker is computed, $s1_m := m(s1)$ (line 9), and is added to *VisitedM* (line 16). At this point, $s1_m$ is asserted as the only marker with its value by adding $s1_m \mapsto s1$ to *one_mk_reached* (line 17). Then, at a later point in checking, the algorithm encounters state $s2$, which is the first state found where $m(s1) = m(s2)$; hence, the condition on line 10 fails, and execution continues from line 18. Now, we use canonicalisation labels to establish whether a state symmetric to $s2$ has previously been checked. The only possible symmetric state is $s1$, since both have the same markers (recall that states with different symmetry markers are never symmetric, see Section 2.8.6). Therefore, we must compute the canonical label for $s1$. This is guaranteed because the expression on line 19 can be satisfied, which initialises s to $s1$. The algorithm then computes the canonical label of s and adds it to *VisitedC* (line 20). Implicitly, this means s is taken as the unique representative of its orbit. On the next line, we remove the maplet from *one_mk_reached*, to ensure the condition on line 19 can no longer be satisfied, therefore preventing superfluous computations of canonical labels in the future. Testing whether a state symmetric to $s2$ has been previously checked then involves computing the canonical label of $s2$, denoted $s2_c$, and testing for its membership in *VisitedC* (lines 23 and 24). This evaluates to true if $s1$ is symmetric to $s2$, so $s2$ need not be checked, and is not added to *Queue*. Otherwise, we ensure $s2$ will be checked in the future, by adding it to *Queue*; and we also add $s2_c$ to *VisitedC* so that we can detect future states encountered that are symmetric to it.

The approach presented in this section aims to gain the advantages of both symmetry markers and canonical labels. That is, a symmetry reduced model checking procedure that achieves verification, but which does not require the (possibly expensive) computation of canonical labels for each state encountered. In Chapter 5, experimental results are presented to illustrate the effectiveness of the technique. Furthermore, the results are compared with corresponding results of complementary techniques for symmetry reduction in PROB, and we discuss any drawbacks and improvements of the strategy.

4.8 Integrating Symmetry Reduction into the Architecture of PROB

This section presents an overview of the architecture of PROB, to show how the new techniques for symmetry reduced model checking described in this chapter integrate into the tool. The architecture is presented in Figure 4.15.

PROB has been developed primarily using SICStus Prolog, with a graphical user interface written in Tcl/Tk [Tcl/Tk]. The front end of the tool is responsible for transforming a B machine into a corresponding Prolog term representation, which is made available to the main components of the model checking system. A Prolog encoding of the B machine is given as input to the ProB interpreter, which makes calls to the 'ProB Kernel'

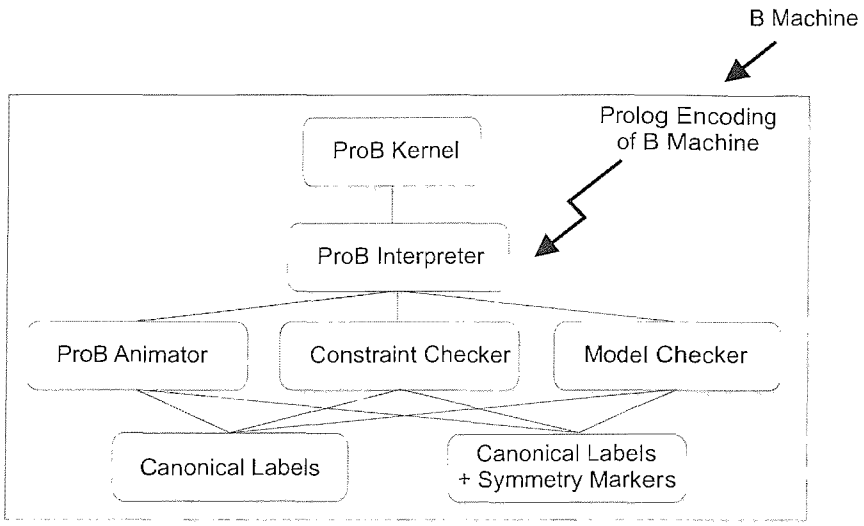


FIGURE 4.15: Integration of the Symmetry Reduction methods into the existing architecture of PROB.

so that the B system is stored. Separate components in the tool-set exist for the animation of B machines ('ProB Animator' in Figure 4.15), constraint-based model checking ('Constraint Checker') and standard model checking ('Model Checker'). The techniques presented for symmetry reduced model checking in this chapter, namely Canonical Labels, and Canonical Labels + Symmetry Markers, are encoded in Prolog within their own components so that they are easily 'plugged' in to other parts of the tool, as necessary. As can be seen, the symmetry reduction components are available to the animation and model checking components of PROB. In the current implementation of the tool, an options menu enables a user to select whether to use symmetry reduction, and if so, which method. For further information about the original components of PROB, refer to [Leuschel and Butler, 2003].

4.9 Summary

In this chapter, we have presented a framework for an automatic method that performs classical symmetry reduction in model checkers of B systems. All of the techniques developed have been integrated into PROB using SICStus Prolog. Symmetries are induced by abstract elements of *deferred sets*, and manifest in data structures used to represent B machine state; as described in Section 4.1.1. Identification of symmetries takes place by translating system states to directed, labelled and coloured graphs, called state graphs (Section 4.3), and subsequently computing *canonical labels* using an extension to the underlying algorithm of the graph isomorphism program *nauty* (Section 4.5): states are symmetric *iff* they have the same canonical label.

The line of research that has developed symmetry reduction techniques based on *scalarset* data types [Ip and Dill, 1993] is the inspiration for the work in this chapter. More recent work includes the SymmSpin package [Bosnacki et al., 2002], which uses scalarsets to integrate symmetry reduction into the SPIN model checker. These special types contain only symmetric data values and are thus similar to deferred sets in B. However, the scalarsets approach is not automatic; it requires the user to identify and assert the presence of symmetries. Our approach is fully automatic since the deferred set is a key component of the B language. Thus, the automation of model checking is not compromised. Furthermore, our reduction strategy will always apply to a large proportion of B machines since deferred sets are frequently used, e.g., in abstract machines near the top of refinement chains.

To our knowledge, this is the first technique to integrate symmetry reduction into model checkers of B. In addition, this research has stimulated the development of three separate approaches to symmetry reduced model checking in B; all of which exploit symmetries induced by deferred sets. An alternate approach to addressing the orbit problem is implemented in a strategy known as *permutation flooding* [Leuschel, Butler, Spermann, and Turner, 2007], and is described in Section 2.8.5. In this method when a new state is encountered, all symmetric states are added to the state space and are marked as 'already processed'. Following this, separate work was developed by Leuschel and Massart, which shows how graph invariants can be used as part of an efficient approximate-verification technique, called *symmetry markers* – see Section 2.8.6. In collaborative work, symmetry reduction by canonical labels is integrated with symmetry markers (Section 4.7, *canonical labels + symmetry markers*) to generate a new technique that aims to gain the advantages of both: a model checking algorithm with increased efficiency, compared to the canonicalisation approach, yet which performs full verification (within the bounds of the system). Furthermore, it is feasible that the techniques presented could be generalised to exploit symmetries in model checkers of other languages, providing they use data structures that induce symmetries, as with the deferred set.

In Chapter 5 we present an empirical analysis and discussion that compares the performance of each of the four techniques for symmetry reduction in PROB: *canonical labels*, *permutation flooding*, *symmetry markers*, and *canonical labels + symmetry markers*.

Chapter 5

Empirical Evaluation

This chapter presents an evaluation of the four techniques for symmetry reduction in ProB; *permutation flooding* (Section 2.8.5), *symmetry markers* (Section 2.8.6), *canonical labels* (Section 4.6) and *canonical labels + symmetry markers* (Section 4.7). Evaluation comprises experimentation and discussions for each method when applied to a range of B machines. Results from standard model checking are used as a control. The empirical data obtained provides evidence of the effectiveness of symmetry reduction, in addition to insights into aspects affecting performance that may need to be explored in the future.

All experiments are performed on a PC with a 2.8GHz Intel Pentium 4 processor, 1Gb of available main memory, running SICStus Prolog 3.12.0 (x86-win32-nt-4) on Windows XP Professional (Service Pack 2), and using ProB version 1.2.0.

The chapter is organised as follows. In Section 5.1, details are given on the performance of the Prolog implementation of the canonical labelling algorithm. Then, Section 5.2 presents an executive summary of the performances of each technique for symmetry reduction in ProB. We present the B machines used during experimentation in Section 5.3. Empirical results for model checking in the presence, and absence of errors, are split over the next two sections, respectively. Section 5.4.1 compares the performance of symmetry reduction via canonical labels with that of standard checking, and Section 5.4.2 compares results for the canonical label approach with the those of permutation flooding. Next, Section 5.4.3 compares the technique of canonical labels + symmetry markers, with that of canonical labels. Section 5.5 then shows how these techniques compare with symmetry markers, and standard model checking, when applied to B machines containing errors. Finally, a summary of the chapter is given in Section 5.6.

5.1 Performance Issues with Prolog Data Structures

To illustrate certain details of the performance of our implementation that computes canonical labels (Chapter 4), we apply it to a simple B machine that populates a single binary relation, given in Figure 5.1. For each test, we apply exhaustive standard model checking, and exhaustive symmetry reduced model checking via canonical labels, while varying the size of deferred sets D and R.

MACHINE
BinaryRelations
SETS
D;
R
VARIABLES
rel
INVARIANT
$rel \in D \leftrightarrow R$
INITIALISATION
$rel := \varnothing$
OPERATIONS
add(d , r) $\widehat{=}$
PRE $d \in D \wedge r \in R \wedge d \mapsto r \notin rel$
THEN
$rel := rel \cup \{d \mapsto r\}$
END;
END

FIGURE 5.1: Binary Relations Machine

The results of the test are displayed in Table 5.1, showing the number of different relation instances computed when symmetry reduction is either on or off, and whether automorphism pruning is being used (denoted 'aut on' or 'aut off'); *rel* has k elements in its domain and k elements in its range, time is recorded in seconds, and OT indicates the test took more than two hours to complete.

Only non-symmetric relations are computed when the reduction technique is used. Conversely, all $2^{k \times k}$ instances are computed when it is off. For example, when the size of the deferred sets, D and R, is 4, the symmetry reduced approach identifies 317 instances of the *rel* relation, whereas standard checking identifies 65537 instances. As can be seen, for $1 \leq k \leq 4$, the savings in time made when searching a reduced state space can greatly outweigh the time overhead of computing canonical labels for its constituent states. In

	Symmetry off		Symmetry via Canonical Labels		
k	States	Time	States	Time (aut off)	Time (aut on)
0	1	0.05	1	0.05	0.05
1	3	0.05	2	0.05	0.05
2	17	0.80	7	0.63	0.61
3	513	8.90	36	1.76	2.16
4	65,537	5070.12	317	133.44	172.27
5	33,554,433	OT	5624	OT	OT

TABLE 5.1: Comparison of the number of different relations

the case where, k is 4, the time required by the standard approach is almost 40 times that of the symmetry reduced approach. Thus, we have demonstrated that our method accomplishes the fundamental premise of symmetry reduction for this example. In addition, note that when k is 5, the reduced and standard strategies for model checking take longer than 2 hours to complete. This highlights that it may still be intractable for either approach to model check systems with large bounds on its types.

In Section 4.5, we describe that in our canonical labelling function we implement the optimisation of automorphism pruning – and provide an option to turn this on or off. The technique of automorphism pruning requires an efficient storage of the automorphism group of a graph, since such groups can be very large for large graphs. Our implementation makes use of the Schreier-Sims Algorithm [Kreher, 1998, Section 6.2.3, pages 203-211], which relies heavily on the manipulation of arrays. This is not so much of a problem for programs written in languages such as C, such as nauty, since constant time access to arrays is available. However, for the integration into ProB, this research uses Prolog; which has no built-in array data type. Therefore, our implementation models arrays as AVL-Trees, which have logarithmic access times, and for which SICStus Prolog has a specific module[1]. The effects of this can be seen in Table 5.1. Automorphism pruning produces a worse average execution time than when pruning turned off, e.g., when each deferred set contains 4 elements, there is a difference of approximately 40s. So, the overhead of using automorphism pruning outweighs the benefit of traversing smaller, constrained search trees, during the computation of canonical labels. Despite this, considerable savings are made in both computation time and size of the state space when compared to results from standard exhaustive model checking. Therefore, automorphism pruning is not used by default in the current implementation of symmetry reduction via canonical labels in ProB. Furthermore, although empirical data collected from experimentation in this chapter is encouraging, this research can still be viewed as proof of concept. Also, note that our implementation *does* use lexicographic pruning when computing canonical labels, since this is computationally inexpensive; the strategy only requires the construction of partial adjacency matrices, and a conditional statement that tests the order of two matrices (refer to Algorithm 3 and its optimisations).

[1]However, such AVL-Trees are still encoded as relatively high-level Prolog terms.

5.2 Executive Summary

This section provides an executive summary of the set of empirical results obtained from the experimentation of techniques for symmetry reduction in B, described in Chapter 4: symmetry reduction via *canonical labels* (Sections 4.1–4.5), *permutation flooding* (Section 2.8.5), efficient approximate verification by *symmetry markers* (Section 2.8.6), and *canonical labels + symmetry markers* (Section 4.7). Experimentation involved applying each method to a range of B machines (Section 5.3) that use many typical expressions and values in B, to ensure the results obtained are representative of a larger sample of machines. The precise details of experimentation can be found in Sections 5.4 and 5.5.

Let us first recall that the goal of model checking a B specification is to identify the presence of system errors, or the absence of errors; the knowledge of either is very useful. When we discover an error in ProB, we can analyse the value of the state that violates the invariant (or which is deadlocked), and view the sequence of operations that lead to it. Referring back to the specification, we can then attempt to resolve the causes of the error. This method is particularly useful during the development of a B specification. Alternately, if we discover the absence of errors in a B specification, we have an increased confidence in our system, and a guarantee that within its current bounds, the invariant holds and there are no deadlocks. If this precision does not suffice, one might then increase these bounds, or seek to apply theorem proving. To ensure our experimentation considers systems containing errors, or no errors, we apply each of the techniques named above to 8 B machines without errors, and 4 B machines with errors present. We justify this disparity by noting that model checking in ProB uses a combination of depth/bread first search, which changes on-the-fly, and is determined by a random factor, see Algorithm 1. This does not affect the amount of time required to exhaustively check state spaces containing no error. Neither does it affect the number of states/transitions inside these state spaces. Hence, we can directly compare results for the different techniques, with respect to a particular machine. Given that these machines exhibit symmetry in a variety of data structures, an arbitrary B machine is likely to use such constructs and should observe similar data trends if applied to the reduction techniques. Thus, we examine 8 machines, as opposed to 4, to increase our confidence in the performance of a particular technique, for a particular B machine. Such comparisons are not possible for machines containing errors, since we cannot guarantee the trace to the error found by ProB[2]. Therefore, we aim demonstrate the results that are possible for machines containing errors, and discuss the factors that influence performance.

[2]We could change the random factor to a constant in ProB's model checking algorithm, to force only a depth first, or breadth first search. However, for best performance one should know which to use. This is a non-trivial task and requires prior knowledge of a system error, and its location in the state space. Therefore, our experimentation uses the default random factor in ProB, i.e., for each new successor state, there is probability 1/3 it will be added to the front of the queue (for depth-first search); and 2/3 chance it is added to the back of the queue (breadth-first).

We first summarise the results from experimentation conducted over machines containing errors. Each technique is found to perform substantially better than standard model checking, and in some cases, speedups can exceed two orders of magnitude. In terms of the reduction in the number of states and transitions, the quotient models are found to be no less than 20 times smaller than their original models, and in one case is approximately 3500 times smaller. The significance of this reduction is made more prominent given that ProB encodes states explicitly - a state consisting of a single variable consumes less memory than a state containing two or more variables; in addition, a variable $v \in \mathbb{P}(DS)$ whose value has a cardinality of 1 consumes less memory than the same variable whose value has a cardinality of two or more. Therefore, reductions in the size of the state space, and so reductions in memory consumption, are more prominent for machines with complex state. To provide a better idea of the amount of memory that can be saved, we note that a Prolog fact, `state(var(v1,1))` consumes 80 bytes[3]. Supposing the state space has been reduced by a factor of 3500, and contains 350000 states in the unreduced model, then one would save 28000000 − 8000 = 27992000 bytes, or approximately 28 megabytes. In practice, facts representing states in ProB are more complicated than this example. A state with n variables will consist of a Prolog fact of the form `state(var(V1,VAL),...,var(Vn,VAL))`, where each argument `var(VAR,VAL)` encodes the value of a state variable, and each value, `VAL`, can be a nested prolog term. In addition, transitions between states of the form, `operation(state1ID,op(VAL1,...,VALN),state2ID)`, must also be stored, where `state1ID` and `state2ID` are effectively pointers to the states, and `op(VAL1,...,VALN)` is the operation between them, where each `VAL` can be a nested term representing a B value. Thus, one can see that the symmetry reductions made using the Canonical Labels technique can reduce memory consumption by a significant factor.

One factor influencing the magnitude of the speedups is the complexity of expressions used in the invariant or properties clauses of a B machine. Computationally expensive expressions include universal quantifiers, such as $\forall(x).(x \in S \Rightarrow f(x) \leq MAX)$, and closure operations, since their evaluation usually consists of many individual evaluations. Given that symmetry reduced checking only analyses a unique representative per orbit, it does not involve redundant analyses of expensive expressions. The result is a large amount of time saved during verification.

Common to all methods, the time to compute the representative of a B state increases with state complexity (e.g., which increases the size of a state graph). As a consequence, verification times tend to increase with the number of elements of deferred sets used by the state.

In addition to the general reason for the effectiveness of symmetry reduction, there are particular aspects of each method that affect performance. Symmetry reduction

[3]This is true for our test environment, given in the introduction to this chapter.

via canonical labels performs notably well on machines containing relations (including total functions, partial functions, bijections etc.), when compared to standard model checking. More specifically, the overhead of computing canonical labels is outweighed by the time saved in checking a smaller state space; whereas, standard model checking generates all symmetric instances of relations and requires considerably more time to exhaust the state space. Experimentation shows that speedups exceeding 230 times that of standard checking are possible. In contrast, results also reveal the method does not perform as well for machines containing multiple variables whose values are sets of deferred elements. The precise reason for this is difficult to determine. However, we can identify that a significant factor is the structure of corresponding state graphs. Consider a state graph representing a single deferred set, as described in Section 4.3. Then, we know that each of its vertices has the same number of in-coming and out-going edges. In addition, they have the same neighbouring vertices. As a consequence, partition refinement (Algorithm 12) cannot easily distinguish non-symmetric vertices when computing canonical labels, and so more vertex orderings need to be analysed to identify the canonical label. Therefore, computing representatives for may become excessively expensive.

Permutation flooding is a useful alternate strategy for symmetry reduction, which performs better than the canonical label technique for machines containing multiple variables whose values are sets. A factor contributing towards this result is the simplicity of permuting states (to identify symmetries) in a programming environment such as Prolog; which, moreover, has backtracking capabilities. Empirical data indicates that for such machines the method reaches speedups of up to 64 times that of standard checking, and 30 times faster than the canonical labels approach. Note however, that in the prominent cases where permutation flooding outperforms canonical labels, the times involved are still small, e.g., < 1 minute. Another aspect that makes this method effective is that the complexity of computing symmetric permutations depends only on the number of deferred set elements occurring inside the state. Therefore, the method is still valuable when the data values inside individual states become complicated.

On the other hand, permutation flooding typically does not perform as well for machines involving relations – which can have many more symmetries than a set. For example, a relation over two deferred sets, with a maximum of k elements in both the domain and range, has $2^{k \times k}$ different instances (see Table 5.1); whereas a set containing up to k deferred elements has 2^k instances. Given that permutation flooding generates all symmetric states, it follows that this method should take longer for machines containing symmetric relations than machines with symmetric sets. Indeed, results confirm this observation. In addition, the results show that permutation flooding can be outperformed by the canonical label approach by a factor of approximately 15, for machines whose symmetric data structures involve mainly relational types.

