On Symmetry Reduction in Model Checking via Graph Canonicalisation

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Corinna Spermann
aus Ludwigslust

Düsseldorf, November 2009

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Leuschel
Koreferent: Prof. Dr. Rothe

Tag der mündlichen Prüfung: 20.01.2010

Abstract

This thesis continues the work on symmetry reduction for model checking in B. It picks up the idea of translating states into graphs, such that symmetric states correspond exactly to isomorphic graphs. This reduces the orbit problem in symmetry reduction to the graph isomorphism problem. The graph isomorphism problem is a well-studied mathematical problem, although a polynomial time algorithm to solve it has yet to be found.

However, existing tools are capable of detecting isomorphic graphs via graph canonicalisation very efficiently. One of these tools, named NAUTY, is integrated through an interface into the ProB model checker. This new way of applying graph canonicalisation in model checking is then compared empirically with the already existing symmetry reduction approaches in ProB.

Zusammenfassung

Diese Doktorarbeit setzt die Arbeit über Symmetriereduktion für Modelchecking in B fort. Sie nimmt die Idee auf, Zustände in Graphen zu übersetzen, sodass symmetrische Zustände genau isomorphen Graphen entsprechen. Dies reduziert das Orbit Problem in Symmetriereduktion auf das Graph Isomorphie Problem. Das Graph Isomorphie Problem ist ein gut studiertes mathematisches Problem, obwohl ein polynomialer Algorithmus zur Lösung dieses Problems erst noch gefunden werden muss.

Jedoch gibt es Werkzeuge, die fähig sind isomorphe Graphen mittels Graph Normalisierung zu entdecken. Eines dieser Werkzeuge heißt NAUTY und wird über ein Interface in den ProB Model Checker integriert. Dieser neue Weg die Graph Normalisierung in Model Checking anzuwenden, wird dann mit den bereits existierenden Methoden zur Symmetriereduktion in ProB verglichen.

Acknowledgments

First of all, I would like to thank my supervisor, Michael Leuschel, for allowing me to concentrate on my thesis during a time of world-wide recession. I also want to thank him, together with Michael Butler and Edward Turner, for introducing me to the topic of symmetry reduction in model checking. Coming from a mathematical background, I know that mathematics is the art to avoid calculation, and symmetry reduction is exactly that.

Most of all, I want to thank my fiancée, Oliver, for his love, support and encouragement throughout my work. Special thanks go to Jim Davies for encouraging me to continue writing my thesis on a day when its completion looked still unachievable. Last, but certainly not least, I wish to thank my parents and brother for everything.

Contents

Chapter 0

Introduction

In today's world, we are surrounded by technology almost 24/7. We are woken up by a radio-controlled alarm clock, and while the coffee machine does its magic, other electronic gadgets make sure that our shower water has the right temperature. Our car guides us around the rush hour traffic, hopefully, and its electronically-enhanced brakes work as well as ever. Those who rather take the train to work are happy that the computer-aided signal controls ensure that only one train occupies the same space at the same point in time. Having arrived at work, the lift takes us to the right floor, and our office is already nicely pre-heated by the running PCs. In many daily situations, we more or less rely on technology, and its failure can lead to anything from mild annoyance (by having a cold coffee in the morning), to a fatal car or train accident. Although we may have missed all that, because our alarm clock failed...

Generally, we expect (and, indeed, depend on) everything working, even if we do not understand - or just forget - how all these everyday devices work. Technology has become too complex in many cases to be fully understood by a single human, anyway. Consequently, errors are easily made during its development. The software for the train signal controls may miss a safety-critical feature, or an error in the program code can bring the controls to a standstill - and at best, all trains, too. This example shows two different kinds of error that can be introduced in any complex hard- or software system, during its development. First, errors can be made during the design phase, such as a missing feature, and secondly during its implementation phase, such as a part not being properly fixed to a car, or an uncaught exception in the program code of a software application. The development of complex systems with a formal method can help to prevent these causes of errors. In the following, we will mostly refer to software systems, although formal methods have also been successfully used in hardware development. The usage of formal methods during the design of software can help structure the whole process by allowing the designer to first concentrate on the main features, called *abstract specification*, then stepwise add more details to the specification, to finally reach a *concrete specification* with all necessary design features. From that point on, formal methods are even more helpful, since further refinement, i.e. the translation from abstract data structures to

implementable ones, can be partly automated, and also proven correct, with regard to the specification. The resulting implementation can be translated automatically into program code that has been proven to meet the specification, so that errors of the second type cannot occur. We have to make several assumptions here, such as the tools for proving the correctness, the translation to and compiling of program code, all work correctly. We further assume that the underlying hardware works correctly, and is not affected by cosmic rays, electro-magnetic interference, etc.