A consequence of the explicit storage of symmetric states in permutation flooding is that the method is not as memory efficient as the canonical label technique, which stores only one representative per orbit. One set of results illustrates that permutation flooding may even run out of memory, while the canonical labels technique successfully exhausts the state space.

Recall that symmetry reduction by canonical labels + symmetry markers aims to gain the benefits of canonical labels and symmetry markers by computing canonical labels for states *only* when identical symmetry markers are found (and therefore non-symmetry cannot be established). In practice, the method performs very well and experimentation indicates possible speedups exceeding 450 times faster than standard checking. In comparison to permutation flooding, the drawbacks include those experienced by the canonical labels approach, thus lower speedups are witnessed for machines containing variables that are sets of deferred elements. Similarly, the method retains the advantages of computing canonical labels, and so is especially suitable for machines containing variables whose values involve relations with symmetries. Furthermore, results highlight that computing symmetry markers is indeed advantageous, since typically verification time is less than that of the canonical labelling technique alone; in certain cases verification using this combined technique is an order or magnitude faster than canonical labelling alone.

We now discuss the performance of the symmetry reduction techniques, in addition to that of symmetry marking, when applied to machines containing errors[4]. The results obtained confirm that a major factor influencing counterexample/deadlock detection is the random element used during state space exploration in model checking in ProB. Therefore, we observe that the trace to an error, and the time required for its discovery, does not depend solely on the number of deferred elements used, or the model checking strategy. Additional factors that can affect error detection include the number of errors present and their location in the state space – although, these assume prior knowledge of the errors.

The efficiency of computing new states/representatives is another significant factor that affects detecting errors. Ideally, this should be as fast as possible, to minimise the time required to reach the error. Regarding the experimental results, we see that model checking using symmetry markers identify the machine errors faster than the any of other techniques. This method benefits from performing efficient computations for state representatives. In addition, note that although this technique performs approximate verification, all violations and deadlocks were correctly identified.

The size of the state space also influences error detection. A large state space provides more opportunity than a small state space for the randomised search of ProB

[4]We do not apply symmetry marking to error free machines, since it is an approximate technique that does not guarantee checking one state per orbit.

to explore regions that are error free. However, once again, we cannot guarantee this claim, and indeed, experimentation highlights that in some cases errors are discovered more quickly in larger state spaces than smaller ones. Some of the results for standard model checking fall into this category. We also note that this method outperforms the strategies of canonical labels, permutation flooding and canonical labels + symmetry markers, for certain machines with errors. Given that these techniques found the errors after exploring a similar number of states, we can be explain the difference in time by the increased efficiency of generating states in standard checking.

When finding errors, a significant bottleneck experienced by canonical labels, permutation flooding and canonical labels + symmetry markers is that of computing state representatives. Accordingly, the methods exhibit the limitations witnessed during experimentation of machines containing no errors (described earlier in this section). That is, results indicate that the canonical labels approach identifies errors quickly in machines whose symmetries involve mainly relations over deferred sets; permutation flooding identifies errors quickly in machines containing mainly sets of deferred elements; and canonical labels + symmetry markers generally outperforms the canonical label approach due to symmetry marking efficiently distinguishing non-symmetric states.

5.3 Machines used in Experimentation

In this section, we provide a brief description of each machine used in experimentation. The B machines can be found in full in Appendix C.

i.) *scheduler0* defines a process scheduling specification, and is given in Leuschel and Butler [2005], where each process can be in one of three states (*idle*, *ready* or *active*), and the deferred sets are the process identifiers.

ii.) *scheduler0** is a version of (i) in which there is an error in the precondition of the *enter* operation, which leads to a deadlock, where all processes are in the *ready* state.

iii.) *scheduler* is a variation of *scheduler0* and is taken from Legeard et al. [2002]. The deferred sets it uses are also process identifiers, as in (i).

iv.) *RussianPostalPuzzle* is a specification of a cryptographic puzzle [Flannery and Flannery, 2001], which involves safely transmitting a message between two parties. The deferred sets are sets of available keys and locks.

v.) *phonebook* is a slightly more elaborate version of the phonebook defined in Figure 4.2, and has two additional variables; *active*, which represents the set of names in the phonebook, and *activec*, which represents the set of numbers in the phonebook. The B specification is given in Figure 1.3.

vi.) *phonebook** is the *phonebook* specification with an under-constrained precondition of the *delete* operation, which leads to an invariant violation.

vii.) *FileSystem* is an abstract specification of a filesystem in a Windows NT operating system. There are three deferred sets used that represent sets of system users, file names and file identifiers. This machine has been developed as part of the separate research of Damchoom (a Ph.D student in the Dependable Systems and Software Engineering group, at Southampton University, UK). Operations enable one to login to the system, or logout, and perform basic file operations including; creation, deletion, copy, move, rename, read, write, create directory, delete directory, change directory and list the contents of a directory.

viii.) *FileSystem** is a version of the *FileSystem* specification, which contains an invariant violation that arises when certain *read* operations do not observe write-locks. This is caused by an under-constrained *read* precondition.

ix.) *DiningPhilosophers* is a B machine of the famous dining philosophers problem given in Dijkstra [1971], where deferred sets represent the forks and philosophers.

x.) *Peterson's* is a B encoding of Peterson's mutual exclusion problem for n processes, as presented in Peterson [1981], where deferred sets are the process identifiers.

xi.) *Peterson's** is an alternate specification of (x), in which invariant violations arise when two processes occupy the critical region at the same time. This is caused by a small error in the body of the *wait_until* operation (which determines if a process can enter the critical region), where the binary operator, $\leq$ is used instead of $<$.

xii.) *HotelKeys* is a refinement machine that models the use of key cards for room access in a hotel, whose abstract specification is based on the machine given in [Jackson, 2006b, Section E.2.2, page 308]. The deferred sets are the sets of keys, cards, rooms and guests. The abstract specification also specifies deferred sets representing the rooms and guests.

In addition to the machine descriptions, we also provide details of the data types of these machines that exhibit symmetry (i.e., involving deferred sets), which will be exploited by our symmetry reduction method. This will aid the analysis of the results from experimentation, presented in the following sections. Table 5.2 presents the types of variables used by the machines, which involve deferred sets[5]. We denote deferred sets, $DS1 \ldots DSn$, and enumerated sets (including Boolean values and integers), E. For example, *scheduler0* contains a variable that is an element of $\mathbb{P}(DS1)$ (i.e., a subset of process identifiers). This machine also contains a total function, whose domain is a

[5]Machines containing errors, e.g., *scheduler0**, have the same type definitions as their error free specification. Therefore, these machines do not appear in Table 5.2.

subset of $DS1$, and whose range is an enumerated set. Note that deferred sets with the same name, in the same machine, indicate the same deferred set. However, deferred sets with the same name, in different machines, do not indicate the same deferred set.

Machine	Types of Constants/Variables involving Deferred Sets
scheduler0	$\mathbb{P}(DS1), \mathbb{P}(DS1) \rightarrow E$
scheduler	$\mathbb{P}(DS1), \mathbb{P}(DS1), \mathbb{P}(DS1)$
Russian Postal Puzzle	$\mathbb{P}(DS1), \mathbb{P}(DS1), E \rightarrow \mathbb{P}(DS1)$
phonebook	$DS1 \rightarrow DS2, \mathbb{P}(DS1), \mathbb{P}(DS2)$
FileSystem	$DS1(\times 2), DS2(\times 2), DS3,$ $\mathbb{P}(DS1)(\times 3), DS1 \rightarrow DS3$
DiningPhilosophers	$DS1 \rightarrowtail\!\!\!\!\!\rightarrow DS2(\times 2), DS2 \rightarrowtail\!\!\!\!\!\rightarrow DS1$
Peterson's	$DS1 \rightarrow E(\times 2), DS1 \nrightarrow E,$ $E \nrightarrow DS1$
HotelKeys	$DS1 \nrightarrow DS2, \mathbb{P}(DS3), \mathbb{P}(DS4),$ $DS4 \rightarrowtail DS3(\times 2), DS4 \rightarrow DS1,$ $DS2 \rightarrowtail DS3(\times 2)$

TABLE 5.2: Deferred set information for machines tested

5.4 Identifying the Absence of Errors

The following section presents the empirical data obtained from applying each of the following symmetry reduction techniques to the error free B machines, described in Section 5.3: canonical labels, permutation flooding and canonical labels + symmetry markers. The results from standard model checking in PROB are used as a control. For reliability, the data shown is the average taken over 3 tests.

5.4.1 Canonical Labels versus Standard Model Checking

In Table 5.3 we present the results from applying symmetry reduction in PROB via *canonical labels* to eight B specifications. For each specification, we vary the cardinality of deferred sets (Cd) and record the time required for model checking to terminate. The table also shows the number of states and transitions in the reachable state space, with and without (wo) symmetry reduction, and the speedups obtained with the reduction technique.

The results obtained are encouraging. As can be seen, symmetry reduction reduces verification time up to some point, in each machine. Also, there is a large reduction in the number of states and transitions. The most prominent savings are for the *phonebook* machine, where a linear increase in size of deferred sets leads to a combinatorial saving in time, for $1 \leq n \leq 6$, where n is the number of elements in each deferred set; see

Machine	Cd	Time		Speed-	States		Trans	
		wo (s)	with	up	wo	with	wo	with
scheduler0	1	0.01	0.01	1.0	5	5	6	6
	2	0.05	0.05	1.0	16	10	37	23
	3	0.26	0.15	1.7	55	17	190	59
	4	1.24	0.52	2.4	190	26	865	121
	5	7.78	2.20	3.5	649	37	3646	216
	6	35.35	12.03	2.9	2188	50	14581	351
scheduler	1	0.01	0.01	1.0	4	4	5	5
	2	0.03	0.02	1.5	11	7	25	16
	3	0.14	0.08	1.8	36	11	121	37
	4	0.64	0.30	2.1	125	16	561	71
	5	3.02	2.78	1.1	438	22	2418	121
	6	22.93	27.03	0.8	1523	29	10489	190
Russian	1	0.05	0.08	0.6	15	15	24	24
Postal	2	0.31	0.42	0.7	81	48	177	105
Puzzle	3	2.13	2.05	1.0	441	119	1277	331
	4	17.34	9.80	1.8	2325	248	7869	838
	5	158.61	48.83	3.2	11985	459	47795	1826
	6	738.14	266.37	2.8	60981	780	279969	3571
phonebook	1	0.01	0.01	1.0	3	3	4	4
	2	0.06	0.03	2.0	10	5	37	17
	3	0.69	0.17	4.1	65	8	433	50
	4	12.03	1.09	11.0	626	13	6001	125
	5	280.38	7.51	37.3	7777	20	97201	269
	6	13944.97	60.36	231.0	117650	31	1815157	541
File-	1	0.01	0.01	1.0	3	3	4	4
System	2	12.85	2.57	5.0	649	82	2864	365
	3	15588.06	360.09	43.3	362260	1927	2109780	11554
Dining-	2	0.09	0.05	1.8	21	7	52	18
Philosophers	3	2.88	0.28	10.3	337	11	1320	52
	4	167.41	3.20	52.3	11809	20	62496	302
Peterson's	2	0.57	0.47	1.2	49	27	96	52
	3	65.96	16.36	4.0	884	174	2648	518
	4	22125.87	1053.92	21.0	22283	1134	89126	4530
HotelKeys	3	0.03	0.01	3.0	4	2	3	3
	4	10.48	1.23	8.5	718	6	1636	29
	5	3692.92	28.83	128.1	101311	20	444474	271
	6	>15 hrs	617.24	>87.5	>940575	62	>1834527	409

TABLE 5.3: Experimental results for eight B-specifications

the 'Speedup' column in Table 5.3. In the case where deferred sets have a size of 6, reduced checking is 231 times faster than standard checking. In terms of the state space, the quotient model is approximately 3500 times smaller the original model, with respect to both states and transitions. Results for the machines, *Peterson's*, *FileSystem*, *DiningPhilosophers*, and *HotelKeys* also indicate the benefit of applying the reduction technique, observing speedups of approximately 20, 40, 60 and 130 respectively; and

in which case the quotient models are no less than 50 times smaller than the original models.

Let us examine further the *DiningPhilosophers* machine, which is given in Figure 5.2, to see where savings can be made. The machine contains two constants, *lFork* and *rFork*, representing the two forks available to each philosopher. There is also a single variable called *taken*, which indicates for a philosopher the forks they are currently holding, if any.

MACHINE
DiningPhilosophers
SETS
Phil;
Forks
CONSTANTS
lFork,*rFork*
PROPERTIES
$lFork \in Phil \rightarrowtail\!\!\!\!\!\!\rightarrow Forks \wedge$
$rFork \in Phil \rightarrowtail\!\!\!\!\!\!\rightarrow Forks \wedge$
$\mathrm{card}(Forks) = \mathrm{card}(Phil) \wedge$
$\forall (pp).(pp \in Phil \Rightarrow lFork(pp) \neq rFork(pp))$
VARIABLES
taken
INVARIANT
$taken \in Forks \nrightarrow Phil \wedge$
$\forall (xx).(xx \in \mathrm{dom}(taken) \Rightarrow (lFork(taken(xx)) = xx \text{ or } rFork(taken(xx)) = xx))$
INITIALISATION $taken := \varnothing$
OPERATIONS
TakeLeftFork(p,f) = **PRE** $p \in Phil \wedge f \in Forks \wedge f \notin \mathrm{dom}(taken) \wedge lFork(p) = f$
THEN
$taken(f) := p$
END;
TakeRightFork(p,f) = **PRE** $p \in Phil \wedge f \in Forks \wedge f \notin \mathrm{dom}(taken) \wedge rFork(p) = f$
THEN
$taken(f) := p$
END;
DropFork(p,f) = **PRE** $p \in Phil \wedge f \in Forks \wedge f \in \mathrm{dom}(taken) \wedge taken(f) = p$
THEN
$taken := \{f\} \lhd taken$
END
END

FIGURE 5.2: The Dining Philosophers Machine

The use of canonical labels during model checking determines that a single state can represent all possible instantiations of the constants. Subsequent checking then needs

only to check the successors of the representative state, in contrast to standard checking, which must check the successors of every different valid instantiation of the constants. The effect is made more pronounced given that redundant checking of symmetric states requires the analysis of the universal quantification, $\forall(xx).(xx \in \mathrm{dom}(taken) \Rightarrow (lFork(taken(xx)) = xx \text{ or } rFork(taken(xx)) = xx))$, in the invariant of this machine; which specifies that each philosopher may only take their designated forks. Thus, for this machine, a substantial speedup is observed, e.g., 52.3 times faster than standard checking when there are 4 *Fork*s and 4 *Phil*s.

Predicting the magnitude of a speedup for some machine is non-trivial, although the results show that a major factor is the amount of symmetry exhibited by a system, and the data structures that induce them. Table 5.3 shows that machines involving symmetric relational types witness better speedups than machines with only symmetric sets, e.g., compare the *phonebook* with the *scheduler* machine. This difference in speedup can be explained by the size of orbits of states for such machines. Table 5.1 illustrates the number of instances of a relation, including all symmetries, over two deferred sets, with cardinality $k < 6$. Standard model checking may need to generate and check all $2^{k \times k}$ instances of such a relation, to exhaust the state space. In comparison, there are vastly fewer isomorphically distinct relations (as k increases), which the canonical labels approach analyses. Therefore, we see that the benefits of searching a reduced state space can outweigh the overhead of computing canonical labels. One should note that the specific type of the relational value can also influence the speedups possible, since this affects the number of symmetries. For example, an injective function over a deferred set (of size k) and an enumerated set, has $k!$ different instances (all symmetric). Therefore, there are fewer symmetries to exploit. The smaller speedup for such cases contribute towards the results for the *DiningPhilosophers* machine. Similar reasoning explains the smaller speedups observed for machines involving only values that are sets of deferred elements, which may have up to 2^k instances of a set, as with the *scheduler* machine.

Another factor is the structure of state graphs representing a B state, which are used in the computation of canonical labels. Given a partition for a graph, recall that Algorithm 3 applies partition refinement successively to find a set of discrete partitions (vertex orderings) from which it can determine the canonical label. Each application of partition refinement aims to generate a significantly finer partition, to reduce the number of partitions analysed in this computation. Partition refinement fails to meet this goal when it is difficult to distinguish the non-symmetry of vertices in the graph; in which case, more partitions must be analysed, and finding the canonical label requires more time. Therefore, the structure of the state graph for a B state influences the speed of finding a canonical label. We can identify that state graphs involving sets of deferred elements can be difficult to find a canonical label for. For example, consider a state containing a single variable, $v = \{d_1, d_2, d_3, d_4\}$, where d_1, d_2, d_3, d_4 are elements

of the same deferred set. The corresponding state graph consists of 4 vertices of colour $col1$, each with a single directed edge, labelled with v, whose target vertex is the *root*, of colour $col2$ (refer to Section 4.3 for more information on state graphs). Since the vertices representing the deferred elements have the same number of in-coming and out-going edges, and their neighbours are the same, partition refinement cannot distinguish any non-symmetry between them (because they are symmetric). Given the initial partition, $[\{root\}, \{d_1, d_2, d_3, d_4\}]$, Algorithm 3 must therefore analyse all 4! discrete partitions finer than it, to find the canonical label. In contrast, state graphs for relational types may not have such regular structure, which will enable partition refinement to establish any non-symmetry between vertices, reducing the number of discrete partitions to be analysed, and reducing the time required to compute a canonical label.

In Figure 5.3, we illustrate exactly how the speedup varies with the size of deferred set, for a selection of the machines. Data points correspond to the cardinality of deferred set used, i.e., the n^{th} point on a line shows the speedup for the n^{th} set of results in Table 5.3, for a particular machine.

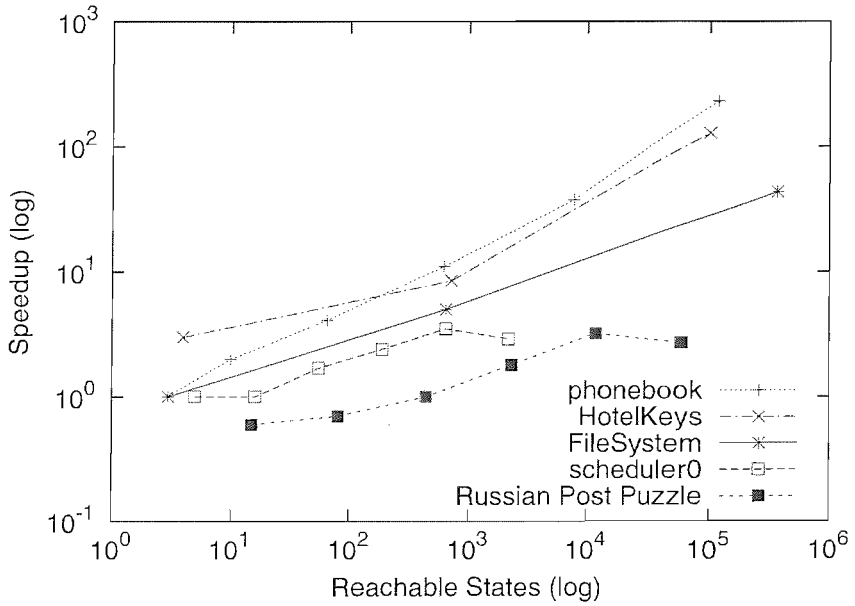


FIGURE 5.3: Variation of speedups with cardinality of deferred sets

As can be seen, the reduction method can be very effective, with speedups exceeding an order of magnitude and increasing linearly (or greater) with respect to the size of the state space in certain cases, as illustrated by the lines for the *phonebook*, *HotelKeys* and *FileSystem* machines. However, in some cases, a speedup drop off is observed, as with the *scheduler0* and *RussianPostalPuzzle* machines.

Speedup drop off points occur when the benefits of using canonical labelling is outweighed by the overhead of their computation. To study this problem, and reveal any

programming inefficiencies in our implementation of this procedure, the execution of computing a canonical label has been analysed using the Gauge profiling module in SICStus Prolog. The results show that approximately 72% of the computation time is spent accessing Prolog terms that model arrays (AVL-Trees)[6]. This figure could be *significantly* reduced if we had available data structures with constant-time access, such as C-language arrays. Additionally, several major optimisations used in nauty are not used by our algorithm (discussed in Section 4.5 and Section 5.1). The result of any algorithmic optimisation would be to increase the speedups achieved (i.e., increase the gradient of lines in Figure 5.3) and delay the speedup drop off points.

5.4.2 Canonical Labels versus Permutation Flooding

In this section, we compare symmetry reduction via canonical labels to *permutation flooding* [Leuschel et al., 2007], introduced in Section 2.8.5. Recall that in this method, the problem of identifying representatives is viewed from a different perspective. Instead of computing the representative of a newly encountered state, s, all symmetric permutations of s are computed, and added to the state space; while marking s as the representative. This is relatively unproblematic for B's data structures since it involves only the permutation of deferred set elements. Empirical results confirm this approach is effective in practice; in some cases, speedups exceed an order of magnitude. In addition, there are cases where it outperforms canonical labels. Table 5.4 presents the results of applying permutation flooding to the eight machines tested in the previous section, to illustrate the differences between the two strategies.

Observe that the numbers of states and transitions computed are the same as those when canonical labels are used. Note however, that this is because Table 5.4 only counts representative states: permutation flooding still generates all state symmetries via symmetric permutations (and therefore will generate the same number of states checked by standard model checking).

As with the canonical labels approach, the magnitude of the speedups in Table 5.4 depends largely on the B machine being checked. For the machines, *Peterson's*, *DiningPhilosophers*, *FileSystem* and *scheduler0*, we see speedups of approximately 20, 35, 40 and 65 respectively over standard exhaustive checking. Those obtained for the *RussianPostalPuzzle*, *scheduler0* and *scheduler* machines are actually greater than those found when using symmetry reduction via canonical labels. However, observe that when permutation flooding substantially outperforms the canonical labels method, the times involved are typically less than 1 minute, e.g., for 4 deferred set elements, the *scheduler* machine is 64.0 times faster than standard checking, whereas canonical labels is only 2.1 times faster; although, the actual times are still only, 0.01s and 0.30s, respectively. Note that as the number of deferred elements increases, the speedups tend to drop off,

[6]By modelling arrays as lists, this figure increases to approximately 78% of the computation time.

Machine	Permutation Flooding (*pf*)				Speedup	
	Cd	Time(s)	States	Trans	*pf*	*cl*
scheduler0	1	0.01	5	6	1.0	1.0
	2	0.04	10	23	1.3	1.0
	3	0.09	17	59	2.9	1.7
	4	0.26	26	121	4.8	2.4
	5	0.92	37	216	8.5	3.5
	6	5.65	50	315	6.3	2.9
scheduler	1	0.01	4	5	1.0	1.0
	2	0.02	7	16	1.5	1.5
	3	0.01	11	37	14.0	1.8
	4	0.01	16	71	64.0	2.1
	5	0.10	22	121	30.2	1.1
	6	0.98	29	190	23.4	0.8
Russian	1	0.03	15	24	1.7	0.6
Postal	2	0.18	48	105	1.7	0.7
Puzzle	3	0.71	119	331	3.0	1.0
	4	3.04	248	838	5.7	1.8
	5	21.91	459	1826	7.2	3.2
	6	153.86	780	3571	4.8	2.8
phonebook	1	0.01	3	4	1.0	1.0
	2	0.02	5	17	3.0	2.0
	3	0.08	8	50	8.6	4.1
	4	0.81	13	125	14.9	11.0
	5	22.21	20	269	12.6	37.3
	6	909.78	31	541	15.3	231.0
FileSystem	1	0.01	3	4	1.0	1.0
	2	2.01	82	365	6.39	5.0
	3	374.58	1927	11554	41.6	43.3
Dining-	2	0.03	7	18	3.0	1.8
Philosophers	3	0.17	11	52	16.9	10.3
	4	10.93	20	302	15.3	52.3
Peterson's	2	0.33	27	52	1.72	1.2
	3	14.52	174	518	4.5	4.0
	4	1126.10	1134	4530	19.6	21.0
HotelKeys	3	0.01	2	3	3.0	3.0
	4	1.29	6	29	8.1	8.5
	5	295.55	20	271	12.5	128.1
	6	OM	-	-	-	>87.5

TABLE 5.4: Comparison of speedups: flooding (*pf*) vs canonical labels (*cl*)

as with the results for canonical labels. This occurs when the benefits of only checking representative states becomes outweighed by the time required to generate all symmetric permutation states.

On the other hand, there are cases where reduction through canonical labels outperforms permutation flooding, e.g., the *phonebook*, *FileSystem*, *DiningPhilosophers*, *Peterson's*

and *HotelKeys* machines. In fact, it is found that reduction via canonical labels can be an order of magnitude faster than permutation flooding, as with the machines of the *phonebook* (231.0 versus 15.3) and *HotelKeys* (128.1 versus 12.5).

It is possible to identify factors that contribute to the variation in speedups between canonical labels and permutation flooding. Results indicate that a significant factor is the amount of symmetry within a system. In the previous section, we have discussed how machines generally exhibit less symmetry when they use mainly variables whose types are deferred sets, when compared to machines with variables of relational types over deferred sets. Given that permutation flooding generates all symmetric states, results show that the technique outperforms canonical labels for machines exhibiting fewer symmetries, such as with the *scheduler* and *RussianPostalPuzzle* machines, where the effectiveness of an efficient permutation procedure can be seen. However, as the number of symmetries within a state space increases, the overhead of generating all state symmetries becomes more prominent, and the benefits of canonical labels becomes evident – as with the *phonebook*, *DiningPhilosophers* and *HotelKeys* machines.

The explicit storage of states, including representative states and their orbits, also affects the performance of permutation flooding. A relatively extreme consequence of this is witnessed for the *HotelKeys* machine, when deferred sets have cardinality 6. That is, symmetry reduced checking fails for permutation flooding since all available RAM on the test PC (up to 1GB) is consumed during the generation of permutation states. This is indicated by 'OM' in Table 5.4. However, by storing only representative states, symmetry reduction via canonical labels succeeds for this machine and terminates with a speedup of > 87.5. Only a lower bound of the speedup can be given since standard exhaustive checking for the *HotelKeys* machine exceeded the maximum test time of 15 hours.

5.4.3 Canonical Labels versus Canonical Labels + Symmetry Markers

Introduced in Section 4.7, *canonical labels + symmetry markers* is a technique that combines symmetry markers with reduction via canonical labels. The culmination is a technique that aims to gain the benefits of both approaches; an *exact* approach that does not always require the use of a graph isomorphism algorithm to compute unique representatives. This section tests this conjecture by applying the method to the B machines previously tested for the canonical labelling (Section 5.4.1) and permutation flooding techniques (Section 5.4.2). The results obtained from experimentation are given in Table 5.5.

Empirical results indicate the effectiveness of the combined technique. For the machines of *phonebook*, *DiningPhilosophers* and *HotelKeys*, the method performs very well, with speedups over standard checking of 238.4, 478.3 and 103.5 respectively. Indeed, for these

Machine	Labels and Markers ($cl + sm$)				Speedup		
	Cd	Time(s)	States	Trans	cl	pf	$cl + sm$
scheduler0	1	0.01	5	6	1.0	1.0	1.0
	2	0.05	10	23	1.0	1.3	1.0
	3	0.14	17	59	1.7	2.9	1.9
	4	0.47	26	121	2.4	4.8	2.6
	5	1.77	37	216	3.5	8.5	4.4
	6	9.38	50	315	2.9	6.3	3.8
scheduler	1	0.01	4	5	1.0	1.0	1.0
	2	0.01	7	16	1.5	1.5	3.0
	3	0.01	11	37	1.8	14.0	14.0
	4	0.33	16	71	2.1	64.0	1.9
	5	2.44	22	121	1.1	15.1	1.2
	6	24.64	29	190	0.8	12.7	0.9
Russian	1	0.01	15	24	0.6	1.7	5.0
Postal	2	0.33	48	105	0.7	1.7	0.9
Puzzle	3	1.56	119	331	1.0	3.0	1.4
	4	7.40	248	838	1.8	5.7	2.3
	5	36.86	459	1826	3.2	7.2	4.3
	6	200.46	780	3571	2.8	4.8	3.7
phonebook	1	0.01	3	4	1.0	1.0	1.0
	2	0.03	5	17	2.0	3.0	2.0
	3	0.18	8	50	4.1	8.6	3.8
	4	1.01	13	125	11.0	14.9	11.9
	5	6.97	20	269	37.3	12.6	40.2
	6	58.50	31	541	231.0	15.3	238.4
FileSystem	1	0.01	3	4	1.0	1.0	1.0
	2	2.98	82	365	5.0	6.39	4.3
	3	380.59	1927	11554	43.3	41.6	41.0
Dining-	2	0.04	7	18	1.8	3.0	2.3
Philosophers	3	0.12	11	52	10.3	16.9	24.0
	4	0.35	20	302	52.3	15.3	478.3
Peterson's	2	0.05	27	52	1.2	2.0	1.2
	3	17.08	174	518	4.0	4.5	3.9
	4	1166.87	1134	4530	21.0	19.6	19.0
HotelKeys	3	0.01	2	3	3.0	3.0	3.0
	4	1.06	6	29	8.5	8.1	9.9
	5	26.19	20	271	128.1	12.5	141.0
	6	521.64	62	409	>87.5	-	>103.5

TABLE 5.5: Comparison of speedups: canonical labels (cl) vs flooding (pf) vs canonical labels + symmetry markers ($cl + sm$)

machines the technique outperforms both symmetry reduction via canonical labels and permutation flooding. Given that the presence of symmetry is established by canonical labels, also note that the sizes of the state spaces are the same as those for the canonical label technique.

As with reduction via canonical labels, this method is outperformed by permutation flooding for the *scheduler0*, *scheduler* and *RussianPostalPuzzle* machines. However, it is only noticeably outperformed when the times are less than 1 minute. These results indicate that symmetry markers do not distinguish non-symmetry for a large proportion of the reachable states, and therefore canonical labels must be computed. The reasons given in the previous section, describing the differences in performance between canonical labels and permutation flooding, with respect to these machines, can then be applied here.

In contrast, the results for the *DiningPhilosophers* machine, show a speedup of approximately 10 times that of the approach using canonical labels alone (478.3 compared to 52.3). We can therefore deduce that many of the states encountered during model checking can be distinguished using symmetry markers, which are less expensive to compute than canonical labels. Thus, this example highlights the value of the combined technique.

The *Peterson's* machine produces results that are slightly worse than both canonical labels and permutation flooding. A possible reason is that computation of symmetry markers and/or canonical labels may be complicated, and many states have orbits containing more than 1 state. Therefore, both symmetry markers and canonical labels are frequently computed.

5.5 Identifying the Presence of Errors

In this section we present the empirical data obtained from applying each of the symmetry reduction techniques to the B machines containing errors, described in Section 5.3: canonical labels, permutation flooding, symmetry markers and canonical labels + symmetry markers. The results from standard model checking in ProB are used as a control. Given that ProB uses a randomised search algorithm, and therefore we cannot guarantee the path to an error found, each experiment is conducted only once (no average is taken). Results then serve to illustrate the possible performances of each technique.

The results from standard model checking are presented in Table 5.6. We see that standard model checking identifies errors in under 10s for the *scheduler0**, *phonebook** and *FileSystem** machines, while searching substantially fewer states and transitions than the error free versions of these machines, e.g., for *FileSystem** and deferred set size 3, the error is found after 4.46s, 439 states, and 818 transitions; corresponding results for the error free machine are 15588.06s, 362260 states and 2109780 transitions.

We might expect the time to find errors to increase, as we increase the size of the deferred sets (and therefore the size of the state space). However, this is not always the case, as with the *phonebook** machine for deferred set sizes 5 and 6: the former

(5 deferred elements) requires 0.08s, 185 states and 211 transitions, whereas, the latter requires 0.01s, 131 states, and 156 transitions. This is an effect of the randomised search algorithm, and indeed, which error in the state space it finds (if there is more than one).

Machine	Standard Checking			
	Cd	Time(s)	States	Trans
*scheduler0**	1	<0.01	5	6
	2	<0.01	16	34
	3	0.20	55	179
	4	0.10	98	136
	5	4.77	649	3429
	6	7.12	1933	4719
*phonebook**	1	<0.01	3	4
	2	<0.01	10	19
	3	<0.01	35	68
	4	0.01	46	57
	5	0.08	185	211
	6	0.01	131	156
*FileSystem**	2	2.19	218	566
	3	4.46	439	818
*Peterson's**	2	0.21	27	32
	3	13.53	443	758
	4	179.24	586	598

TABLE 5.6: Identifying errors using Standard Checking

Results from the canonical labels method are given in Table 5.7. As with standard checking, the errors are found in less than 10s for the *scheduler*0*, *phonebook** and *FileSystem** machines. In fact, for 5 deferred elements, the *scheduler*0* error is found almost 3 seconds faster (2.07s versus 4.77s). A significant factor here, however, is likely to be the larger section of the state space (649 states, 3429 transitions), which standard checking has to cover, in comparison to the section covered by canonical labels (32 states, 190 transitions). Canonical labels do not always lead to the discovery of errors more quickly than standard checking, as seen with the machines other than *scheduler*0*. For example, for *FileSystem** and 3 deferred elements, canonical labels takes 7.06s - 4.46s = 2.60s longer than the standard strategy. Given that canonical labels reach 439 - 158 = 281 fewer states and 828 - 387 = 441 fewer transitions, we may infer that the bottleneck experienced is that of computing canonical forms for states. A more prominent example of this appears with the results for the *Peterson's** machine, where canonical labels takes over 100s longer to discover the error.

Table 5.8 presents the data from experimentation when using permutation flooding. Comparing the results with those for the canonical labels method, we see that permutation flooding identifies the error for the *scheduler*0* machine almost 2s quicker (when there are 6 deferred set elements). We may explain this using reasoning given in the previous section, i.e., the efficiency of generating the symmetric states dominates the

Machine	Canonical Labels			
	Cd	Time(s)	States	Trans
*scheduler0**	1	<0.01	5	6
	2	<0.01	10	21
	3	0.01	17	40
	4	0.52	20	106
	5	2.07	32	190
	6	6.95	42	282
*phonebook**	1	<0.01	3	4
	2	0.03	4	15
	3	0.09	7	41
	4	0.90	10	108
	5	0.20	10	88
	6	1.92	19	178
*FileSystem**	2	1.01	60	163
	3	7.06	158	387
*Peterson's**	2	0.17	19	22
	3	9.01	160	308
	4	281.62	274	746

TABLE 5.7: Identifying errors using Canonical Labels

overhead of computing canonical labels. In contrast, as illustrated by the *phonebook** machine results, the generation of permutation states requires more time when there are more symmetric states, and therefore the canonical label approach is quicker. Let us now analyse the results for the *Peterson's** machine. We note that permutation flooding finds an error almost 94s quicker than the canonical labels approach. This appears to be a surprising result, especially given that it contains 4 functions, each over a deferred set and an enumerated set. However, we must note the canonical labels method checks an extra 78 states, and 291 transitions. Additionally, the time that this requires is exaggerated by the use of two universal quantifications in the machine (one in the invariant and one in an operation).

In Table 5.9, we present the results obtained using symmetry markers. Recall that this method performs approximate verification by efficiently generating a symmetry marker for states, such that symmetric states have the same symmetry marker; although in some cases, non-symmetric states may also have the same marker. First, experimentation shows that this method finds the errors in all the B machines. Also, we see that it outperforms all of the techniques discussed so far in this section: thus, indicating the value of using an efficient computation to find representatives. The errors for the *scheduler*0*, *phonebook** and *FileSystem** machines were found in under 1s. Regarding the *Peterson's** machine, the error is found approximately 50s faster than permutation flooding, and 140s faster than the canonical labels technique. In comparison to standard checking, it is 40s faster – note however, that the symmetry markers approach covers in excess of 200 fewer states.

Machine	Permutation Flooding			
	Cd	Time(s)	States	Trans
*scheduler0**	1	<0.01	5	6
	2	0.02	10	18
	3	0.01	17	49
	4	0.02	22	113
	5	0.82	31	186
	6	5.04	46	345
*phonebook**	1	0.01	3	4
	2	0.03	4	12
	3	0.11	6	42
	4	1.67	10	102
	5	2.82	12	175
	6	7.58	10	152
*FileSystem**	2	1.08	37	89
	3	4.25	33	102
*Peterson's**	2	0.31	27	36
	3	7.75	172	508
	4	187.92	201	455

TABLE 5.8: Identifying errors using Permutation Flooding

Machine	Symmetry Markers			
	Cd	Time(s)	States	Trans
*scheduler0**	1	<0.01	5	6
	2	<0.01	10	18
	3	0.06	17	44
	4	0.17	24	146
	5	0.32	35	202
	6	0.58	48	259
*phonebook**	1	<0.01	3	4
	2	<0.01	4	13
	3	<0.01	6	42
	4	<0.01	10	115
	5	<0.01	12	183
	6	0.01	19	322
*FileSystem**	2	0.24	44	86
	3	0.36	110	98
*Peterson's**	2	0.28	27	52
	3	6.84	174	518
	4	142.62	263	625

TABLE 5.9: Identifying errors using Symmetry Markers

Finally, Table 5.10 presents the results from canonical labels + symmetry markers. The results observed are as one might expect. The method identifies errors slightly quicker than the canonical label method, when the number of states and transitions covered are similar, as with the *scheduler0** machine. Also, we see the worth of the

combined technique for the *Peterson's** machine, where the error is found about 40s faster, even though 31 more states and 67 more transitions are covered. When compared to permutation flooding and standard model checking, the dominant issues concern those for canonical labels, as described previously (since experimentation shows the computation of symmetry markers is efficient). Similarly, comparing this approach to symmetry markers, we again look to the bottlenecks experienced by canonical labels, since this constitutes the key difference.

Machine	Canonical Labels + Symmetry Markers			
	Cd	Time(s)	States	Trans
*scheduler0**	1	<0.01	5	6
	2	<0.01	10	21
	3	0.01	17	45
	4	0.97	22	107
	5	2.48	34	202
	6	6.42	40	327
*phonebook**	1	<0.01	3	4
	2	0.01	4	12
	3	0.16	7	44
	4	1.03	11	102
	5	0.35	12	175
	6	1.89	15	155
*FileSystem**	2	0.60	41	91
	3	3.68	79	212
*Peterson's**	2	0.41	21	46
	3	14.86	163	459
	4	246.67	305	813

TABLE 5.10: Identifying errors using Canonical Labels + Symmetry Markers

5.6 Summary

In this chapter, we have presented an evaluation of the four techniques for symmetry reduction in PROB, with an emphasis on illustrating the performance of our canonical labelling technique, described in Section 4. Experimentation constituted application of the different techniques to a range of B machines to provide reliable performance information.

The empirical data obtained overall is encouraging and shows for each technique that speedups over standard model checking can exceed at least an order of magnitude, when applied to certain B machines. An Executive Summary of the data is given in Section 5.2.

Specifically, one observes that symmetry reduction via *canonical labels* performs notably well on machines involving a large amount of symmetry. Often, this is a consequence of the use of relations, and any relational derivatives, over 1 or 2 deferred sets. For such

machines, speedups can be 2 orders of magnitude (the maximum recorded is 231). Results also highlight the method is influenced by the structure of state graphs, from which canonical labels are computed. In particular, more time is required to compute canonical labels when partition refinement cannot easily establish non-symmetry between the vertices of a state graph.