No matter what assumptions we make about the environment, proving the correctness of a specification, also called model, is still an involved task. Even if 95% of the proofs can be done automatically by a prover, some user intervention is still needed to do the rest of the proofs, which can be very time-consuming. So, is there a way to verify a model without significant user intervention? Yes, there is, and it is called model checking, a completely different approach from mathematical proofs.

We are going to explain the basics of model checking in Chapter 1, and then concentrate on the B-method [1]. We will briefly demonstrate modelling with B, and what proving the correctness of a model means. Then we will concentrate on model checking of B models, and introduce ProB - a tool for animating and model checking B models.

The main focus of this thesis is on symmetry reduction in B. In Chapter 2, we want to provide a motivation for this topic. We will explain why symmetry reduction and the B language go well together. This thesis improves a symmetry reduction approach [56], that puts the symmetry of B-states into a direct relationship with the isomorphism of graphs. We will explain in Chapter 2 some mathematical background on symmetry, and especially how B-states can be transformed into graphs.

Knowing how B-states can be interpreted as graphs, we introduce in Chapter 3, nauty [42], a tool developed by B. D. McKay to help solving graph isomorphism and related problems. We will explain how nauty works from a mathematical point of view, and also give a brief insight into the data structures used by nauty. Our work includes a mechanism to translate graphs resulting from B-states into the notation used by nauty. We implemented an interface between nauty and ProB, to allow communication between the two tools. This makes it possible for ProB to detect symmetries, with the help of nauty. We describe in detail the functionality of the interface, then we finish Chapter 3 with an overview on related work.

In Chapter 4, we want to present some empirical results. Symmetry reduction in B with nauty improves on the previous work by E. Turner et al. [56], and so we will draw a comparison. We will also compare our approach with the work of Leuschel and Massart [40] on "Efficient approximate verification of B via symmetry markers" and the work of Leuschel et al.[39] on "Symmetry reduction for B by permutation flooding." Finally, in Chapter 5, we will draw some conclusions and give ideas on future work.

Chapter 1

Model Checking

1.1 Basics of Modelling

Model checking is one way to find errors in a software system; while the (usually too complex) software system is not checked for errors, a much simpler specification of it, is. A specification is a model of a software system that also describes the correct behaviour of the system.

The process of model checking is automated by a model checking tool, which searches each state the system can be in (also called the state space of the model), for violations of the specification. This obviously works only for finite state spaces. Since most systems have an infinite number of states, the system parameters - such as the domain of a variable, for example - are limited by the model checking tool. Within those boundaries, it can be verified that a model fulfils the specification. It is important to note that if the specification itself is not correct, or is incomplete, then nothing can be said about the behaviour of the real system.

The idea of model checking is quite intuitive: The correct behaviour of the model is checked in each and every reachable state, starting from the initial state. Before we can reason anything about a complex system, though, we first need to have the appropriate model for it. A model is a simplified description of the real system that is ideally easy enough to understand, but complex enough to explain the behaviour of the real system.

For example, we all learned in chemistry lessons to first regard atoms as balls. That is a very simple model, and easy enough to be understood by children. This model, though, cannot explain why solid salt does not conduct electricity, but why salt dissolved in water does. To understand this behaviour, we had to learn about the more complicated model of an atom that has a positively-charged nucleus, with several orbits of negatively-charged electrons. Only this model allowed us to understand how free electrons occur in salt water, so that it can conduct electricity.

In software engineering, it is very important to be able to model the state of a system. For instance, if we model software that controls the movements of a lift, then we need to describe the position of the cabin. It is necessary to know its position

to decide if the lift is still in a correct state, after the cabin moves another floor up. This example gives us another property that we need to be able to describe: The behaviour of the system in time. To move a cabin from the first to the third floor, it must first go through the second floor.

A *Kripke structure* is such a model that can describe the state and the temporal behaviour of a system. We give a formal description of a Kripke structure:

Definition 1.1 A Kripke structure K is a tuple $K = (S, S_0, R, AP, L)$, where

1. S is a finite set of states
2. S_0 is the set of initial states
3. $R \subseteq S \times S$ is a relation
4. AP is a set of atomic propositions
5. $L : S \rightarrow 2^{AP}$ is a function, that labels each state with the set of atomic propositions true in that state.

Let's have the Kripke structure $K = (S, S_0, R, AP, L)$ describing a system. A *path in* M is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that $(s_i, s_{i+1}) \in R$ for every $i \geqslant 0$. The requirements of a system are written in temporal logic formulas, such as CTL^* formulas. In CTL^*, there are two types of formulas, *state formulas* and *path formulas*. The notation

$$K, s \models f$$

means that the formula f holds in the state s of K, and the notation

$$K, \pi \models f$$

means that f holds in some state along the path π in K.

Example 1.2 Let's take as example the description of a security door, that consists of two subsequent doors d_1 and d_2, such that there is, at most, one door open at all times. In the initial state, both doors are closed. Figure 1.1 is a graphical representation of the state space.

We can translate the informal description of the security door into a Kripke structure $K = (S, S_0, R, AP, L)$ with:

$$S = \{s_0, s_1, s_2\}$$
$$S_0 = \{s_1\}$$
$$R = \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1)\}$$
$$AP = \{d1 = op, d1 = cl, d2 = op, d2 = cl\}$$
$$L(s_0) = \{op, cl\}, L(s_1) = \{cl, cl\}, L(s_2) = \{cl, op\}$$

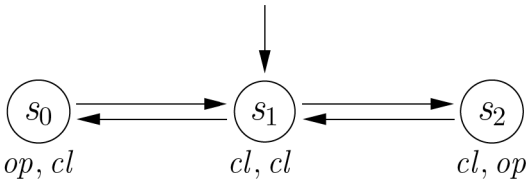


Figure 1.1: State graph

An example for a path π in K is:

$$\pi = s_1, s_0, s_1, s_2, s_1, s_0, \ldots$$

Let's consider the formula $f := `d1 = op` \wedge `d_2 = cl`$ then we have

$$K, s_0 \models f \quad \text{and}$$
$$K, \pi \models f$$

The formula f is obviously true in the state s_0, and with s_0 there is a state on the path π such that f holds.

For further details of temporal logic, especially CTL^*, see [13]. The CTL^* formulas can be automatically verified by a model checker. In most cases it is required that a model is deadlock free. Consequently, the transition relation R in the respective Kripke structure must be a total relation, that means for each state $s \in S$, there is a state $s' \in S$ such that $(s, s' \in R)$. In the following, a Kripke structure always has a total relation.

1.2 Modelling with B

The B-method is a method for systematic development of large software systems. It was invented by Jean-Raymond Abrial [1] and has been used for a variety of safety-critical systems all over the world. One example is the Meteor project, the implementation of the driverless Paris Métro Line 14. The B-method is supported by a number of tools such as Atelier-B [53], the B-toolkit [6] or ProB [36]. Atelier-B and B-Toolkit both allow the mathematical proof of the correctness of software developed with B, and an automated translation from B to programming language code (mainly C). ProB is a different kind of tool in that it provides validation through model checking rather than proofs, and also gives the user the possibility to animate his model. For a more complete list of tools supporting B, see [35].

The development of a large system is broken down by using *abstract machines* as building blocks of the specification, and the refinement of abstract machines. An abstract machine, also called B-model or just model in the following, is written in

Abstract Machine Notation, abbreviated AMN, which is a mathematic framework based on set theory and predicate logic.

Using set theory and predicate logic allows an abstract and systematic software development. The refinement aspect of the B-method allows, first to stepwise add more and more details of the real system to the specification, and then to refine the data structures within a specification such that they can easily be translated to a programming language, such as C. That means a software system, or a part of a software system, is developed throughout a chain of models, starting from a very abstract specification and finishing with a concrete model, that can be translated into code. The process of software development with B is depicted in Figure 1.2. Here we have an abstract machine *M0.mch* which is refined by the machine *M1.ref*. A number of further refinements finally leads to the machine *MI.imp* on the so-called *B0 level*. At this level, only a subset of B constructs can be used, which are directly implementable in program code.

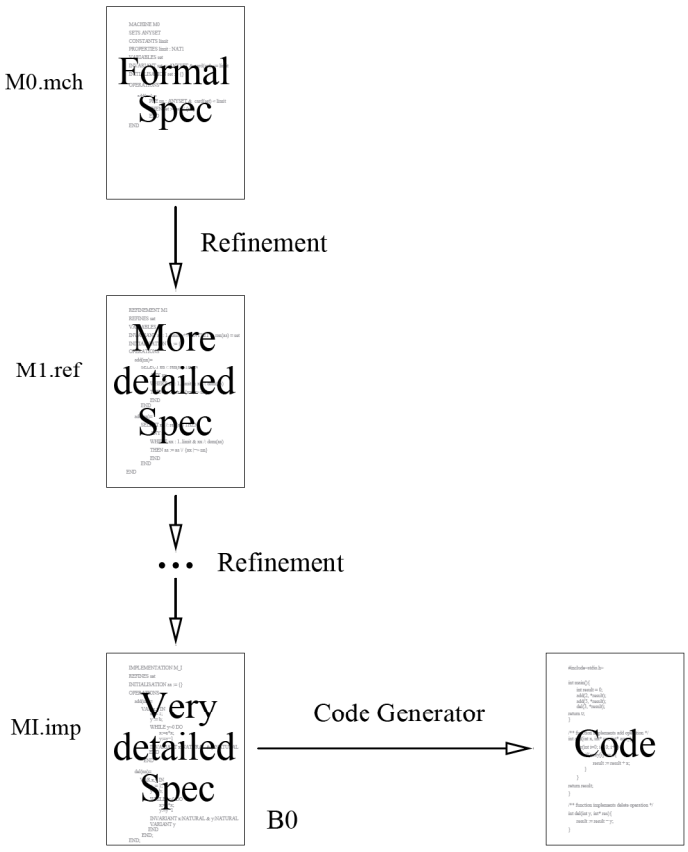


Figure 1.2: Development in B

Since B models are developed systematically using mathematical constructs, it is possible to mathematically prove that each refinement and therefore the resulting program code meets its specification.