Results show that *permutation flooding* is more effective than canonical labels for machines containing variables whose values do not involve relations over deferred sets, e.g., subsets of deferred sets. When such relations are involved, the system generally has more symmetric states, which permutation flooding must generate. We believe that the generation of the numerous symmetric states means permutation flooding is outperformed by canonical labels. In the converse situation, when systems exhibit fewer symmetries, the overhead of computing canonical labels becomes the dominant factor, and permutation flooding outperforms canonical labels. Note that the speedups recorded for permutation flooding do not exceed 1 order of magnitude (the maximum is 64), and in the cases where it notably outperforms canonical labels, the times involved are less than 1 minute.

The combination of our technique and symmetry markers, which we call *canonical labels + symmetry markers*, is shown to generally perform better than the canonical labels technique alone. This demonstrates the value of using symmetry markers to efficiently distinguish non-symmetric states. Experimentation shows speedups can exceed 450, when compared to standard checking (or 10 times faster than canonical labels). In comparison to permutation flooding, the performance bottlenecks appear to be those identified for canonical labels.

Regarding error detection, we find that the most effective method is the efficient approximate verification performed by *symmetry markers*. Therefore, this technique is recommended for use after making changes to a B specification. Canonical labels, permutation flooding, and canonical labels + symmetry markers are also effective at identifying the errors; however, we believe their drawback mainly concerns the expense of computing representatives. In fact, results indicate that one may choose standard model checking over these reduction techniques, since the randomised search algorithm used in ProB appears particularly effective at identifying error states. We recommend using these symmetry reduction methods once there is confidence that fewer errors exist in the specification e.g., symmetry markers fails to find an error, or standard checking finds no error after a significant amount of time, such as 5 minutes[7]. In which case, we will see the benefits of the symmetry reductions, i.e., exploring a reduced state space, with the guarantee that errors will not be missed.

In the next chapter, we formalise our algorithm for symmetry reduction via canonical labels, and show that it is sound with respect to the algorithm for standard model

[7]Given that standard checking took no longer than 3 minutes to find the errors during experimentation.

checking. The formalisation uses B refinement, where, for a single abstract machine, two chains of refinement specify the standard and reduced approaches.

Chapter 6

Correctness of Algorithms

6.1 Introduction

In Chapters 2, 4 and 5 of this thesis, the focus is the development and evaluation of a technique that performs symmetry reduction in model checking in B. This method relies on symmetric states in B satisfying the same predicates, as described in Section 4.2, and the use of a canonical labelling algorithm to identify symmetric states. Complementary to this, it is very important to guarantee the soundness of symmetry reduction via canonical labels, with respect to standard model checking. For example, if standard model checking exhausts its search space without finding a counterexample, then symmetry reduced checking must exhaust its constrained search space without finding a counterexample. This chapter addresses this issue using formal specifications in B, for both methods, that constitute soundness proofs for symmetry reduction by refinement.

The formal specifications have been developed and proved interactively using the free, single-user version of the Atelier-B tool-set, called B4Free, and its graphical interface, Click'n Prove [Abrial and Cansell, 2003]. They consist of an abstract B machine specifying the generalised goal of model checking, from which two separate chains of refinement specify the standard and reduced approaches. The following sections of this chapter present the machines that constitute the two chains. For clarity of the 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, e.g., if an operation in a refinement machine modifies its more abstract specification. Full versions of these machines can be found in Appendix D. The abstract specification is given first, in Section 6.2.

6.2 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 6.1.

MACHINE
mc0

SETS
S;
$ANSWER = \{Pass, Fail\}$

CONSTANTS
i, /* *special initial state* */
tr, /* *transition relation* */
inv, /* *states satisfying invariant* */
$reach$ /* *reachable states* */

PROPERTIES
$tr \in S \leftrightarrow \mathrm{S} \wedge$
$inv \in \mathbb{P}(S) \wedge$
$i \in inv \wedge$
$i \notin \mathrm{ran}(tr) \wedge$

/* *the reachable states* */
$reach \in \mathbb{P}(S) \wedge$
$i \in reach \wedge$
/* *reach is a fix point* */
$tr[reach] \subseteq reach \wedge$

/* *reach is the smallest fix point of the reachable states* */
$\forall(r).(r \in \mathbb{P}(S) \wedge$
$i \in r \wedge$
$tr[r] \subseteq r \Rightarrow$
$reach \subseteq r)$

FIGURE 6.1: The Sets, Constants and Properties of the Abstract Machine, *mc0*

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 it uses). Given that parameters are placed on system types in practice, $|S|$ 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 counterexample), or *Fail* (a reachable counterexample has been found).

There are four important constants used for the abstract specification. 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 model checking of B systems in ProB, since checking involves only safety constraints on variables. 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 include all initial states of the system. It follows that i is always the root state of the search space. The set of states encountered during model checking, denoted $reach$, can be defined by a fixpoint on tr, where $tr[reach] \subseteq$ reach, i.e., the successor of any reachable state is also reachable. Further, we specify $reach$ as the smallest fixpoint of tr.

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

$ok \leftarrow fail \mathrel{\widehat{=}}$
WHEN $reach \not\subseteq inv$
THEN $ok := Fail$
END
END

FIGURE 6.2: The Operations of the Abstract Machine, *mc0*

The operations of *mc0* are given in Figure 6.2. These include the operations, *pass* and *fail*, which are mutually exclusive events that specify the conditions under which model checking terminates. The *pass* operation is enabled (forever) 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 (forever) if the set of reachable states do not satisfy the correctness conditions. In which case, the *Fail* message is output by the algorithm. In contrast to an implementation 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*.

In the next section, the first level of refinement of *mc0* is given.

6.3 Refinement Level 1

Let us now present *mc1*, the first level of refinement for *mc0*, denoted $mc0 \sqsubseteq mc1$. 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 future refinements can specify further their precise behaviour. Figure 6.3 presents the new variables, invariant and initialisation clauses of the refinement machine, *mc1*.

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 of this machine, *mc1* introduces two events used during the *traversal* of the state space in model checking. 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

REFINEMENT
mc1

REFINES
mc0

VARIABLES
rac, /* *reached and checked* */
err

INVARIANT
$rac \subseteq reach \wedge$
$rac \subseteq inv \wedge$
$i \in rac \wedge$
$err \subseteq reach - inv$

INITIALISATION
$rac := \{i\} \parallel$
$err := \varnothing$

FIGURE 6.3: The Variables, Invariant and Initialisation of the *mc1* refinement machine

counterexamples. We separate the events for state space traversal since we find this style convenient for proof, and presentation. The operations of *mc1* are given in Figure 6.4.

OPERATIONS
$add\_inv \mathrel{\widehat{=}}$
ANY *ss* **WHERE**
$ss \subseteq reach\text{-}rac \wedge$
$ss \subseteq inv \wedge$
$ss \neq \varnothing$
THEN
$rac := rac \cup ss$
END;

$add\_err \mathrel{\widehat{=}}$
ANY *ss* **WHERE**
$ss \subseteq reach\text{-}rac \wedge$
$ss \neq \varnothing \wedge$
$ss \cap inv = \varnothing$
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 $reach \not\subseteq inv$
THEN
$ok := Fail$
END
END

FIGURE 6.4: The Operations of the *mc1* refinement machine

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*. Let us assume all reachable states satisfy the correctness conditions. Then, it follows that after an exhaustive search of the reachable states, we have the condition, $rac \subseteq inv$. Using this, one can refine the *pass* operation to become enabled when $reach \subseteq rac$.

The *add_err* operation selects a non-empty subset from the reachable states, which are yet to be reached, but which contain no element present in the set of states satisfying the correctness conditions, i.e., are invariant violations. These violations are added to *err* for a permanent record.

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, $| reach - (rac \cup err) |$, 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 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[1]. However, we have provided a variant here to help illustrate the validity of *mc1*.

6.4 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 abstracted away 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 6.5 shows the variables, invariant and initialisation clauses of this machine.

REFINEMENT
mc2

REFINES
mc1

VARIABLES
unex, /* *reached, checked and not explored* */
rac, /* *reached and checked* */
err /* *reached errors* */

INVARIANT
$unex \subseteq rac \wedge$
$tr[rac\text{-}unex] \subseteq rac \cup err$

INITIALISATION
$unex := \{i\} \;||$
$rac := \{i\} \;||$
$err := \varnothing$

FIGURE 6.5: The Variables, Invariant and Initialisation of the *mc2* refinement 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 initial state i, and subsequently checked.

[1]The new generation of B, called Event-B [Métayer et al., 2005, Abrial et al., 2006], and its provers provide support for variants.

In addition, note that a new invariant condition is added: $tr[rac - unex] \subseteq rac \cup err$. 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 6.6, and the assertions proved for this machine[2], given in Figure 6.7.

OPERATIONS

add_inv $\widehat{=}$
ANY *s1*,*s2* **WHERE**
$s1 \in unex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \in inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$s2 \notin rac \wedge$
$err = \varnothing$
THEN
$unex := unex \cup \{s2\} \parallel$
$rac := rac \cup \{s2\}$
END;

add_err $\widehat{=}$
ANY *s1*,*s2* **WHERE**
$s1 \in unex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$s2 \notin rac \wedge$
$err = \varnothing$
THEN
$err := err \cup \{s2\}$
END;

remove $\widehat{=}$
ANY *s1* **WHERE**
$s1 \in unex \wedge$
/* *all successors of s1 have been checked* */
$\mathrm{tr}[\{s1\}] \subseteq rac \wedge$
$err = \varnothing$
THEN
$unex := unex - \{s1\}$
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
END

FIGURE 6.6: The Operations of the *mc2* refinement machine

Regarding the operations of this machine, *remove* is introduced 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, $| unex |$, 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

[2]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 for the machine, e.g., discharging proof obligations.

add_err operation. Addition to either *unex* or *rac* would, otherwise, break the invariant, $unex \subseteq rac \wedge rac \subseteq inv$.

ASSERTIONS

/* *add_inv* */
$\exists(s1,s2).(s1 \in unex \wedge$
$s2 \in ran(\mathrm{tr}) \wedge$
$s2 \in inv \wedge$
$s1 \mapsto s2 \in \mathrm{tr} \wedge$
$s2 \notin rac \wedge$
$err = \varnothing) \vee$

/* *add_err* */
$\exists(s1,s2).(s1 \in unex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$s2 \notin rac \wedge$
$err = \varnothing) \vee$

/* *remove* */
$\exists(s1).(s1 \in unex \wedge$
$tr[\{s1\}] \subseteq rac \wedge$
$err = \varnothing) \vee$

/* *pass* */
$(unex = \varnothing \wedge$
$err = \varnothing) \vee$

/* *fail* */
$(err \neq \varnothing)$

FIGURE 6.7: The Assertions of the *mc2* refinement machine

A number of assertions are also specified in *mc2*, to verify that the specified model checking algorithm is deadlock free, with respect to the enabled operations. The assertions are comprised of a disjunction of the guards of each operation. Therefore, given their interactive proof in B4Free, it is guaranteed 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 (forever).

Given the deadlock-freeness of the 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 more abstract specification, which we have proved interactively using the B prover. We now illustrate the proof, which will involve the invariant condition introduced into this machine, $tr[rac - unex] \subseteq rac \cup err$, see Figure 6.5. Let us assume that model checking reaches state, s, where $s \in inv$ and no counterexamples have been found so far (e.g., $s = i$):

i.) if all successors of s satisfy the invariant, all will be explored eventually as they will be added to *unex* in *add_inv*. In the case where all reachable states satisfy the correctness conditions, all successors will be explored, such that eventually $unex = \varnothing$ and $err = \varnothing$, and *pass* enables. This implies, $tr[rac - \varnothing] \subseteq rac \cup \varnothing$. By the fix point property of *reach* (see the last property in Figure 6.1, for *mc*0), we have $reach \subseteq rac$: the guard of *pass* in *mc1*. Therefore, *pass* in *mc2* is a valid refinement of its more abstract specification in, *mc1*.

ii.) Otherwise, if a successor of s is an error state, it is added to *err* (in the *add_err* operation), and *fail* enables. We prove this operation refines its more abstract specification, using $err \neq \varnothing$ and $err \subseteq reach - inv$, which imply there exists a reachable state that violates the invariant, i.e., the guard of *fail* in *mc1*, $reach \not\subseteq inv$.

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 use B to specify the notion of symmetry reduction in model checking.

6.5 Refinements for Symmetry Reduced Model Checking

This section presents two refinement machines that specify symmetry reduced model checking through the refinement of *mc1* (Section 6.3), namely *rmc1* and *rmc2*. These refinements follow closely the specification of *mc2*, except they utilise the concept of symmetry between states of a system.

6.5.1 Level 1

In the first refinement, we introduce the notion of state symmetries into the constants and properties clauses, as presented in Figure 6.8.

Symmetries are defined by automorphisms over the transition relation, and are modelled as a set of permutations (bijections) over states, denoted *aut*. Also specified are two key properties of automorphisms, as given in [Clarke et al., 1999, Chapter 14];

- an inverse of an automorphism is itself an automorphism, and
- automorphisms preserve the transition relation (a result also shown in Section 4.2).

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 Section 4.2, 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.

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. Our implementation of this function determines a unique representative from each orbit of states by computing canonical labels (see *canon* in Definition 4.4). It follows that checking one state during the reduced search, corresponds to checking all symmetric states in the standard search.

REFINEMENT
rmc1

REFINES
mc1

CONSTANTS
/ automorphisms on tr */*
aut,
/ representative function */*
rep

PROPERTIES
$aut \in \mathbb{P}(S \rightarrowtail\!\!\!\!\!\!\rightarrow S) \wedge$
$id(S) \in aut \wedge$
$\forall(p).(p \in aut \Rightarrow p^{-1} \in aut) \wedge$
$\forall(p).(p \in aut \Rightarrow i \mapsto i \in p) \wedge$

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

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

$rep \in S \rightarrow S \wedge$

/ 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. */*
$\forall(s1,s2).(s1 \mapsto s2 \in rep \Rightarrow$
$\exists(p).(p \in \text{aut} \wedge s1 \mapsto s2 \in p)) \wedge$

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

FIGURE 6.8: The Constants and Properties of the Machine, *rmc1*

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 6.9, whose proof simplifies later proof activities required to guarantee its consistency and show that it is a valid refinement of *mc1*.

ASSERTIONS
/ representatives preserve invar. */*
$\forall(s1,s2).(s1 \in S \wedge$
$s2 \in S \wedge$
$s1 \mapsto s2 \in rep \Rightarrow$
$((s1 \in inv) \Leftrightarrow (s2 \in inv))) \wedge$

$rep(i) = i \wedge$
$rep^{-1}[\{i\}] = \{i\} \wedge$

$\forall(s1,s2).(s1 \mapsto s2 \in tr \Rightarrow$
$\exists(ss2).(rep(s1) \mapsto ss2 \in tr \wedge$
$rep(s2) = rep(ss2))) \wedge$

/ s is reachable iff*
*rep(s) is reachable */*
$\forall(s).(s \in S \Rightarrow$
$((s \in reach) \Leftrightarrow (rep(s) \in reach)))$

FIGURE 6.9: The Assertions of the *rmc1* refinement machine

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 (see Equation 6.4) requires proof that for any reachable state its representative state is also reachable. This is provable using induction and three specified properties of automorphisms, given again in Equations 6.1 – 6.3. Proof by induction is difficult using the B provers,

therefore, we describe an inductive proof in the following[3]:

$$\textit{Property: } \forall(p, s1, s2).(p \in aut \wedge s1 \in S \wedge s2 \in S \Rightarrow (s1 \mapsto s2 \in tr) \Leftrightarrow (p(s1) \mapsto p(s2) \in tr)) \tag{6.1}$$

$$\textit{Property: } \forall(p).(p \in aut \Rightarrow (i \mapsto i \in p)) \tag{6.2}$$

$$\textit{Property: } \forall(s1, s2).(s1 \mapsto s2 \in rep \Rightarrow \exists(p).(p \in aut \wedge s1 \mapsto s2 \in p)) \tag{6.3}$$

$$\textit{Assertion: } \forall(s).(s \in S \Rightarrow ((s \in reach) \Leftrightarrow (rep(s) \in reach))) \tag{6.4}$$

Proof. By Equation 6.1 we have $i \mapsto s \in tr \Leftrightarrow p(i) \mapsto p(s)$, for all automorphisms p. We extend this idea to sequences of states. Let $\gamma = s_0 \rightarrow s_n$, denote any sequence of states $s_0, \ldots, s_n$, where $s_0 = i$ (Equation 6.2) and $s_j \mapsto s_{j+1} \in tr$, for $0 \leq j < n$. Then, by Equation 6.1, for all automorphisms p there is a sequence of states $p(\gamma) = p(s_0) \rightarrow p(s_n)$, which denotes the sequence $p(s_0), \ldots, p(s_n)$, where $p(s_0) = i$ (Equation 6.2) and $p(s_j) \mapsto p(s_{j+1}) \in tr$, for $0 \leq j < n$. Given that representatives are automorphisms of states (Equation 6.3), it follows that a state s is reachable if and only if $rep(s)$ is also reachable. □

VARIABLES

/ vars for standard checking */*
rac,unex,err,

/ vars for reduced approach */*
rrac,runex,rerr

INVARIANT

$unex \subseteq rac$

$rrac \subseteq ran(rep) \wedge$
$rrac \subseteq rac \wedge$
$runex \subseteq rrac \wedge$
$rerr \subseteq err \wedge$

$rep^{-1}[rrac] = rac \wedge$
$rep^{-1}[runex] = unex \wedge$
$rep^{-1}[rerr] = err \wedge$

$tr[rac\text{-}unex] \subseteq rac \cup err$

INITIALISATION

$rac := \{i\} \;||\; rrac := \{i\} \;||$
$unex := \{i\} || \; runex := \{i\} ||$
$err := \varnothing \;||\; rerr := \varnothing$

FIGURE 6.10: The Variables, Invariant and Initialisation of the *rmc1* refinement machine

Six variables are used by this machine. Intuitively, they can be split into three pairs of variables, where each pair consists of a variable used in the B specification of standard model checking (*rac*, *unex* or *err*), and a corresponding variable introduced to specify

[3]Inductive proofs are not directly supported by the B provers, although they are possible through the introduction of extra properties and assertions. This carries the risk of introducing errors into the specification that cannot identified.

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, $tr[rac - unex] \subseteq rac \cup err$ 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 6.4). This will be proved correct in the next refinement using only *rrac*, *runex*, and *rerr*. The operations of *rmc1* are given next, in Figure 6.11.

OPERATIONS

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

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

$ok \leftarrow pass \widehat{=}$
WHEN
$reach \subseteq rep^{-1}[rrac]$
THEN
$ok := Pass$
END;

$ok \leftarrow fail \widehat{=}$
WHEN
$reach \not\subseteq inv$
THEN
$ok := Fail$
END

END

FIGURE 6.11: The Operations of the *rmc1* refinement machine

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*. (Note the similarity this technique shares with that of permutation flooding, described in Section 2.8.5.) The *remove* operation follows this trend, 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*. The variant, $\mid reach - (rrac \cup rerr) \mid + \mid runex \mid$, can be provided to show that these operations will block eventually, when all representative states have been encountered. In which case, *pass* or *fail* may become enabled. Finally, we see that the guards on the operations of *pass* and *fail* are equivalent to those for the corresponding operations in *mc*1 (Figure 6.4).

Note that the intention of this refinement machine is to integrate symmetry into the B specification of standard model checking (*mc2*) and link the variables used by the standard and reduced approaches. It is not intended as a specification from which an implementation may be constructed. However, this could be achieved for the next refinement, which abstracts away the details of the standard approach.

6.5.2 Level 2

Having specified a relatively detailed refinement machine in *rmc*1, the last refinement needs only retain three variables *rrac*, *runex* and *rerr*, upon which we specify a minimal set of constraints. The variables and invariant of this machine are presented in Figure 6.12.

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
$rrac := \{i\} \parallel$
$runex := \{i\} \parallel$
$rerr := \varnothing$

FIGURE 6.12: The Variables, Invariant and Initialisation of the *rmc2* refinement machine

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 abstracted away. The same applies for the operations of this machine, which are given in Figure 6.13.

OPERATIONS

add_inv $\widehat{=}$
ANY *s1,s2* **WHERE**
$s1 \in runex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \in inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$rep(s2) \notin rrac \wedge$
$rerr = \varnothing$
THEN
$runex := runex \cup \{rep(s2)\} \parallel$
$rrac := rrac \cup \{rep(s2)\}$
END;

add_err $\widehat{=}$
ANY *s1,s2* **WHERE**
$s1 \in runex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$rep(s2) \notin rrac \wedge$
$rerr = \varnothing$
THEN
$rerr := rerr \cup \{rep(s2)\}$
END;

remove $\widehat{=}$
ANY *s1* **WHERE**
$s1 \in runex \wedge$
/* *all successors of s1 have been checked* */
$rep[tr[\{s1\}]] \subseteq rrac \wedge$
$rerr = \varnothing$
THEN
$runex := runex - \{s1\}$
END;

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

ok $\leftarrow$ *fail* $\widehat{=}$
WHEN
$rerr \neq \varnothing$
THEN
$ok := Fail$
END
END

FIGURE 6.13: The Operations of the *rmc2* refinement machine

Similar to *mc2*, a disjunction of the operation guards are specified as assertions, and proved, to guarantee that one of the operations is enabled at all times in this refinement machine. The assertions of *rmc2* are given in Figure 6.14.

Using reasoning analogous to that in *mc2*, the deadlock-freeness of operations for this machine, together with the variant described in *rmc2*, showing *add_inv*, *add_err* and *remove* eventually block, it is ensured that either *pass* or *fail* will become enabled. If no reachable counterexample exists, *pass* will become enabled eventually. We can show that its guard, $runex = \varnothing$ and $rerr = \varnothing$ implies that termination should occur, since $(runex = \varnothing \Rightarrow unex = \varnothing)$ and $(rerr = \varnothing \Rightarrow err = \varnothing)$ implies $tr[rac - \varnothing] \subseteq rac \cup \varnothing$. Therefore, using the fixpoint property in Figure 6.1, we have $reach \subseteq rac$. That is, the reachable state space has been reached and checked, without finding an error. In the contrary situation, whenever a reachable error is found, it is added to *rerr* (in *add_err*). Therefore, *fail* will become enabled. We can show that the guard of *fail*, $rerr \neq \varnothing$ implies an error exists, using $rerr \neq \varnothing \Rightarrow err \neq \varnothing$, and $err \subseteq reach - inv$. Therefore $reach \not\subseteq inv$: a reachable error exists.

The chain of refinement for symmetry reduced model checking consists of: $mc0 \sqsubseteq mc1 \sqsubseteq$

ASSERTIONS

/* *add_inv* */
$\exists(s1,s2).(s1 \in runex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \in inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$rep(s2) \notin rrac \wedge$
$rerr = \varnothing) \vee$

/* *add_err* */
$\exists(s1,s2).(s1 \in runex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$rep(s2) \notin rrac \wedge$
$rerr = \varnothing) \vee$

/* *remove* */
$\exists(s1).(s1 \in runex \wedge$
$rep[tr[\{s1\}]] \subseteq rrac \wedge$
$rerr = \varnothing) \vee$

/* *pass* */
$(rerr = \varnothing \wedge$
$runex = \varnothing) \vee$

/* *fail* */
$(rerr \neq \varnothing)$

FIGURE 6.14: The Assertions of the *rmc2* refinement machine

$rmc1 \sqsubseteq rmc2$. Therefore, symmetry reduced model checking is sound with respect to the abstract specification of model checking. Moreover, by identifying a single representative from each class of symmetric states, model checking with symmetry reduction need only analyse a subset of the states that constitute the model of the system.

6.6 Summary

This chapter presents a B specification of the type of model checking performed by ProB, namely that of checking safety properties in B systems, for both standard model checking, and the symmetry reduced approach developed in this thesis.

An abstract specification for model checking is given in *mc0* in Section 6.2, which is refined by *mc1* in Section 6.3. From *mc1*, two separate chains of refinement specify certain details of algorithms that may implement standard and reduced approaches to model checking. Given that both chains have been proved using the interactive theorem prover, B4Free, it follows that both approaches are sound with respect to the original abstract specification for model checking, defined in *mc0*.

The specification of the standard approach consists of a single refinement. This machine requires 3 key variables, and has properties that indicate the method eventually terminates having found a counterexample (*fail*) or without finding counterexamples (*pass*), after exploring the reachable state space.

The specification of the reduced approach comprises two refinements. The first refinement makes use of the set of variables used in the standard approach (*mc2*), and links them to an analogous set of variables to be used in the reduced approach. The second

refinement then retains only the variables used by a reduced model checking approach. Symmetries are defined by automorphisms over the transition relation of the B system being checked, and symmetries are exploited using a function, called *rep*, which computes for some state, s, a unique representative from the orbit of s. In practice, our implementation of a representative function makes use of an algorithm that computes canonical labels for graphs representing B states, as described in Chapter 4. As with the specification of standard model checking, its properties indicate the method eventually terminates either having found counterexamples (*fail*) or without finding counterexamples (*pass*). In the latter case, all of the constrained reachable search space will have been explored.

Chapter 7

Conclusions and Future Work

This thesis describes the development of a number of techniques that improve the process of model checking B systems. These techniques include visual state space reductions that enable one to view various succinct properties about the reachable states of a system, in addition to the first approach to classical symmetry reduction in B. The use of each technique has been illustrated, formalised, and integrated into the ProB tool-set, to produce a model checker with several novel features. Subsequent evaluations indicate the effectiveness of each approach.

The main contribution to visual state space reductions include two algorithms that can be applied to state spaces explored by model checking in ProB. The resulting graphs, which are generally significantly smaller in size, can then be visualised. The premise of this research is that model checking often encounters state spaces too large to be viewed; although, visual feedback can be very beneficial to the understanding of a system since human perception is adept at identifying structural similarities and symmetries. A significant factor for the reductions obtained by both methods is an abstraction function, called α, that diminishes the size of the transition relation of the original state space, by discarding B operation parameters. Further to this, the DFA-Abstraction algorithm applies the classical determinisation algorithm of finite automata to merge states while preserving trace information, which is generally useful to a user. The Signature-Merge algorithm, on the other hand, generates larger reductions by merging all states with the same outgoing transitions (signature). The minimised graph then has the useful property that any trace that is not possible in the reduced graph also is not possible in the original graph; or, if a deadlocked state exists in the reduced graph, then one also exists in the original graph. Since Leuschel and Turner [2005], these algorithms have been extended by diminishing the α-abstraction so that it only abstracts away certain arguments. This is guided by the user, to give more control over the reductions possible. In addition, a new graphical user interface of ProB has been developed, and algorithms have been added for viewing sets of states satisfying a user defined expression in B, or subgraphs between two states chosen by a user, e.g., initial states and error states.

Given a graph generated by Signature-Merging, the signature of a node can be viewed as all traces of length 1 that can be performed from that node. This can be improved in the future by extending the notion of a signature and comparing all traces of length $2, 3, \ldots, n$, where n is specified by the user. This enables more detailed trace information to be read from graphs generated by the Signature-Merge method. Another improvement is to make the α-abstraction less precise to achieve greater reductions. For instance, it could be made more aggressive by mapping several operations together, which the user may not be interested in. Alternatively, Signature-Merge could be modified so that nodes are merged if signatures are sufficiently similar, as opposed to requiring identical signatures. Furthermore, it would be possible to combine both of these methods: the user could request a certain number as a target for the ideal number of nodes, and then the graph is progressively made less/more precise to approach that number.

This thesis also contributes the first technique to achieve classical symmetry reduction for model checkers of B specifications, using an algorithm for identifying graph isomorphism. This technique is called, symmetry reduction via Canonical Labels. The line of research that has developed symmetry reduction techniques based on scalarsets is the inspiration for this approach. The symmetries exploited by our method [Turner et al., 2007] are induced by the deferred set; a key component of the B language. Therefore, in contrast to the scalarset approach, this technique is fully automatic. The algorithm computes representatives in two phases. First, it computes the state graph of a state, s. This is followed by computation of the canonical label of this state graph, which corresponds to the unique representative of s. This thesis describes the translation of B states to state graphs, and presents the relation of symmetric B states to isomorphic state graphs. It presents the extension to the underlying algorithm of nauty, developed for computing canonical labels for state graphs, so that symmetric B states can be identified. Furthermore, we describe the integration of these techniques into standard model checking in ProB, to produce the method of symmetry reduction via Canonical Labels.

The research on symmetry reduction via Canonical Labels has stimulated the development of three more strategies for symmetry reduction in B, including Permutation Flooding [Leuschel, Butler, Spermann, and Turner, 2007], Symmetry Markers [Leuschel and Massart, 2007], and combined Canonical Labels + Symmetry Markers. An evaluation has been provided for all of these techniques. The empirical results obtained highlight their effectiveness, where speedups over standard checking can exceed two orders of magnitude for Canonical Labels and Canonical Labels + Symmetry Markers, or one order of magnitude for Permutation Flooding. More specifically, results suggest Symmetry Markers or standard checking should be used after making changes to a B specification (e.g., during the early stages of development), to quickly identify any errors. However, as confidence increases in fewer errors existing (e.g., Symmetry Markers fails to find errors, or standard checking finds no errors after 5 minutes), it is recommended that Canonical Labels/Canonical Labels + Symmetry Markers/Permutation Flooding

is used (depending on the symmetric data types used, discussed in Chapter 5), where the benefits of symmetry reduction can be more pronounced, i.e., exploring a reduced state space, with the guarantee that errors are not missed. We also note that it would be possible to generalise these methods for application to model checkers of other formal languages, e.g., using a translation that constructs state graphs for a specific language.

Future work aims to increase the speedups of computing representatives by optimising the implementation of the canonical labelling algorithm, so that its performance is closer to that of nauty. Immediate changes include translating Prolog procedures into C/Java, while utilising automorphism pruning, and using only two array variables to represent branches of the search tree, to significantly improve performance by reducing memory consumption and computation time. In addition, the partition refinement step during computation of a canonical label can be optimised to better suit the properties of state graphs in B, and break more symmetries in each step. We may also devise a new technique for constructing state graphs, for which partition refinement is better suited. For both, we can use saucy [Darga et al., 2004] as inspiration, which optimises nauty for CNF formulae. One improvement is to use the fact that edges never originate from the root vertex, or indeed from the set of vertices, X, introduced to represent nested values. Therefore, the algorithm is not required to compute d_{out} for any vertex $v \in X \cup \{root\}$.

Proof of soundness of symmetry reduction via Canonical Labels involves proving the soundness of three of the techniques it uses. First, we have formalised the type of symmetry exploited, and proved that the symmetric states induced are indeed symmetric, as defined by Clarke et al. [1999]. Second, it is required that symmetry reduced model checking suffices to construct quotient models progressively by checking a unique state from each orbit. This is proved using refinement in B. A separate chain of refinement is specified for both standard model checking in ProB, and the symmetry reduced approach. Since both chains refine the abstract specification of model checking, which is also specified, it can be deduced that our symmetry reduction is sound with respect to standard model checking. This contribution is novel in the field of B. In addition, the use of B and refinement provides an alternate (and simplified) reference for the concepts used during model checking and symmetry reduction. Finally, it should be proved that the implementation of the extended version of McKay's canonical labelling algorithm will assign the same label to two states if and only if the states are indeed symmetric (according to our definition). This is a complex task that involves formally expressing the numerous concepts and optimisations used by the algorithm. This is yet to be proved and remains as future work.

The final proposal for future work regards an interesting, and more importantly, successful symmetry reduction technique, used notably by the Alloy Analyser; that is, symmetry-breaking. In this system, a SAT solver attempts to instantiate a formula that contains state information. To reduce the number of possible interpretations, symmetry-breaking predicates are added to the formula. Their effect is to *break* symmetries in the

formula by preventing the satisfaction of a large proportion of isomorphic interpretations. Application of the ALLOY approach to systems that do not use SAT solvers, such as explicit state model checkers, is not straightforward. However, its premise highlights new ways that could improve the performance of model checking systems in B using PROB. For example, it could be used to break symmetries during the evaluation of non-deterministic assignments. This includes statements in B such as *ANY* x *WHERE* y, which attempts to find some instance of x, where y constrains x. Optimisations could be gained by a scheme that analyses the form of y, and inserts into it extra predicates that break symmetries of x, so that x may take only a small number of values. The effect is to prune certain symmetric states from the state space, and reduce memory consumption during verification. As with the ALLOY approach, it is suggested that the new predicates do not break all symmetries except one, as this problem becomes intractable. Instead, it is desirable to break a significant proportion of symmetries, while retaining computational efficiency. This gives the potential for savings in verification time. Extending this idea, symmetry breaking may also be used to compute (multiple) representatives for each orbit. In contrast to the standard approach of performing a computation on a given state, symmetry breaking could be applied when *generating* the values of variables for a new state. It is proposed that a preprocess analyses the types of the system variables, and constructs automatically the corresponding symmetry breaking predicates, using the methods of Shlyakhter [2007] as inspiration. Let us illustrate this idea by examining a simple example. Consider a B machine containing variable, $p \in DS \rightarrow E$, where DS is a deferred set and E is an enumerated set. The bipartite graph for p, denoted, $p_{\mathcal{B}}$, has isomorphisms (symmetries of p), which can be obtained by permuting the rows of the adjacency matrix for $p_{\mathcal{B}}$ – since this corresponds to permuting elements of DS. (Here we assume the adjacency matrix has a y-axis comprised of the elements of DS, and whose x-axis is comprised of elements of E.) Then, a predicate that requires *sorted* rows in this matrix will break many symmetries of p, while generating a normal form for use in the state. Should a new symmetry-breaking scheme be employed, the constraint logic module of SICStus Prolog should facilitate an implementation into the PROB tool-set. The strategies proposed would be novel in the field of model checking of the B language. Moreover, background research indicates they hold a large scope for their application and can be effective.

Appendix A

Finding a Canonical Label: A Worked Example

A.1 A Worked Example

The following section contains a worked example of finding a canonical label for the graph, $G = (V, E)$ given in Figure A.1. More information on this procedure can be found in McKay [1981], Kocay [1996] and Kreher [1998].

The main structure used within this example is the ordered partition: a list of disjoint subsets of the vertices of the graph, V, whose union is V. In brief, the algorithm works by *refining* and *fixing/splitting* the partitions, to result in a characteristic set of discrete partitions. Initially, all vertices are assumed to be symmetric, and are therefore placed in the same cell, in the *unit* partition. This is then refined to an equitable partition. If this partition is not discrete – the first non-singleton cell is *fixed* in an attempt to make it non-equitable; this entails removing one element from this cell and putting it into its own cell, and leaving the remainder. So splitting a partition whose first non-singleton cell has a size of 3, will result in 3 different partitions. From here, the new partitions created from splitting are refined. Refinement and splitting continues iteratively until a set of discrete partitions are obtained. When finished, the algorithm chooses the least matrix of the characteristic partitions as the canonical label of the graph.

The refinement algorithm makes use of two variables: π, the partition value throughout the procedure, and α, the list of cells to check against (as used by Algorithm 2, which formalises partition refinement).

The steps of the algorithm are now described, and can be seen in the search tree of Figure A.2:

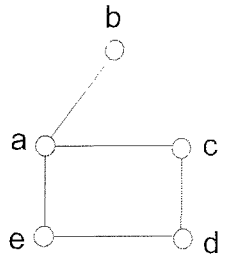


FIGURE A.1: A simple graph

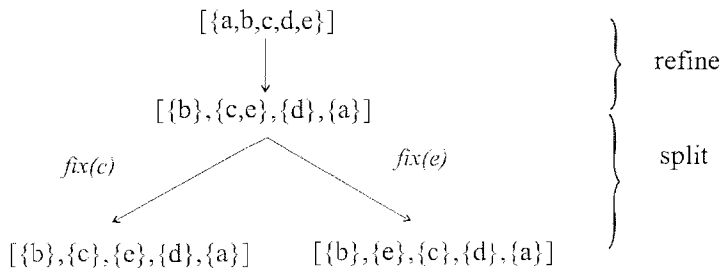


FIGURE A.2: An example search tree generated by *stabilise*($[\{a, b, c, d, e\}]$), for the graph in A.1.

1.) $\tilde{\pi} = [\{a^3, b^1, c^2, d^2, e^2\}]$ (unit partition), $\alpha = [\{a, b, c, d, e\}]$: To begin with, we establish the number of vertices in the cell in α, which are adjacent to vertices in the cell of $\tilde{\pi}$. The number of adjacencies are indicated by superscripts above each vertex $v \in S$, where S is a cell in $\tilde{\pi}$. For example, a^3 denotes that a is neighbours with 3 vertices in $\{a, b, c, d, e\}$, i.e., b, c, e. In the next step, we split $\tilde{\pi}$ into three cells, ordered by the adjacencies shown, and we remove the cell we used in α, and append to α the new cells just created.

2.) $\tilde{\pi} = [\{b\}, \{c, d, e\}, \{a\}]$, $\alpha = [\{b\}, \{c, d, e\}, \{a\}]$: Now, perform a similar computation, to find the number of adjacencies of vertices in cell, $\{b\}$, with the first cell of α, $\{b\}$. This does not lead to any splitting ($\{b\}$ is already discrete). Comparing the first cell of α to the other cells of $\tilde{\pi}$ also causes no splitting. Hence, we remove the first cell of α.

3.) $\tilde{\pi} = [\{b\}, \{c^1, d^2, e^1\}, \{a\}]$, $\alpha = [\{c, d, e\}, \{a\}]$: The first cell of α, $\{c, d, e\}$, causes $\tilde{\pi}$ to be split, so update $\tilde{\pi}$ accordingly. Also, remove the first cell of α, and append to it the newly created cells.

4.) $\tilde{\pi} = [\{b\}, \{c, e\}, \{d\}, \{a\}]$, $\alpha = [\{a\}, \{c, e\}, \{d\}]$: The first cell of α, $\{a\}$, does not cause any cell of $\tilde{\pi}$ to by split, so remove it from α.

5.) $\tilde{\pi} = [\{b\}, \{c, e\}, \{d\}, \{a\}]$, $\alpha = [\{c, e\}, \{d\}]$: The first cell of α, $\{c, e\}$, does not cause any cell of $\tilde{\pi}$ to by split, so remove it from α.

6.) $\tilde{\pi} = [\{b\}, \{c, e\}, \{d\}, \{a\}]$, $\alpha = [\{d\}]$: The first cell of α, $\{d\}$, does not cause any cell of $\tilde{\pi}$ to by split, so remove it from α.

7.) $\tilde{\pi} = [\{b\}, \{c, e\}, \{d\}, \{a\}]$, $\alpha = []$: Now, we have that α is empty, and so $\tilde{\pi}$ is equitable. However, it is not yet discrete, so we must apply fixing, to generate two finer partitions.

8a.) $\tilde{\pi} = [\{b\}, \{c\}, \{e\}, \{d\}, \{a\}]$, $\alpha = [\{b\}, \{c\}, \{e\}, \{d\}, \{a\}]$: In this partition, c has been fixed. In doing so, $\tilde{\pi}$ is now discrete so we stop searching this branch of the tree.

8b.) $\tilde{\pi} = [\{b\}, \{e\}, \{c\}, \{d\}, \{a\}]$, $\alpha = [[\{b\}, \{e\}, \{c\}, \{d\}, \{a\}]]$: In this partition, e has been fixed. In doing so, $\tilde{\pi}$ is now discrete so we stop searching this branch of the tree.