In the following, we want to briefly explain the main parts of a B-model with a small example describing a database for a course enrollment. For a deeper introduction into modelling with B, see [49].

MACHINE *Course*

SETS
Student

CONSTANTS
max_num

PROPERTIES
$max\_num = 20$

VARIABLES
class

INVARIANT
$class \subseteq Student \wedge$
$card(class) \leq max\_num$

INITIALISATION
$class := \emptyset$

OPERATIONS
$enroll(s) \mathrel{\widehat{=}}$
PRE
$s \in Student \wedge$
$s \notin class \wedge$
$card(class) < max\_num$
THEN
$class := class \cup \{s\}$
END;

$out \leftarrow leave(s) \mathrel{\widehat{=}}$
PRE
$s \in class$
THEN
$class := class - \{s\} \;||$
$out := s$
END
END

The keywords in upper case have the following function:

- MACHINE: Introduces the name of the model, here *course*. Alternatively the keyword MODEL can be used.

- SETS: Sets define new datatypes, that can be used throughout the model. There are two types of sets: The enumerated sets and the deferred sets. Enumerated sets have their elements given explicitly. For deferred sets this is not the case, their definition is deferred to a later stage in the modelling process. In the example there is one deferred set named *Student*.

- CONSTANTS: Declares the constants used in the machine. The example has one constant, max_num.

- PROPERTIES: Determines the properties of the sets and constants. In the example we have $max\_num = 20$.

- VARIABLES: The variables define the state of a model. In this case we have just the variable *class*, to keep track of the students that take part in the course.

- INVARIANT: The invariant is a very important part of a model. It has for a model a similar function as temporal logic formulas have for a Kripke structure, in that it expresses the requirements of the modelled system. Secondly, the invariant gives the datatypes for each variable. All invariant conditions have to be met in every state of the model. In the example, we have the variable *class* as a subset of *Student*, and the condition that the size of the class cannot exceed the value 20, which is given through the constant max_num.

- INITIALISATION: Gives the initial state of the model. All variables need to be initialised here.

- OPERATIONS: Together with the initialisation, the operations describe the behaviour of the model. There are two operations in the example: One for enrolling a student, where the student is given by the input parameter s, and one for a student leaving the class. An operation can also have output parameters, such as the parameter *out* of the operation $leave(s)$. Usually, an operation has preconditions, introduced by the keyword PRE. An operation executed under these conditions should preserve the invariant[1]. Indeed, if the operation $enroll(s)$ is executed to add another student to the course under the condition that the course is not yet full ($card(class) < max\_num$), then the invariant $card(class) \leq max\_num$ is preserved. The preconditions also give the type of any input parameters of the operation, such as $s \in Student$ for the parameter of the $enroll(s)$ operation. The keyword THEN introduces the action of an operation. In the example, the operation $leave(s)$ removes one student from the class and gives that student as output. The symbol || in the operation means that both actions are executed in parallel. Each operation is finished with the keyword END. The last END denotes the end of the model.

[1] An operation can be called outside its preconditions, but in that case, there is no guarantee of correct behaviour of the model.

For a large specification, it is usually advisable to divide it up into smaller parts, and model each part separately. This way a large system becomes more manageable and also allows several people to work on it at the same time. The B-language supports this kind of development by supplying several keywords for structuring. There are four of these keywords in the B-language, that allow the import of one machine to another. To do so the line

KEYWORD *machine_name*

is inserted usually after the machine name. We want to give an overview here, for more details about structuring with B, see again [49]. Let's have in the following a machine $M1$ that is imported somehow by a machine $M2$.

- INCLUDES: This is the most frequently used structuring keyword. it means that all information of the included machine $M1$, is now part of the including machine $M2$. Sets and constants can be directly used in $M2$, and the invariant can refer to variables from $M1$. $M2$ can also call all the operations of $M1$ within its operations. In order to make an operation of $M1$ an operation of $M2$ though, an additional keyword PROMOTES is used. The machine $M2$ then contains the line

  PROMOTES *name_of_operation_of_*$M1$.

  Any operation in that line is now an operation of $M2$, just as its original operations.

- EXTENDS: Has the same effect as an INCLUDES clause with all operations promoted. So this keyword just saves a bit of typing.

- SEES: This clause allows only read access of $M2$ to $M1$. That means that $M2$ cannot change the state of $M1$ by calling its operations as for the INCLUDES clause. The only operations of $M1$ that $M2$ can call are so called query operations, which give back only a value of a variable, but do not change the state. Sets and constants can be read as before, but the invariant of $M2$ cannot have the variables of $M1$ in its predicates.