The search produces two discrete partitions, $[\{b\}, \{c\}, \{e\}, \{d\}, \{a\}]$ and $[\{b\}, \{e\}, \{c\}, \{d\}, \{a\}]$ – whose adjacency matrices are shown in Figures A.1 and A.2.

	b	c	e	d	a
b	0	0	0	0	1
c	0	0	0	1	1
e	0	0	0	1	1
d	0	1	1	0	0
a	1	1	1	0	0

TABLE A.1: Adjacency matrix of $[\{b\}, \{c\}, \{e\}, \{d\}, \{a\}]$

	b	e	c	d	a
b	0	0	0	0	1
e	0	0	0	1	1
c	0	0	0	1	1
d	0	1	1	0	0
a	1	1	1	0	0

TABLE A.2: Adjacency matrix of $[\{b\}, \{e\}, \{c\}, \{d\}, \{a\}]$

As can be seen, the matrices of both discrete partitions are the same (this is not always the case) – therefore they must both identify the lowest adjacency matrix. Hence, the canonical label of the graph G, is 000010001100011011001100, where the value is taken as the concatenation of rows of the matrix, from left to right, top to bottom[1].

[1] Given that these adjacency matrices are the same, the permutation between the two partitions, "swap c for e", is an automorphism of the graph.

Appendix B

Detailed Results from Experimentation of Visualisation Algorithms

Table B.1 presents all of the results obtained from testing, including the precise number of states and transitions found in the original and reduced state spaces.

TABLE B.1: Numbers of States and Transitions in original and reduced state space.

Machine Name	Original State Space				Signature Merge		DFA Abstraction	
	States	Transitions	Self Loops	Distinct transitions	States	Transitions	States	Transitions
Ambulances	2552	77271	11110	6	6	18	22	79
Baskets	79	396	120	5	5	8	17	34
B_Clavier_code	3	19	16	6	3	8	4	8
bibliotheque	15	53	10	9	11	31	14	40
B_Site_central	5	24	20	3	3	3	4	3
CarlaTravelAgency	198	226	14	12	18	68	120	178
CarlaTravelAgencyErr	120	139	4	11	16	60	81	99
countdown	3085	5236	0	5	4	5	246	407
Cruise	1361	25695	4716	27	402	4675	16386	231601
CSM	77	209	0	14	64	181	78	209
DAB	5	41	10	2	2	2	4	3
dfa	4	7	4	3	3	4	6	7
dijkstra	7	9	1	3	3	3	7	6
DSP0	49	66	0	7	6	7	8	8
Fermat	17	81	40	3	2	3	10	17
FinalTravelAgency	1079	5608	2156	13	10	32	83	343

Table B.1 – continued from previous page.

Machine Name	Original State Space				Signature Merge		DFA Abstraction	
	States	Transitions	Self Loops	Distinct transitions	States	Transitions	States	Transitions
FunLaws	257	1793	768	10	11	44	52	276
FunLaws	257	2049	64	5	5	13	38	133
GAME	613	4675	3078	10	55	248	201	956
GSM_revue	11	28	10	8	4	8	7	14
Inscription	27	131	97	7	7	21	9	25
inst_adapted	466	3655	112	6	5	15	80	244
Jukebox	40	309	63	6	6	14	490	1906
Level0	769	6147	3072	2	2	2	11	10
m0	65	9923	9408	164	65	9921	98	14913
Main	2	1	0	1	2	1	3	1
mm0	197	2707	2079	18	7	66	86	1097
monitor	81	529	0	5	8	19	32	100
phonebook	65	433	144	4	4	7	6	10
Queues	7	18	12	4	3	4	4	4
Results	6	22	11	5	4	10	5	10
Rubik2	3514	3925	0	4	3	4	3515	3925
RussianPostalPuzzle	441	1227	0	5	9	21	120	274
rw	20	37	0	8	18	35	21	37
scheduler	36	121	0	5	8	17	12	25
SensorNode	5	11	0	2	3	2	4	2

Table B.1 – continued from previous page.

Machine Name	Original State Space				Signature Merge		DFA Abstraction	
	States	Transitions	Self Loops	Distinct transitions	States	Transitions	States	Transitions
SeqLaws	38	58	10	5	6	13	27	59
SetLaws	730	5104	243	7	9	37	127	601
station	28	89	0	5	7	13	8	13
Teletext	14	122	88	12	3	12	15	122
Teletext	25	210	174	12	4	12	12	75
TheSystem	114	123	3	11	16	53	83	86
TransactionsSimple	131	153	0	11	22	51	100	127
TravelAgency	176	191	4	12	16	66	105	120
TravelAgency_trace_check	28610	31012	0	10	95	288	11087	12689
TravelProB	626	3469	492	3	5	9	24	33
UndefinedFunctions	17	193	144	9	5	27	12	73

Appendix C

Machines Used for Empirical Results

This appendix presents the twelve B machines used in Chapter 5 during the evaluation of the symmetry reduction techniques for ProB. Note that in Chapter 5, we use the asterisk (*) to distinguish machines containing errors from their valid versions, e.g., for the *scheduler*0 machine, its version that contains an error is denoted, *scheduler*0*. In practice, we do not use these names since the asterisk is a reserved character in B. Instead, we append '*_err*' on to the base machine name, e.g., *scheduler0_err*.

C.1 Process Scheduler 1

MACHINE *scheduler0*

SETS

$PROC$;
$STATE = \{idle, ready, active\}$

VARIABLES *proc*, *pst*

INVARIANT

$proc \in \mathbb{P}(PROC) \wedge$
$pst \in proc \rightarrow STATE \wedge$

$\text{card}(pst^{-1}[\{active\}]) \leq 1$

INITIALISATION

$proc := \varnothing \;||\; pst := \varnothing$

OPERATIONS

$new(p) \mathrel{\widehat{=}}$

PRE

$p \in PROC - proc$

THEN

$pst(p) := idle \;||$

$proc := proc \cup \{p\}$

END;

$del(p) \mathrel{\widehat{=}}$

PRE

$p \in PROC \wedge$

$pst(p) = idle$

THEN

$proc := proc - \{p\} \;||$

$pst := \{p\} \lhd pst$

END;

$ready(p) \mathrel{\widehat{=}}$

PRE

$p \in PROC \wedge$

$pst(p) = idle$

THEN

$pst(p) := ready$

END;

$enter(p) \mathrel{\widehat{=}}$

PRE

$p \in PROC \wedge$

$pst(p) = ready \wedge$

$pst^{-1}[\{active\}] = \varnothing$

```
    THEN
      pst(p) := active
    END;
  leave(p) ≙
    PRE
      p ∈ PROC ∧
      pst(p) = active
    THEN
      pst(p) := idle
    END
END
```

C.2 Process Scheduler 1 with Error

The B machine is the same as *scheduler0*, except for an error that now occurs in the *enter* operation, which leads to a deadlocked state (when all elements of *PROC* are *ready*).

```
MACHINE scheduler0_err
  ⋮
  enter(p) ≙
    PRE
      p ∈ PROC ∧
      pst(p) = /* MISSING ready */ idle ∧
      pst^-1[{active}] = ∅
    THEN
      pst(p) := active
    END;
  ⋮
END
```

C.3 Process Scheduler 2

MACHINE *scheduler*

SETS *PID*

VARIABLES *active, ready, waiting*

INVARIANT

$active \in \mathbb{P}(PID) \wedge ready \in \mathbb{P}(PID) \wedge waiting \in \mathbb{P}(PID) \wedge$ /* *the types* */

/* *and now the rest of the invariant* */

$active \subseteq PID \wedge$

$ready \subseteq PID \wedge$

$waiting \subseteq PID \wedge$

$(ready \cap waiting) = \varnothing \wedge$

$active \cap (ready \cup waiting) = \varnothing \wedge$

$card(active) \leq 1 \wedge$

$((active = \varnothing) \Rightarrow (ready = \varnothing))$

INITIALISATION

$active := \varnothing \;||\; ready := \varnothing \;||\; waiting := \varnothing$

OPERATIONS

$new(pp) \mathrel{\widehat{=}}$

SELECT

$pp \in PID \wedge$

$pp \notin active \wedge$

$pp \notin (ready \cup waiting)$

THEN

$waiting := (waiting \cup \{\ pp\ \})$

END;

$del(pp) \mathrel{\widehat{=}}$

SELECT

```
      pp ∈ waiting
    THEN
      waiting := waiting - { pp }
    END;
  ready(rr) ≙
    SELECT
      rr ∈ waiting
    THEN
      waiting := (waiting - {rr}) ||
      IF (active = ∅) THEN
        active := {rr}
      ELSE
        ready := ready ∪ {rr}
      END
    END;
  swap ≙
    SELECT
      active ≠ ∅
    THEN
      waiting := (waiting ∪ active) ||
      IF (ready = ∅) THEN
        active := ∅
      ELSE
        ANY pp WHERE pp ∈ ready
        THEN
          active := {pp} ||
          ready := ready - {pp}
        END
      END
    END
END
```

C.4 Russian Postal Puzzle

MACHINE *RussianPostalPuzzle*

SETS

$KeyIDs$;

$PERSONS = \{natasha, boris\}$

/ **DEFINITIONS** */*

/ GOAL == (padlocks =* $\varnothing \wedge$ *box_contains_gem=TRUE* $\wedge$ *hasbox = natasha) */*

VARIABLES

keysforsale, hasbox, padlocks, has_keys, box_contains_gem

INVARIANT

$keysforsale \in \mathbb{P}(KeyIDs) \wedge$

$hasbox \in PERSONS \wedge$

$padlocks \subseteq KeyIDs \wedge$

$has\_keys \in PERSONS \rightarrow \mathbb{P}(KeyIDs) \wedge$

$box\_contains\_gem \in BOOL$

INITIALISATION

$keysforsale := KeyIDs \;||$

$hasbox := boris \;||$

$padlocks := \varnothing \;||$

$has\_keys := \{natasha \mapsto \varnothing,\ boris \mapsto \varnothing\} \;||$

$box\_contains\_gem := TRUE$

OPERATIONS

$buy\_padlock\_and\_key(keyid, person) =$

PRE $keyid \in keysforsale \wedge person \in PERSONS \wedge person = hasbox$

THEN

$has\_keys(person) := has\_keys(person) \cup \{keyid\} \;||$

$keysforsale := keysforsale - \{keyid\}$

END;

add_padlock(keyid,person) =
PRE *keyid*∈*KeyIDs* ∧ *person*∈*PERSONS* ∧
person=hasbox ∧ *keyid* ∈ *has_keys(person)* ∧
keyid ∉ *padlocks*
THEN
padlocks := *padlocks* ∪ {*keyid*}
END;

remove_padlock(keyid,person) =
PRE *keyid*∈*KeyIDs* ∧ *person*∈*PERSONS* ∧
person=hasbox ∧ *keyid*∈ *padlocks* ∧
keyid∈ *has_keys(person)*
THEN
padlocks := *padlocks* - {*keyid*}
END;

send_box(from,to) =
PRE *from*∈ *PERSONS* ∧ *from* = *hasbox* ∧
to ∈ *PERSONS* ∧ to ≠ *hasbox*
THEN
IF *padlocks* = ∅ **THEN**
box_contains_gem := *FALSE*
END ||
hasbox := to
END
END

C.5 Phonebook

MACHINE *phonebook*

SETS
Name;
Code

VARIABLES *db*, *active*, *activec*

INVARIANT

$db \in Name \pfun Code \land$
$active \in \mathbb{P}(Name) \land$
$activec \in \mathbb{P}(Code) \land$
$dom(db) = active \land$
$ran(db) = activec$

INITIALISATION

$db := \varnothing \parallel active := \varnothing \parallel activec := \varnothing$

OPERATIONS

$cc \leftarrow lookup(nn) \mathrel{\widehat{=}}$
PRE
$nn \in Name \land nn \in active$
THEN
$cc := db(nn)$
END;

$add(nn,cc) \mathrel{\widehat{=}}$
PRE
$nn \in Name \land cc \in Code \land nn \notin active$
THEN
$db := db \cup \{ nn \mapsto cc\} \parallel$
$active := active \cup \{nn\} \parallel$
$activec := activec \cup \{cc\}$
END;

$delete(nn,cc) \mathrel{\widehat{=}}$
PRE
$nn \in Name \land cc \in Code \land nn \in active \land cc \in activec \land db(nn) = cc$
THEN
$db := db - \{ nn \mapsto cc\} \parallel$
$active := active - \{nn\} \parallel$
$activec := db[(active - \{nn\})]$
END

END

C.6 Phonebook with Error

The B machine is the same as *phonebook*, except for an error that now occurs in the *delete* operation, which leads to a state violating the invariant (cannot guarantee, $dom(db) = active \wedge ran(db) = activec$).

MACHINE *phonebook_err*
⋮
$delete(nn,cc) \widehat{=}$
PRE
$nn \in Name \wedge cc \in Code \wedge nn \in active \wedge cc \in activec$
/* *MISSING* $\wedge db(nn) = cc$ */
THEN
$db := db - \{ nn \mapsto cc\} \;||$
$active := active - \{nn\} \;||$
$activec := db[(active - \{nn\})]$
END
END

C.7 Windows NT File System

MACHINE *FileSystem*
/* *Author: KRIANGSAK DAMCHOOM;* */
/* *kd06r@ecs.soton.ac.uk;* */
/* *last modified 20 November, 2006* */
/* ———————————————————————————————— */
/* *This machine was written to simulate the model of* */
/* *basic file system (based on windows)* */
/* *which is included basic operations∈* */

/ create, delete, copy, move, rename, read, write */*
/ mkdir, deldir, change directory and list file */*
/ Nature of windows (stand alone) file system∈ */*
/ - Only one user can login at anytime */*
/ - File/directory name in each directory must be unique. */*
/ - One name can appear in more than one directory. */*
/ - Only object's owner can delete and modify the object's details. */*
/ - An object can be written, deleted, renamed, moved */*
/ if it has not been reading or writing */*
/ - An object can be read, copied if it has not been writing. */*
/ - Object mentioned above is instant of file or directory. */*
/ - Each object has properties∈ id, name, location, content and owner */*
/ represented by relationships which are described in invariant section.*/*
/ - Tree structure is used to implement */*
/ ———————————————————————————— */*

SETS

OID; */* object ID (of files/directories) */*
NME; */* object name (of files/directories) */*
USER; */* users */*
$CONTENT = \{ct1\}$ */*set of file content*/*

CONSTANTS

root, rt, sp

PROPERTIES

$root \in OID \wedge$
$rt \in NME \wedge$
$sp \in USER$

VARIABLES

curdir, *located_in*, *owner*, *oid*, *name*, *c_usr*,
readingset, *writingset*, *files*, *directories*,
fcontent, *dcontents*

INVARIANT

$c\_usr \in USER \wedge$

$curdir \in OID \wedge$

$oid \in \mathbb{P}(OID) \wedge$

$files \subseteq oid \wedge$

$directories \subseteq oid \wedge$

$name \in oid \rightarrow NME \wedge$

*/*all objects except root have a location*/*

$located\_in \in oid\text{-}\{root\} \rightarrow oid \wedge$

/ no repeated name in the same directory */*

$\forall o.(o \in directories \Rightarrow \mathrm{card}(located\_in^{-1}[\{o\}]) = \mathrm{card}(name[located\_in^{-1}[\{o\}]])) \wedge$

*/*all objects can map to root based on relative closure*/*

$\forall o.(o \in oid\text{-}\{root\} \Rightarrow o \mapsto root \in \mathrm{closure1}(located\_in)) \wedge$

*/*no cycles in tree structure*/*

$\forall o.(o \in oid \Rightarrow o \mapsto o \notin \mathrm{closure1}(located\_in))$

$owner \in oid \rightarrow USER \wedge$

$fcontent \in files \rightarrow CONTENT \wedge$

$dcontents \in oid \leftrightarrow oid$ */*contents of directories*/* $\wedge$

$readingset \subseteq oid \wedge$

$writingset \subseteq files \wedge$

$writingset \cap readingset = \varnothing \wedge$

$files \cap directories = \varnothing \wedge$

$\forall o.(\ o \in oid \Rightarrow dcontents[\{o\}] = located\_in^{-1}[\{o\}]\ ) \wedge$

INITIALISATION

$c\_usr := \mathrm{sp}$ */* c_usr is a current user */* $||$

$curdir := root$ */* curdir is a current directory */* $||$

$located\_in := \varnothing$ */* no location for root */* $||$

$name := \{root \mapsto \mathrm{rt}\}\ ||$

owner := {*root*↦sp} ||
oid := {*root* ||
files := ∅ ||
directories := {*root*} ||
readingset := ∅ ||
writingset := ∅ ||
fcontent := ∅ ||
dcontents := ∅ ||

OPERATIONS

*/*assume that there is only one user can login at any time*/*

login(usr)=
PRE usr∈*USER*
THEN
SELECT usr ≠ *c_usr*
THEN
c_usr:=usr ||
curdir :=*root* ||
readingset := *readingset*∪{*curdir*}
END
END;

logout(usr)=
SELECT usr=*c_usr*
THEN
curdir:=*root* ||
readingset:=∅ ||
writingset:=∅
END;

crt_file(oi,nn,ct)=
PRE oi∈*OID* ∧ ct∈*CONTENT* ∧ nn∈*NME*
THEN
SELECT
oi∉*oid*

$\wedge$ nn$\notin$*name*[*dcontents*[{*curdir*}]]

THEN

located_in := *located_in* $\cup$ {oi $\mapsto$ *curdir*} ||

dcontents := *dcontents* $\cup$ {*curdir* $\mapsto$ oi} ||

oid := *oid* $\cup$ {oi} ||

name := *name* $\cup$ {oi$\mapsto$nn} ||

owner := *owner* $\cup$ {oi$\mapsto$*c_usr*} ||

files:=*files*$\cup${oi} ||

fcontent(*oi*) := ct

END

END;

rename(oi,old,nn) =

PRE oi$\in$*OID* $\wedge$ nn$\in$*NME* $\wedge$ old$\in$*NME*

THEN

SELECT

oi$\notin$*writingset*$\cup$*readingset* $\wedge$

nn $\notin$ *name*[*dcontents*[{*curdir*}]] $\wedge$

name(*oi*)= old $\wedge$

owner(*oi*)=*c_usr* $\wedge$

located_in(*oi*)=*curdir* $\wedge$

THEN *name*(*oi*):=nn

END

END;

move(oi,nn,dr1,dr2)=

PRE oi$\in$*oid* $\wedge$ nn$\in$*NME*$\wedge$ dr1$\in$*directories*$\wedge$ dr2$\in$*directories*

THEN

SELECT

dr2$\notin$closure1(*dcontents*)[{oi}] $\cup${oi} $\wedge$

oi$\notin$*writingset*$\cup$*readingset* $\wedge$

name(*oi*)=nn $\wedge$

owner(*oi*)=*c_usr* $\wedge$

located_in(*oi*)= dr1 $\wedge$

nn $\notin$ *name*[*dcontents*[{dr2}]]

THEN

located_in(*oi*) := dr2 ||

dcontents:= (*dcontents* - {dr1$\mapsto$oi})$\cup${dr2$\mapsto$oi}

END

END;

copy(oi1,nn1,dr1,oi2,nn2,dr2)=

PRE nn2$\in$*NME* $\wedge$

nn1$\in$*NME* $\wedge$

oi2$\in$*OID* $\wedge$

dr1$\in$*directories* $\wedge$

dr2$\in$*directories* $\wedge$

oi1$\in$*files*

THEN

SELECT

oi2$\notin$*oid* $\wedge$

name(oi1)=nn1 $\wedge$

located_in(oi1)=dr1 $\wedge$

owner(oi1)=*c_usr* $\wedge$

owner(dr2)=*c_usr* $\wedge$

oi1 $\notin$ *writingset* $\wedge$

nn2 $\notin$ *name*[*dcontents*[{dr2}]]

THEN

located_in := *located_in* $\cup$ {oi2 $\mapsto$ dr2} ||

dcontents := *dcontents* $\cup$ {dr2$\mapsto$oi2} ||

oid := *oid* $\cup$ {oi2} ||

name := *name* $\cup$ {oi2$\mapsto$nn2} ||

owner(oi2) := *owner*(oi1) ||

files := *files* $\cup$ {oi2} ||

fcontent(oi2):=*fcontent*(oi1)

END

END;

mkdir(oi,dr)=
PRE oi∈*OID* ∧ dr∈*NME*
THEN
SELECT
oi∉*oid* ∧
dr∉*name*[*dcontents*[{*curdir*}]]
THEN
located_in (*oi*) := *curdir* ||
dcontents := *dcontents* ∪{*curdir*↦oi} ||
oid := *oid* ∪{oi} ||
name(*oi*) := dr ||
owner(*oi*):= *c_usr* ||
directories:= *directories*∪{oi}
END
END;

cdto(oi,drn)=
PRE oi∈*OID* ∧ drn∈*NME*
THEN
SELECT
name(*oi*)=drn ∧
oi≠*root* ∧
oi∈*dcontents*[{*curdir*}] ∧
oi∈*directories* ∧
owner(*oi*)=*c_usr*
THEN
readingset := *readingset* ∪ {oi} ||
curdir:=oi
END
END;

cdbck =
PRE *curdir*≠*root*
THEN

readingset := *readingset* - {*curdir*} ||
curdir:= *located_in*(*curdir*)
END;

delObj(oi,nme)=
PRE oi∈*OID* ∧ nme∈*NME*
THEN
SELECT
oi ∈*dcontents*[{*curdir*}]
∧ *name*(*oi*)= nme
THEN
ANY descendents,all
WHERE
descendents⊆*oid* ∧
all⊆*oid* ∧
descendents = closure1(*dcontents*)[{oi}] ∧
all = descendents∪{oi} ∧
all ∩ (*writingset*∪*readingset*) = ∅ ∧
owner[all]={*c_usr*}
THEN
located_in := all⩤*located_in* ∧
dcontents := *dcontents*|>>all ∧
name := all⩤*name* ∧
owner := all⩤*owner* ∧
fcontent := all⩤*fcontent* ∧
oid:= *oid*-all ∧
files := *files* - all ∧
directories := *directories* - all ∧
END
END
END;

lst ← dir ≙
lst := *name*[*dcontents*[{*curdir*}]];

cnt ← *read*(oi,nn)$\widehat{=}$
  PRE nn∈*NME* ∧ oi∈*files*
  THEN
    SELECT
      oi∈*dcontents*[{*curdir*}] ∧
      name(*oi*)=nn ∧
      oi∉*writingset* ∧
      owner(*oi*)=*c_usr*
    THEN
      readingset := *readingset* ∪ {oi} ||
      cnt:=*fcontent*(*oi*)
    END
  END;

endread(oi,nn)$\widehat{=}$
  SELECT
    oi∈*oid* ∧
    name(*oi*)=nn ∧
    oi∈*readingset* ∧
    oi∈*files*
  THEN
    readingset := *readingset* - {oi}
  END;

write(oi,nn,ct)$\widehat{=}$
  PRE ct∈*CONTENT* ∧ nn∈*NME* ∧ oi∈*OID*
  THEN
    SELECT
      oi∈*files* ∧
      oi∈*dcontents*[{*curdir*}] ∧
      oi∉*writingset*∪*readingset* ∧
      owner(*oi*)=*c_usr* ∧
      name(*oi*)=nn ∧
    THEN

writingset:= *writingset*∪{oi} ||
fcontent(*oi*):=ct
END
END;

endwrite(oi,nn)≙
SELECT *name*(*oi*) = nn ∧ oi∈*writingset*
THEN
writingset:=*writingset* - {oi}
END
END

C.8 Windows NT File System with Error

The B machine is the same as *FileSystem*, except for an error that now occurs in the *read* operation, which leads to a state violating the invariant (cannot guarantee *writingset* ∩ *readingset* = ∅).

MACHINE *FileSystem*
⋮
write(oi,nn,ct)≙
PRE ct∈*CONTENT* ∧ nn∈*NME* /* MISSING ∧ oi∈*OID**/
THEN
SELECT
oi∈*files* ∧
oi∈*dcontents*[{*curdir*}] ∧
oi∉*writingset*∪*readingset* ∧
owner(*oi*)=*c_usr* ∧
name(*oi*)=nn ∧
THEN
writingset:= *writingset*∪{oi} ||
fcontent(*oi*):=ct
END

END;

⋮

END

C.9 Dining Philosophers Machine

MACHINE

DiningPhilosophers

SETS

Phil;

Forks

CONSTANTS

lFork,*rFork*

PROPERTIES

$lFork \in Phil \rightarrowtail\!\!\!\!\!\rightarrow Forks \wedge$

$rFork \in Phil \rightarrowtail\!\!\!\!\!\rightarrow Forks \wedge$

$\mathrm{card}(Forks) = \mathrm{card}(Phil) \wedge$

$\forall(pp).(pp \in Phil \Rightarrow lFork(pp) \neq rFork(pp))$

VARIABLES

taken

INVARIANT

$taken \in Forks \nrightarrow Phil \wedge$

$\forall(xx).(xx \in \mathrm{dom}(taken) \Rightarrow (lFork(taken(xx)) = xx \text{ or } rFork(taken(xx)) = xx))$

INITIALISATION $taken := \varnothing$

OPERATIONS

TakeLeftFork(p,f) = **PRE** $p \in Phil \wedge f \in Forks \wedge f \notin \mathrm{dom}(taken) \wedge lFork(p) = f$

THEN

$taken(f) := p$

END;

TakeRightFork(p,f) = **PRE** $p \in Phil \wedge f \in Forks \wedge f \notin \mathrm{dom}(taken) \wedge rFork(p) = f$

THEN
taken(f) := p
END;

DropFork(p,f) = **PRE** $p \in Phil \wedge f \in Forks \wedge f \in \mathrm{dom}(taken) \wedge taken(f)=p$
THEN
taken := $\{f\} \lhd taken$
END
END

C.10 Peterson's Mutual Exclusion

/ Peterson's algorithm (mutual exclusion for n-processes) */*
/ References in */*
/ Peterson, G.L., Myths about the mutual exclusion problem, */*
/ Information processing letters, Vol 12, No 3, 1981. */*

MACHINE *Petersons*

SETS
PID ;
label_t = {*L0*,*L1*,*L2*,*L3*,*L4*}

CONSTANTS
N

PROPERTIES
$N \in \mathbb{N}_1 \wedge N>1 \wedge card(PID)=N$

VARIABLES
P,
Q,
turn,
localj

INVARIANT

P$\in$($PID \rightarrow label\_t$) $\wedge$
Q$\in$($PID \rightarrow$ (0..N)) $\wedge$
$turn \in ((1..N) \nrightarrow PID) \wedge$
$localj \in (PID \nrightarrow (0..N)) \wedge$
not($\exists$(i1,i2).(i1$\in PID \wedge$ i2$\in PID \wedge$ (not (i1=i2)) $\wedge$ P(i1)=$L4 \wedge$ P(i2)=$L4$))

INITIALISATION

P:= {i,L|i$\in PID \wedge$L$\in label_t \wedge$L=$L0$}||
Q:= {i,v|i$\in PID \wedge$v$\in$NAT$\wedge$v=0} ||
$localj := \varnothing$ ||
$turn := \varnothing$

OPERATIONS

Inc_j_and_while(i) =
PRE i$\in PID \wedge$ P(i)=$L0$
THEN
localj(i):=1 ||
P(i):=$L1$
END;

assign_Qi_j(i) =
PRE i$\in PID \wedge$ P(i)=$L1$
THEN
Q(i):=*localj*(i) ||
P(i):=$L2$
END;

assign_TURN_j(i) =
PRE i$\in PID \wedge$ P(i) = $L2$
THEN
turn(*localj*(i)) := i ||
P(i) := $L3$
END;

wait_until(i) =
PRE (i$\in PID \wedge$ P(i) = $L3$)

```
      THEN
        IF (∀(k).(k∈PID-{i}⇒ Q(k)<localj(i))) or
          not(turn(localj(i)) = i )
        THEN
          IF localj(i)<N-1 THEN
            localj(i) := localj(i) + 1|| P(i) := L1
          ELSE
            localj(i) := localj(i) + 1|| P(i) := L4
          END
        END
      END;
    critical_and_assign_Qi_0(i) =
      PRE i∈PID ∧ P(i)= L4
      THEN
        Q(i) := 1||
        P(i) := L0
      END
END
```

C.11 Petersons Mutual Exclusion with Error

The B machine is the same as *Petersons*, except for an error that now occurs in the *read* operation, which leads to a state violating the invariant (more than one process can access the critical region at one time).

```
MACHINE
  Petersons_err
  ⋮
  wait_until(i) =
    PRE (i∈PID ∧ P(i) = L3)
    THEN
      /* The error is due to the use of, ≤, instead of, <. */
      IF (∀(k).(k∈PID-{i}⇒ Q(k)≤localj(i))) or
```

```
        not(turn(localj(i)) = i )
      THEN
        IF localj(i)<N-1 THEN
          localj(i) := localj(i) + 1|| P(i) := L1
        ELSE
          localj(i) := localj(i) + 1|| P(i) := L4
        END
      END
    END;
  ⋮
END
```

C.12 Hotel Key Card System

The machine applied to symmetry reduction during experimentation is *HotelKeys*, which refines *Hotel*. Therefore, we first present the abstract specification.

MACHINE *Hotel*

/ by Michael Butler {mjb@ecs.soton.ac.uk} */*
/ Abstract model of hotel room key allocation (loosely) based on */*
/ description from Daniel Jackson. */*

SETS
GUEST;
ROOM

/ The enumeration for these types is not necessary, */*
/ but convenient for animation in ProB. */*

VARIABLES *alloc*

INVARIANT $alloc \in ROOM \nrightarrow GUEST$ */* partial function */*

INITIALISATION $alloc := \varnothing$

OPERATIONS

CheckIn(g,r) $\widehat{=}$

PRE

g$\in$*GUEST* $\wedge$ r$\in$*ROOM*

THEN

SELECT /* *Guard*$\in$ */

r $\notin$ *dom*(*alloc*)

THEN

alloc(r) := g

END

END ;

/* *Entering will be separated into 2 cases in the refinement,* */

/* *but these 2 cases look exactly the same at the abstract level.* */

Enter1(g,r) =

PRE g$\in$*GUEST* $\wedge$ r$\in$*ROOM* **THEN**

SELECT

r$\mapsto$g $\in$ *alloc*

THEN

skip

END

END ;

Enter2(g,r) =

PRE g$\in$*GUEST* $\wedge$ r$\in$*ROOM* **THEN**

SELECT

r$\mapsto$g $\in$ *alloc*

THEN

skip

END

END ;

CheckOut(g,r) $\widehat{=}$

PRE g$\in$*GUEST* $\wedge$ r$\in$*ROOM* **THEN**

SELECT
r↦g ∈ *alloc*
THEN
alloc := {r} ⩤ *alloc /* domain subtraction */*
END
END
END

REFINEMENT *HotelKeys*

REFINES *Hotel*

/ by Michael Butler {mjb@ecs.soton.ac.uk} */*
/ Abstract model of hotel room key allocation (loosely) based on */*
/ description from Daniel Jackson. */*

SETS
KEY;
CARD

VARIABLES
alloc, key, *cArd*, ckey1, *ckey2*, *lock*, *prev*, *guest*

INVARIANT
key ∈ ℙ(*KEY*) ∧ */* set of issued keys */*

cArd ∈ ℙ(*CARD*) ∧ */* set of issued cards ('card' is a B keyword) */*

ckey1 ∈ *cArd* ⤔ key ∧ */* Each card has two keys */*
ckey2 ∈ *cArd* ⤔ key ∧ */* ⤔ represents injective functions */*
ran(*ckey1*) ∩ ran(*ckey2*) = ∅ ∧ */* disjoint ranges */*

guest ∈ *cArd* → *GUEST* ∧ */* each card is allocated to a guest */*

lock ∈ *ROOM* ⤔ key ∧ */* current key associated with a room lock */*
prev ∈ *ROOM* ⤔ key */* previous key allocated to a room lock */*

INITIALISATION

ANY ks, f **WHERE**
ks $\in \mathbb{P}(KEY) \wedge$
f $\in ROOM \rightarrowtail$ ks
THEN
key := ks ||
lock := f ||
prev := f ||
cArd, ckey1, *ckey2*, *guest*, *alloc* := $\varnothing$, $\varnothing$, $\varnothing$, $\varnothing$, $\varnothing$
END

OPERATIONS

CheckIn(g,r) $\widehat{=}$
PRE g$\in GUEST \wedge$ r$\in ROOM$ **THEN**
ANY c, k **WHERE**
r $\in ROOM \wedge$ r$\notin dom(alloc) \wedge$
c $\in CARD \wedge$ c $\notin$ *cArd* $\wedge$
k $\in KEY \wedge$ k $\notin$ key
THEN
ckey1(c) := *prev*(r) ||
ckey2(c) := k ||
guest(c) := g ||
prev(r) := k ||
key := key $\cup$ {k} ||
cArd := *cArd* $\cup$ {c} ||
alloc(r) := g
END
END ;

/* *The 2 cases for Enter are specified as separate operations.*
/* *It could be done as a single operation, but I find this*
/* *style more convenient for proof.*

Enter1(g,r) =
PRE g∈*GUEST* ∧ r∈*ROOM* **THEN**
ANY c, k **WHERE**
c∈*CARD* ∧ k∈*KEY* ∧
c ↦ g ∈ *guest* ∧
ckey1(c) = *lock*(r)
THEN
lock(r) := *ckey2*(c)
END
END ;

Enter2(g,r) =
PRE g∈*GUEST* ∧ r∈*ROOM* **THEN**
ANY c, k **WHERE**
c∈*CARD* ∧ k∈*KEY* ∧
c ↦ g ∈ *guest* ∧
ckey2(c) = *lock*(r)
THEN
skip
END
END ;

CheckOut(g,r) ≙
PRE g∈*GUEST* ∧ r∈*ROOM* **THEN**
ANY c
WHERE
c∈*CARD* ∧
c ↦ g ∈ *guest* ∧
ckey2(c) = *prev*(r) ∧
r ↦ g ∈ *alloc*
THEN
alloc := {r} ⩤ *alloc* ||
cArd := *cArd* - {c} ||

$guest := \{c\} \lhd guest \;||$
ckey1 $:= \{c\} \lhd$ ckey1 $||$
$ckey2 := \{c\} \lhd ckey2$
END
END
END

Appendix D

Machines Used in Correctness of Algorithms

This appendix presents full versions of the B machines used in Chapter 6.

D.1 An Abstract Specification for Model Checking

MACHINE

mc0

SETS

S;

$ANSWER = \{Pass, Fail\}$

CONSTANTS

i, /* *special initial state* */

tr, /* *transition relation* */

inv, /* *states satisfying invariant* */

reach /* *reachable states* */

PROPERTIES

$tr \in S \leftrightarrow \mathrm{S} \wedge$

$inv \in \mathbb{P}(S) \wedge$

$i \in inv \wedge$

$i \notin \text{ran}(tr) \wedge$

/ representation of the reachable states */*

$reach \in \mathbb{P}(S) \wedge$

$i \in reach \wedge$

/ reach is a fix point */*

$tr[reach] \subseteq reach \wedge$

/ reach is the smallest fix point of reachable states */*

$\forall(r).(r \in \mathbb{P}(S) \wedge$

$i \in r \wedge$

$tr[r] \subseteq r \Rightarrow$

$reach \subseteq r)$

OPERATIONS

$add\_inv \mathrel{\widehat{=}}$

skip;

$add\_err \mathrel{\widehat{=}}$

skip;

$remove \mathrel{\widehat{=}}$

skip;

$ok \leftarrow pass \mathrel{\widehat{=}}$

WHEN $reach \subseteq inv$

THEN $ok := Pass$

END;

$ok \leftarrow fail \mathrel{\widehat{=}}$

WHEN $reach \not\subseteq inv$

THEN $ok := Fail$

END

END

D.2 Refinement Level 1

REFINEMENT

mc1

REFINES

mc0

VARIABLES

rac, /* *reached and checked* */

err

INVARIANT

$rac \subseteq reach \wedge$

$rac \subseteq inv \wedge$

$i \in rac \wedge$

$err \subseteq reach - inv$

INITIALISATION

$rac := \{i\} \parallel$

$err := \varnothing$

OPERATIONS

$add\_inv \mathrel{\widehat{=}}$

ANY *ss* **WHERE**

$ss \subseteq reach\text{-}rac \wedge$

$ss \subseteq inv \wedge$

$ss \neq \varnothing$

THEN

$rac := rac \cup ss$

END;

$add\_err \mathrel{\widehat{=}}$

ANY *ss* **WHERE**

$ss \subseteq reach\text{-}rac \wedge$

$ss \neq \varnothing \wedge$

$ss \cap inv = \varnothing$

THEN

$err := err \cup ss$

END;

$remove \mathrel{\widehat{=}}$

ANY *ss* **WHERE**

$ss \subseteq rac$

THEN
skip
END;

ok ← *pass* ≙
WHEN
reach ⊆ *rac*
THEN
ok := *Pass*
END;

ok ← *fail* ≙
WHEN *reach* ⊈ *inv*
THEN
ok := *Fail*
END
END

D.3 Refinement for Standard Model Checking

REFINEMENT
mc2
REFINES
mc1
VARIABLES
unex, /* *reached not checked* */
rac, /* *reached and checked* */
err /* *reached errors* */

INVARIANT
$\{i\} \subseteq rac \wedge$
$tr[rac\text{-}unex] \subseteq rac \cup err$

ASSERTIONS
/* *operation enabledness preservation* */
/* *add_inv* */
$\exists(s1,s2).(s1 \in unex \wedge$

$s2 \in ran(\mathrm{tr}) \wedge$
$s2 \in inv \wedge$
$s1 \mapsto s2 \in \mathrm{tr} \wedge$
$s2 \notin rac \wedge$
$err = \varnothing) \vee$

/* *add_err* */
$\exists(s1,s2).(s1 \in unex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$s2 \notin rac \wedge$
$err = \varnothing) \vee$

/* *remove* */
$\exists(s1).(s1 \in unex \wedge$
$tr[\{s1\}] \subseteq rac \wedge$
$err = \varnothing) \vee$

/* *pass* */
$(unex = \varnothing \wedge$
$err = \varnothing) \vee$

/* *fail* */
$(err \neq \varnothing)$

INITIALISATION

$unex := \{i\} \;||$
$rac := \{i\} \;||$
$err := \varnothing$

OPERATIONS

$add\_inv \mathrel{\widehat{=}}$
ANY *s1,s2* **WHERE**
$s1 \in unex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \in inv \wedge$

$s1 \mapsto s2 \in tr \wedge$
$s2 \notin rac \wedge$
$err = \varnothing$
THEN
$unex := unex \cup \{s2\} \;||$
$rac := rac \cup \{s2\}$
END;

$add\_err =$
ANY *s1,s2* **WHERE**
$s1 \in unex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$s2 \notin rac \wedge$
$err = \varnothing$
THEN
$err := err \cup \{s2\}$
END;

$remove =$
ANY *s1* **WHERE**
$s1 \in unex \wedge$
$tr[\{s1\}] \subseteq rac \wedge$ /* *all successors of s1 have been checked* */
$err = \varnothing$
THEN
$unex := unex$ - $\{s1\}$
END;

$ok \leftarrow pass \mathrel{\widehat{=}}$
WHEN
$unex = \varnothing \wedge$
$err = \varnothing$
THEN
$ok := Pass$

END;

ok ← *fail* ≙
WHEN *err* ≠ ∅
THEN
ok := *Fail*
END
END

D.4 Refinements for Symmetry Reduced Model Checking

REFINEMENT
rmc1

REFINES
mc1

CONSTANTS
aut, /* *automorphisms on transition relation* */
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 the transition relation* */
$\forall(p,s1,s2).(p \in aut \wedge s1 \in S \wedge s2 \in S \Rightarrow$
$(s1 \mapsto s2 \in tr) \Leftrightarrow (p(s1) \mapsto p(s2) \in tr)) \wedge$