- USES: Has almost the same properties as the SEES clause. The only difference is that the invariant of $M2$ can refer to the variables of $M1$. This clause is used when the state of machine $M2$ depends on the state of machine $M1$.

1.2.1 Proving Machine Consistency

One of the main advantages of using B for modelling, is the possibility to mathematically prove the consistency of a model. Proving consistency means essentially two things: First, if a model has parameters, sets or constants with constraints on

the parameters and properties of the sets and constants, then we must show that such parameters, sets and constants exist. We also need to prove that there are values for the model's variables that fulfil the invariant. This is to show that there are no contradictions in the constraints, properties and invariant. Secondly we need to show that the initialisation establishes and the operations preserve the invariant. Usually the consistency proof of the operations is the hardest. We will give an example for such a, so called proof obligation, for an operation of our *Course* example further below.

First, we want to give a summary of the proof obligations for a model. Let p be the parameters of a model and C the set of constraints of those parameters. Let St be the sets, k the constants and B the properties of those sets and constants. Let further v be the set of machine variables and I the invariant. We have the following existential proofs:

1. $\exists\, p.C$ - there are parameters that fulfil the constraints,

2. $C \Rightarrow (\exists\, St, k.B)$ - given the constraints, there are sets and constants that fulfil the properties,

3. $C \wedge B \Rightarrow \exists\, v.I$ - given the constraints and properties, there are values for the variables, such that they fulfil the invariant.

The second set of proof obligations are towards the consistency of the initialisation and the operations. Before we list those, we need some notation taken from [49].

Definition 1.3 Let S be a statement and P a predicate, then the notation $[S]P$ denotes the *weakest precondition* for S to achieve P.

That means $[S]P$ is a predicate, which is true in all those states, such that executing S always leads to a state where P is true. This notation is needed in the proof obligations for the operations consistency. If S is an operation statement and I the invariant, then for each operation we need to show that the predicate $[S]I$ is true before the operation is executed.

Let C again be the constraints of the parameters and B the properties of the sets and constants. Let I be the invariant, P the preconditions of an operation and T and S the respective statements of the initialisation and operation, then we have the following proof obligations:

4. $C \wedge B \Rightarrow [T]I$ - given the constraints and properties, it follows the weakest precondition, such that executing the initialisation fulfils the invariant.

5. $B \wedge C \wedge I \wedge P \Rightarrow [S]I$ - given the constraints, properties invariant and preconditions of the operation being true in a state, it follows that the weakest precondition $[S]I$ is true in that state.

Both proof obligations say that executing the initialisation or the operation respectively, leads to a state where the invariant is still true. So, by proving those obligations respectively for the initialisation and every operation, we show that the invariant holds in every state. All obligations together prove the consistency of a model.

We take as example the operation $enroll(s)$ from the machine *Course* above. The machine has no parameters and therefore no constraints, so the constraints C are trivially true. There is just one property of the constant max_num, so we have $max\_num = 20$ for the properties B. The invariant states that the variable *class* is a subset of the deferred set *Student*. The precondition of the operation indicates that the element given as parameter to the operation must be an element of the set *Student* and the number of students in the class is limited by the constant max_num. The statement S of the operation adds the element s to the class. In short we have:

$$\begin{aligned} C: &\quad true \\ B: &\quad max\_num = 20 \\ I: &\quad class \subseteq Student \wedge card(class) \leq max\_num \\ P: &\quad s \in Student \wedge s \notin class \wedge card(class) < max\_num \\ S: &\quad class := class \cup \{s\} \end{aligned}$$

The proof obligation for the operation $enroll(s)$ looks as follows:

$$\begin{aligned} & true \wedge (max\_num = 20) \wedge (class \subseteq Student \wedge card(class) \leq max\_num) \\ & \wedge (s \in Student \wedge s \notin class \wedge card(class) < max\_num) \\ \Rightarrow \quad & [class := class \cup \{s\}](class \subseteq Student \wedge card(class) \leq max\_num) \end{aligned}$$

Substituting the variable *class* in the invariant according to the statement leads to

$$\begin{aligned} & true \wedge (max\_num = 20) \wedge (class \subseteq Student \wedge card(class) \leq max\_num) \\ & \wedge (s \in Student \wedge s \notin class \wedge card(class) < max\_num) \\ \Rightarrow \quad & (class \cup \{s\} \subseteq Student \wedge card(class \cup \{s\}) \leq max\_num) \end{aligned}$$

Since we have $class \subseteq Student$ as invariant and $s \in Student$ as precondition of the operation, it follows that $class \cup \{s\}$ is a subset of *Student*. Therefore the first part of the invariant is fulfilled when executing the operation $enroll(s)$.

Now let's look at the second part. We have

$$card(class \cup \{s\}) = card(class) + card(\{s\}),$$

since s is not yet in *class* because of the precondition of the operation. The cardinality of a set with one element is of course 1, so we have

$$card(class \cup \{s\}) = card(class) + 1$$

Now we use the fact from the precondition, that $card(class)$ is truly smaller than max_num, so that $card(class) + 1$ is still smaller or at most equal to max_num. Consequently we have

$$card(class \cup \{s\}) \leq max\_num$$

which proves that the invariant still holds after adding one student to the class, i.e. executing $enroll(s)$.

Proving all obligations by hand would be quite tedious. Fortunately, tools have been developed, such as AtelierB [53] or the B-toolkit [6], that can do most of this work. Unfortunately, unlike our example, proofs can get so complex that user intervention is still necessary when using automatic provers. Model checking is another approach to verify a model and can be done completely automatically. Later in this chapter, we will discuss this approach further.

1.2.2 Refinement

The concept of refinement allows an incremental software development starting with an abstract specification and then stepwise adding more and more details in each refined model, resulting in a chain of refinements, that ends with the stage where executable code can be generated automatically. The process of refinement can be roughly divided in two parts. The first part of the refinement chain usually involves developing a more accurate description of the requested software product. In the second part are then the abstract data structures and non-deterministic statements of the model translated to concrete implementable data structures and deterministic instructions. In order to do so, there are a few additional constructs for B-refinements such as the sequential execution of statements or the introduction of local variables.

The following model is a refinement of the *Course* Machine in Section 1.2. Since the example is very small, we did several steps at once in the refinement. We decided to add another variable sz to store the number of students in the class, we substituted the preconditions of the operation with concrete if-statements and we replaced the parallel statements with sequential statements.

REFINEMENT
CourseR

REFINES
Course

VARIABLES
class,
sz

INVARIANT
$sz \in \mathbb{N} \wedge$

$card(class) = sz$
$sz \leq max\_num$

INITIALISATION
$class := \emptyset$;
$sz := 0$

OPERATIONS
$enroll(s) \mathrel{\widehat{=}}$
IF
$s \in Student \wedge$
$s \notin class \wedge$
$sz < max\_num$
THEN
$class := class \cup \{s\}$;
$sz := sz + 1$
END;

$out \leftarrow leave(s) \mathrel{\widehat{=}}$
IF
$s \in class$
THEN
$class := class - \{s\}$;
$sz := sz - 1$;
$out := s$
END
END

A correct refinement behaves in the same manner as the specification does. That means that any sequence of operations that can be executed in the specification within preconditions, can also be executed in the refinement [11]. We can easily see that the refinement of the *Course* machine fulfils this requirement. For a more detailed introduction to refinement, see [49], and for a more formal description, see [37].

1.3 The ProB Model Checker

For model checking of B-specifications there is currently just one tool available, which is called ProB [36]. ProB is written in Prolog [50] and has a TclTk [54] user interface, which allows the user to edit his model directly in ProB. The user can do much more than just editing and then model checking a specification with ProB, though. The ProB tool allows animation of a specification, which means the user can click on any enabled operation in the *Enabled Operation* field, see Figure 1.3, to walk through the state space manually.