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

$rep \in S \rightarrow S \wedge$

/* *symmetric states have same representatives* */

$\forall(p,s1,s2).(p \in aut \wedge s1 \mapsto s2 \in p \Rightarrow$
$rep(s1) = rep(s2)) \wedge$

/ representatives implies automorphism */*
$\forall(s1,s2).(s1 \mapsto s2 \in rep \Rightarrow$
$\exists(p).(p \in \text{aut} \wedge s1 \mapsto s2 \in p)) \wedge$

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

ASSERTIONS

/ representatives preserve the invariant */*
$\forall(s1,s2).(s1 \in S \wedge$
$s2 \in S \wedge$
$s1 \mapsto s2 \in rep \Rightarrow$
$((s1 \in inv) \Leftrightarrow (s2 \in inv))) \wedge$

$rep(i) = i \wedge$
$rep^{-1}[\{i\}] = \{i\} \wedge$

$\forall(s1,s2).(s1 \mapsto s2 \in tr \Rightarrow$
$\exists(ss2).(rep(s1) \mapsto ss2 \in tr \wedge rep(s2) = rep(ss2))) \wedge$

/ a state is reachable iff its representative is reachable */*
$\forall(s).(s \in S \Rightarrow$
$((s \in reach) \Leftrightarrow (rep(s) \in reach)))$

VARIABLES

/ vars of original spec */*
$rac,unex,err,$
/ reduced reached and checked */*
$rrac,runex,rerr$

INVARIANT

$unex \subseteq rac$

$rrac \subseteq ran(rep) \wedge$
$rrac \subseteq rac \wedge$

$runex \subseteq rrac \wedge$

$rerr \subseteq err \wedge$

$rep^{-1}[rrac] = rac \wedge$

$rep^{-1}[runex] = unex \wedge$

$rep^{-1}[rerr] = err \wedge$

$tr[rac\text{-}unex] \subseteq rac \cup err$

INITIALISATION

$rac := \{i\} \parallel rrac := \{i\} \parallel$

$unex := \{i\} \parallel runex := \{i\} \parallel$

$err := \varnothing \parallel rerr := \varnothing$

OPERATIONS

$add\_inv \mathrel{\widehat{=}}$

ANY *s1,s2* **WHERE**

$s1 \in runex \wedge$

$s2 \in ran(tr) \wedge$

$s2 \in inv \wedge$

$s1 \mapsto s2 \in tr \wedge$

$rep(s2) \notin rrac \wedge$

$rerr = \varnothing$

THEN

$runex := runex \cup \{rep(s2)\} \parallel$

$unex := unex \cup rep^{-1}[\{rep(s2)\}] \parallel$

$rrac := rrac \cup \{rep(s2)\} \parallel$

$rac := rac \cup rep^{-1}[\{rep(s2)\}]$

END;

$add\_err \mathrel{\widehat{=}}$

ANY *s1,s2* **WHERE**

$s1 \in runex \wedge$

$s2 \in ran(tr) \wedge$

$s2 \notin inv \wedge$

$s1 \mapsto s2 \in tr \wedge$

$rep(s2) \notin rrac \wedge$
$rerr = \varnothing$
THEN
$rerr := rerr \cup \{rep(s2)\} \;||$
$err := err \cup rep^{-1}[\{rep(s2)\}]$
END;

remove =
ANY *s1* **WHERE**
$s1 \in runex \wedge$
$rep[tr[\{s1\}]] \subseteq rrac \wedge$ /* *all successors of s1 have been checked* */
$rerr = \varnothing$
THEN
$runex := runex - \{s1\} \;||$
$unex := unex - rep^{-1}[\{s1\}]$
END;

$ok \leftarrow pass \mathrel{\widehat{=}}$
WHEN
$reach \subseteq rep^{-1}[rrac]$
THEN
$ok := Pass$
END;

$ok \leftarrow fail \mathrel{\widehat{=}}$
WHEN
$reach \not\subseteq inv$
THEN
$ok := Fail$
END
END

D.4.1 Level 2

REFINEMENT
rmc2

REFINES
rmc1

VARIABLES

/ reduced reached and checked */*

$rrac, runex, rerr$

ASSERTIONS

/ operation enabledness preservation */*

/ add_inv */*

$\exists(s1,s2).(s1 \in runex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \in inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$rep(s2) \notin rrac \wedge$
$rerr = \varnothing) \vee$

/ add_err */*

$\exists(s1,s2).(s1 \in runex \wedge$
$s2 \in ran(tr) \wedge$
$s2 \notin inv \wedge$
$s1 \mapsto s2 \in tr \wedge$
$rep(s2) \notin rrac \wedge$
$rerr = \varnothing) \vee$

/ remove */*

$\exists(s1).(s1 \in runex \wedge$
$rep[tr[\{s1\}]] \subseteq rrac \wedge$
$rerr = \varnothing) \vee$

/ pass */*

$(rerr = \varnothing \wedge$
$runex = \varnothing) \vee$

/ fail */*

$(rerr \neq \varnothing)$

INVARIANT

$i \in rrac \wedge$

$rrac \subseteq ran(rep) \wedge$

$rrac \subseteq rac \wedge$

$runex \subseteq rrac \wedge$

$rerr \subseteq err$

INITIALISATION

$rrac := \{i\} \parallel$

$runex := \{i\} \parallel$

$rerr := \varnothing$

OPERATIONS

$add\_inv \mathrel{\widehat{=}}$

ANY *s1,s2* **WHERE**

$s1 \in runex \wedge$

$s2 \in ran(tr) \wedge$

$s2 \in inv \wedge$

$s1 \mapsto s2 \in tr \wedge$

$rep(s2) \notin rrac \wedge$

$rerr = \varnothing$

THEN

$runex := runex \cup \{rep(s2)\} \parallel$

$rrac := rrac \cup \{rep(s2)\}$

END;

$add\_err \mathrel{\widehat{=}}$

ANY *s1,s2* **WHERE**

$s1 \in runex \wedge$

$s2 \in ran(tr) \wedge$

$s2 \notin inv \wedge$

$s1 \mapsto s2 \in tr \wedge$

$rep(s2) \notin rrac \wedge$

$rerr = \varnothing$

THEN

$rerr := rerr \cup \{rep(s2)\}$

END;

remove =
ANY *s1* **WHERE**
s1 $\in$ *runex* $\wedge$
rep[*tr*[{*s1*}]] $\subseteq$ *rrac* $\wedge$ /* *all successors of s1 have been checked* */
rerr = $\varnothing$
THEN
runex := *runex* - {*s1*}
END;

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

ok $\leftarrow$ *fail* $\widehat{=}$
WHEN
rerr $\neq$ $\varnothing$
THEN
ok := *Fail*
END
END

Bibliography

Jean-Raymond Abrial. *The B Book: Assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOLs*, pages 1–24, 2003.

Jean-Raymond Abrial and L. Mussat. *Event B On-line Reference Manual.* Clearsy, Parc de la Duranne, 320 avenue Archimède Les Pléiades III - Bât A, 13857 Aix en Provence Cedex 3, France, 2001. URL `http://www.atelierb.societe.com/ressources/evt2b/eventb_reference_manual.pdf`.

Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 Tree Identify Protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for event-b. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006. ISBN 3-540-47460-9.

William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. *SIGOPS Oper. Syst. Rev.*, 34(2):22, 2000. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/346152.346192.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

B-Toolkit, On-line manual. B-Core (UK) Limited, Oxon, UK, 1999. URL `http://www.b-core.com/ONLINEDOC/Contents.html`.

László Babai and Eugene M. Luks. Canonical labeling of graphs. In *STOC*, pages 171–183. ACM, 1983.

Ralph-Johan Back. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1): 49–68, 1981.

Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000. ISBN 3-540-41030-9.

Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 93–106. Springer, 2002. ISBN 3-540-43997-8.

Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. Rulebase: An industry-oriented formal verification tool. In *DAC*, pages 655–660, 1996.

Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999. ISBN 3-540-66587-0.

Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Inf.*, 20:207–226, 1983.

Shoham Ben-David, Orna Grumberg, Tamir Heyman, and Assaf Schuster. Scalable distributed on-the-fly symbolic model checking. *STTT*, 4(4):496–504, 2003.

Eike Best and Maciej Koutny. A refined view of the box algebra. In Giorgio De Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1995. ISBN 3-540-60029-9.

Ritwik Bhattacharya, Steven M. German, and Ganesh Gopalakrishnan. Symbolic partial order reduction for rule based transition systems. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 332–335. Springer, 2005. ISBN 3-540-29105-9.

Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric SPIN. *STTT International Journal on Software Tools for Technology Transfer*, 4(1):92–106, 2002.

Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model checking for real-time systems. In *IEEE Real-Time Systems Symposium*, pages 25–. IEEE Computer Society, 1997.

Don Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-20163-446-5.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

Dominique Cansell and Dominique Méry. Tutorial on the event-based B method, Formal Techniques for Networked and Distributed systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.

Dominique Cansell, Ganesh Gopalakrishnan, Michael D. Jones, Dominique Méry, and Airy Weinzoepflen. Incremental proof of the producer/consumer property for the PCI Protocol. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2002. ISBN 3-540-43166-7.

Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems.* Discrete Event Dynamic Systems. Springer, 1^{st} edition, 1999.

Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988. ISBN 0-201-05866-9.

Edmund M. Clarke and Ernest A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. ISBN 3-540-11212-X.

Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [1993], pages 450–462. ISBN 3-540-56922-7.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking.* The MIT Press, 1999.

Derek G. Corneil and David G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM J. Comput.*, 9(2):281–297, 1980.

Costas Courcoubetis, editor. *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, 1993. Springer. ISBN 3-540-56922-7.

James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.

Kriangsak Damchoom. Dependable Systems and Software Engineering, School of Electronics and Computer Science, University of Southampton, UK, November 2006.

Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 530–534. ACM, 2004. ISBN 1-58113-828-8.

Greg Dennis. Tsafe: Building a trusted computing base for air traffic control software. Master's thesis, Department of Electrical Engineering and Computer Science, 32 Vassar Street, 32-G706, Cambridge, MA 02139, January 2003.

Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1: 115–138, 1971.

David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525. IEEE Computer Society, 1992. ISBN 0-8186-3110-4.

Alastair F. Donaldson, Alice Miller, and Muffy Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci.*, 128(6):161–177, 2005.

DOT. Graphviz, open source graph drawing software, AT&T Labs Research. URL `http://www.research.att.com/sw/tools/graphviz/`.

Ioannis Dravapoulos, Nikos Pronios, and Spyros Denazis. The magic WAND. Wireless ATM MAC, September 1997. Deliverable 3D2.

Nicolas Dulac, Thomas Viguier, Nancy Leverson, and Margaret-Anne Storey. On the use of Visualization in Formal Requirements Specification. In *IEEE Joint International Conference on Requirements Engineering*, pages 71–81, Essen, Germany, September 2002. URL `http://sunnyday.mit.edu/papers/RE02_visualization.pdf`.

Ernest A. Emerson and Aravinda P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, 1997.

Ernest A. Emerson and Aravinda P. Sistla. Symmetry and model checking. In Courcoubetis [1993], pages 463–478. ISBN 3-540-56922-7.

Ernest A. Emerson and Aravinda P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

Ernest A. Emerson and Thomas Wahl. Dynamic symmetry reduction. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2005. ISBN 3-540-25333-5.

Sarah Flannery and David Flannery. *In Code: A Mathematical Journey.* Algonquin Books of Chapel Hill, Chapel Hill, NC, USA, 2001. ISBN 1-56512-377-8.

Pasquale Foggia, Carlo Sansone, and Mario Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. ISBN 3-540-60761-7.

Patrice Godefroid. Model checking for programming languages using Verisoft. In *POPL*, pages 174–186, 1997.

Evguenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002. ISBN 0-7695-1471-5.

Jonathan Gross and Jay Yellen. *Graph Theory and its Applications*. Discrete Mathematics and its Applications. CRC Press, Boca Raton, Fla., USA, 1999.

Elsa L. Gunter and Doron Peled. Unit checking: Symbolic model checking for a unit of code. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 548–567. Springer, 2003. ISBN 3-540-21002-4.

Viktor Gyuris and Aravinda P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design: An International Journal*, 15 (3):217–238, November 1999.

Frank Ham, Huub van de Wetering, and Jack van Wijk. Visualization of State Transition Graphs. In *IEEE Symposium on Information Visualization*, pages 59–63, San Diego, CA, USA, October 2001. URL `http://www.win.tue.nl/~fvham/fsm/Downloads/FSM2002.pdf`.

Vicky Hartonas-Garmhausen, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Probverus: Probabilistic symbolic model checking. In Joost-Pieter Katoen, editor, *ARTS*, volume 1601 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 1999. ISBN 3-540-66010-0.

Jifeng He, Charles A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN -540-16442-1.

Ivan Herman, Guy Melanon, and Scott Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/2945.841119.

Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In R. Langerak, editor, *Proceedings of the 3th International SPIN Workshop*, Twente University, The Netherlands, 1997a. URL `citeseer.ist.psu.edu/holzmann97state.html`.

Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.

Gerard J. Holzmann. The engineering of a model checker: The GNU i-protocol case study revisited. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *SPIN*, volume 1680 of *Lecture Notes in Computer Science*, pages 232–244. Springer, 1999. ISBN 3-540-66499-8.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279, 1997b.

Gerard J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-539925-4.

Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In Dieter Hogrefe and Stefan Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994. ISBN 0-412-64450-9.

Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. Towards reachability trees for high-level petri nets. In Grzegorz Rozenberg, Hartmann J. Genrich, and Gérard Roucairol, editors, *European Workshop on Applications and Theory in Petri Nets*, volume 188 of *Lecture Notes in Computer Science*, pages 215–233. Springer, 1984. ISBN 3-540-15204-0.

Lucian Ilie and Sheng Yu. Algorithms for computing small NFAs. In Krzysztof Diks and Wojciech Rytter, editors, *MFCS*, volume 2420 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2002. ISBN 3-540-44040-2.

Chung-Wah Norris Ip. *State Reduction Methods For Automatic Formal Methods.* PhD thesis, Department of Computer Science, Stanford University, California, 1996.

Chung-Wah Norris Ip and David L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *CHDL*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993. ISBN 0-444-81641-0.

Daniel Jackson. The Alloy Analyser, 2006a. URL `http://alloy.mit.edu/`.

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006b.

Daniel Jackson and Michael Jackson. Separating concerns in requirements analysis: An example. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *RODIN Book*, volume 4157 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006. ISBN 3-540-48265-2.

Somesh Jha. *Symmetry and induction in model checking.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa, 1996.

Sebastian John. Minimal Unambiguous ε-NFA. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *CIAA*, volume 3317 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 2004. ISBN 3-540-24318-6.

Vineet Kahlon, Aarti Gupta, and Nishant Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 286–299. Springer, 2006. ISBN 3-540-37406-X.

Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7): 371–384, 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360248.360251.

Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2003. ISBN 3-540-20851-8.

Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 1973.

William Kocay. *On Writing Isomorphism Programs*, chapter Chapter 6, pages 135–175. Computational and Constructive Design Theory. Kluwer, 1996.

Donald. L. Kreher. *Combinatorial Algorithms: Generation, Enumeration and Search*. Discrete Mathematics and its Applications. CRC Press, 1998.

Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002. ISBN 3-540-43928-5.

Michael Leuchel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. Technical report, Electronics and Computer Science, University of Southampton, 2006. URL `http://eprints.ecs.soton.ac.uk/12886/`.

Michael Leuschel and Michael J. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003. ISBN 3-540-40828-2.

Michael Leuschel and Michael J. Butler. Automatic refinement checking for B. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2005. ISBN 3-540-29797-9.

Michael Leuschel and Thierry Massart. Efficient approximate verification of B via symmetry markers. In *International Symmetry Conference*, page (to appear). Springer, January 2007.

Michael Leuschel and Edd Turner. Visualising larger state spaces in ProB. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 6–23. Springer, 2005. ISBN 3-540-25559-1.

Michael Leuschel, Michael Butler, Corinna Spermann, and Edd Turner. Symmetry reduction for B by permutation flooding. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag, 2007. ISBN 3-540-68760-2.

Roger Lipsett, Carl F. Schraefer, and Cary Ussery. *VHDL: Hardward Description and Design.* Springer, 2^{nd} edition, June 1989. ISBN 1-402-07089-1.

Andreas Malcher. Minimizing finite automata is computationally hard. *Theor. Comput. Sci.*, 327(3):375–390, 2004.

Brendan D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998. ISSN 0196-6774. doi: http://dx.doi.org/10.1006/jagm.1997.0898.

Brendan D. McKay. Practical graph isomorphism. In *Numerical mathematics and computing, Proc. 10th Manitoba Conf., Congr. Numerantium 30*, pages 45–87, 1981.

Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393805.

Christophe Métayer, Jean-Raymond Abrial, and Laurent Voisin. Event B language, RODIN deliverable D7, 2005. URL `http://rodin.cs.ncl.ac.uk/`.

Albert R. Meyer and Michael J. Fischer. Economy of description by automata, grammars, and formal systems. In *FOCS*, pages 188–191. IEEE Computer Society, 1971.

Alice Miller, Alastair F. Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006.

Alice Miller, Muffy Calder, and Alastair F. Donaldson. A template-based approach for the generation of abstractable and reducible models of featured networks. *Computer Networks*, 51(2):439–455, 2007.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001. ISBN 1-58113-297-2.

Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992. ISBN 3-540-55602-8.

Doron Peled. Partial order reduction: linear and branching temporal logics and process algebras. In *POMIV '96: Proceedings of the DIMACS workshop on Partial order methods in verification*, pages 233–257, New York, NY, USA, 1997. AMS Press, Inc. ISBN 0-8218-0579-7.

Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12 (3):115–116, 1981.

Amir Pnueli. The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13: 45–60, 1981.

Denduang Pradubsuwun, Tomohiro Yoneda, and Chris J. Myers. Partial order reduction for detecting safety and timing failures of timed circuits. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2004. ISBN 3-540-23610-4.

Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

RODIN. Rigorous Open Development Environment for Complex Systems (RODIN), EU project IST 511599. URL `http://rodin.cs.ncl.ac.uk/`.

Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs - Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer, 1993. ISBN 3-540-56254-0.

Klaus Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/j.dam.2005.10.018.

SICStus. SICStus prolog user's manual, 2005. URL `http://www.sics.se/isl/sicstus/docs/3.12.3/pdf/sicstus.pdf`.

Aravinda P. Sistla, Viktor Gyuris, and Ernest A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.

Spin 4.3.0. Full Spin Distribution 4.3.0, June 2007. URL `http://spinroot.com/spin/Src/spin430.tar.gz`.

Peter H. Starke. Reachability analysis of petri nets using symmetries. volume 8, pages 293–303, Newark, NJ, USA, 1991. Gordon and Breach Science Publishers, Inc.

Atelier B, User and Reference Manuals. Steria, Aix-en-Provence, France, 1996. URL `http://www.atelierb.societe.com/index_uk.htm`.

Tcl/Tk. Tcl/Tk Developer Exchange Website, 2007. URL `http://www.tcl.tk/doc/`.

Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Springer, 5^{th} edition, June 2002. ISBN 1-402-07089-1.

Edd Turner, Michael Leuschel, Corinna Spermann, and Michael Butler. Symmetry reduced model checking for B. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5-8, 2007, Shanghai, China*, pages 25–34. IEEE Computer Society, 2007.

Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

Willem Visser and Howard Barringer. Memory efficient state storage in Spin. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, editors, *Proceedings of Series on Discrete Mathematics and Theoretical Computer Science, The SPIN Verification System*, volume 32, pages 239–256, Providence, Rhode Island 02940, USA, 1996. American Mathematical Society.

Pierre Wolper and Denis Leroy. Reliable hashing without collosion detection. In Courcoubetis [1993], pages 59–70. ISBN 3-540-56922-7.