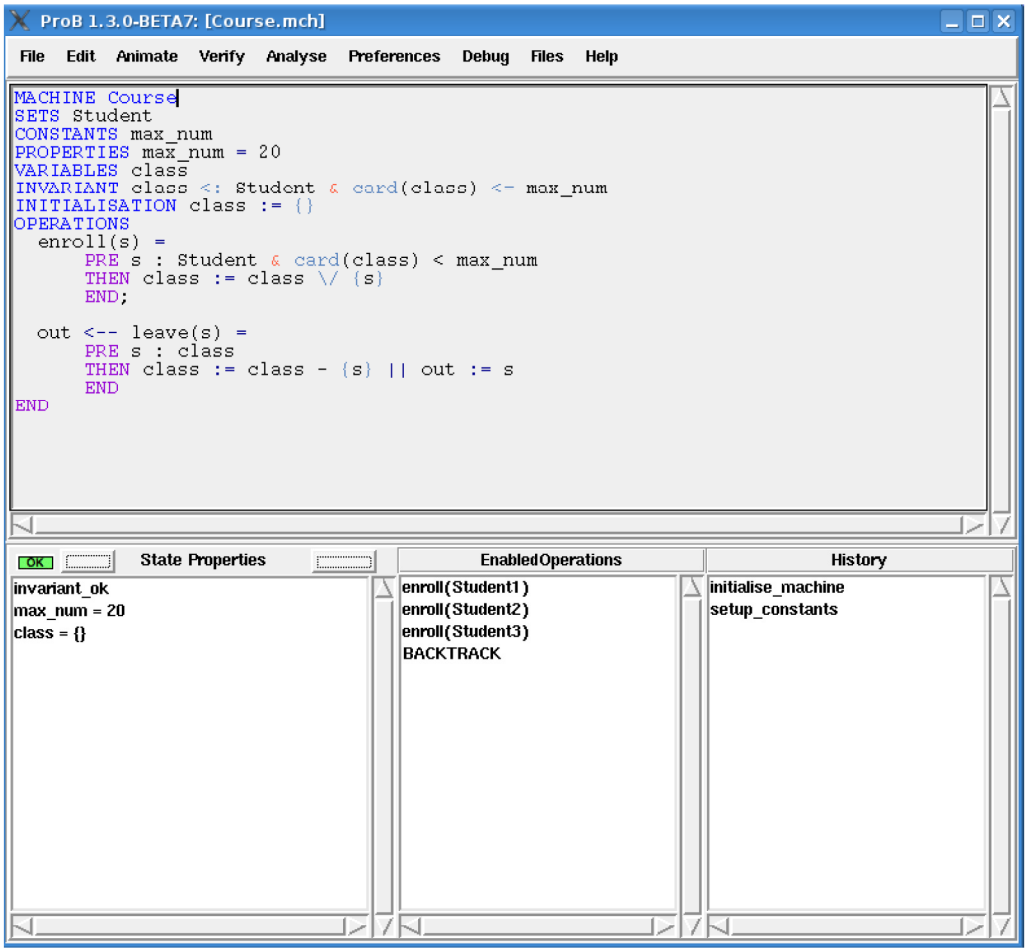


Figure 1.3: Screenshot ProB

This way, the user can gain a much better understanding of his model. Additionally, the user also has the possibility to visualise the state space that has been encountered, through clicking on the enabled operations. ProB uses the graph drawing package *dot* [5] for visualisation.

In Figure 1.4, we can see part of the state space of the *Course* machine from Section 1.2, after setting up the constants, initialising the machine and enrolling *Student*2 into the class. States that have been visited are visualised with green borders, and states that have been encountered, but not visited, are visualised with a red border. The current state, here $class = \{Student2\}$, is especially visualised with a green double border polygon. An arrow going from one state to another is labelled with the operation (including parameters), that leads to the next state.

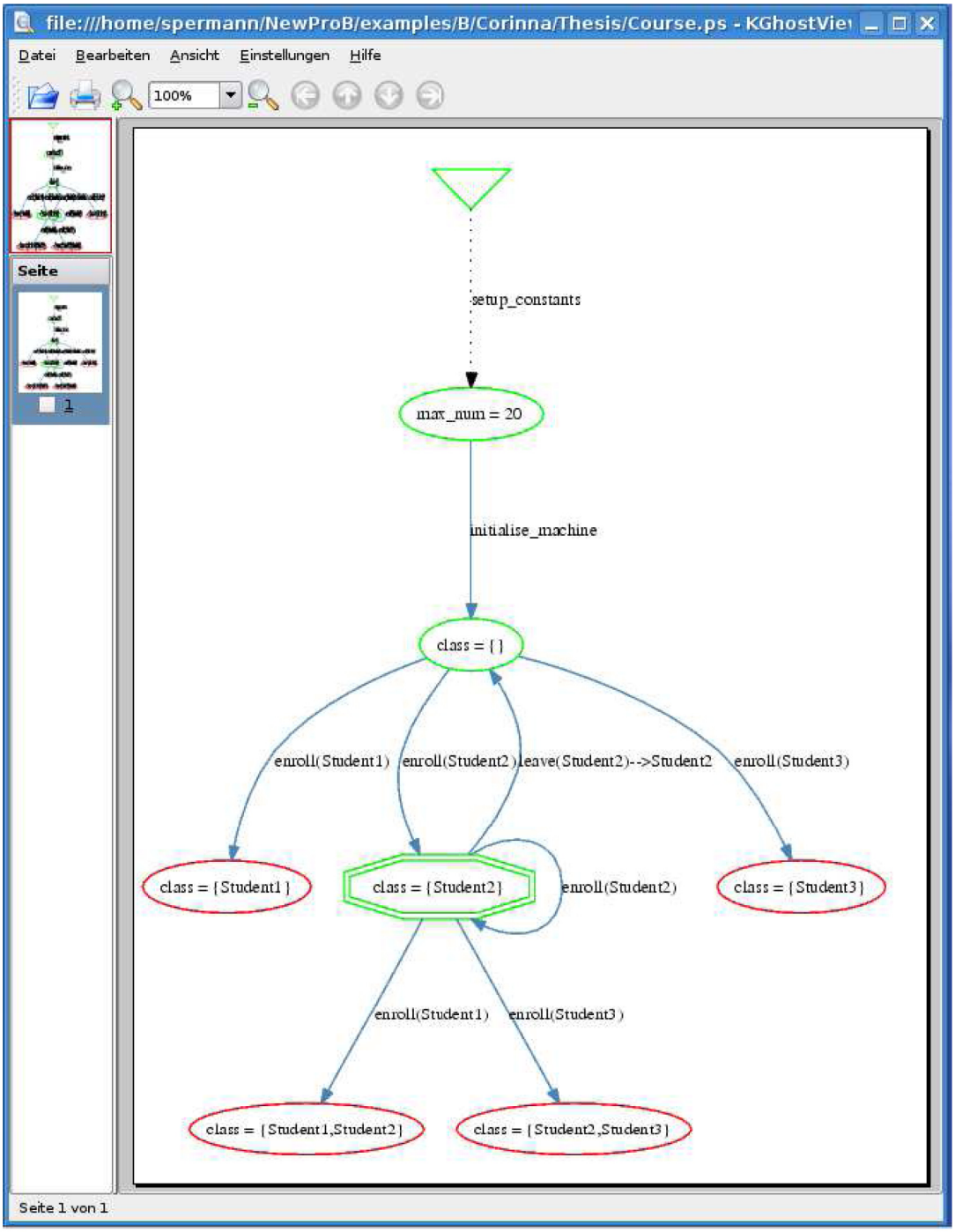


Figure 1.4: Encountered State Space

We now want to decribe the model checking algorithm ProB uses.

Algorithm 1.4[*ProB Model Checking Algorithm*]

Input: An abstract machine M

1. $Queue := \{root\}$; $Visited := \{root\}$; $SGraph := \{root\}$
2. **while** *Queue* is not empty **do**
3. **if** random(1) $< \alpha$ **then**
   state := pop_from_front(*Queue*); /* depth-first */
   else
   state := pop_from_end(*Queue*); /* breadth-first */
   end if
4. **if** $error(state)$ **then**
   return counter-example trace in *SGraph* from *root* to *state*
5. **else**
   for all *succ*,*Op* **such that** $state \rightarrow^{M}_{Op} succ$ **do**
   $SGraph := SGraph \cup \{state \rightarrow^{M}_{Op} succ\}$
   if $succ \notin Visited$ **then**
   add *succ* to front of *Queue*
   $Visited := Visited \cup \{succ\}$
   end if
   end for
   end if
6. **od**
7. **return** ok

The algorithm takes an abstract machine M as input, and returns either *ok* when there is no error in the model, or the trace from the *root* node to an error state. An error state can mean one of three things: The state is in violation of the invariant, or is a deadlock state, or is both of the above. The ProB user can decide if deadlocks are regarded as erroneous, or not.

First, the algorithm initialises three variables used throughout the model checking process. We have the variable *Queue* for states that have been encountered, but not evaluated yet. Evaluation of a state means that it has been model checked, and all its successor states have been computed. The second variable, *Visited*, holds all states that are in the queue, or have been evaluated already. The third variable, *SGraph*, contains the visited state space; that means, all the visited states, together with the transitions between them. This allows the creation of traces to erroneous states, whenever they occur.

At the beginning, all variables contain just the *root* node. The *root* node is a pseudo state that serves as a starting point for the model checking algorithm, and is therefore always a correct state.

The *while* loop starting in step 2 is executed as long as there are states in the variable *Queue* to be evaluated. The randomisation in the next step is not strictly necessary for model checking, since it only decides, depending on the user defined value α, whether the algorithm takes more of a breadth-first or depth-first strategy. This helps to find potential errors more quickly, see [38]. When picking a state from the front of the queue, a depth-first search strategy is chosen, and when a state from the end of the queue is picked, then the algorithm takes a breath-first search approach. Generally, error states are encountered quickest when both strategies are mixed equally.

In the next step, the algorithm makes use of a function *error*(*state*), which decides whether the state picked in step 3 is an error state or not. As mentioned earlier, an error state can be an invariant violation, a deadlock or both, depending on the user's choice. If an error state is encountered, then the algorithm returns, and gives as output the trace from the *root* node to the error state. This trace is visible for the user as a sequence of operations in the *History* window of the ProB tool. The user can also view the history to the error state graphically in ProB[2]. If the state is not an error state, then the algorithm continues with step 5.

In step 5, all successor states are first computed and added to the state graph *SGraph*. Then all successor states that have not been visited yet, are added to the front of the *Queue* and also marked as visited states, because once a state is in the *Queue* it will be evaluated by the algorithm. Once all states are evaluated and no error state has been encountered, the algorithm returns with *ok*, stating that the machine is correct.

We mentioned above that the algorithm uses a mixture of a breadth-first and depth-first strategy for efficiency reasons. This of course only matters, if a model contains errors or deadlocks.[3] Usually an error is one of two types: Either it is an error in an operation, so that it occurs in most states reached by executing this operation, or it is an error (such as a deadlock) that arises when the machine is animated for long enough. In the first case, it is best to try out all operations for all their arguments systematically, which is reflected by a breadth-first search. The second case, when an error occurs in some long path of consecutive states, calls for a depth-first search strategy. Neither strategy works well for both kind of errors, but from experience mixing them equally leads to the best result. For more information on this and the ProB tool in general, see [38].

[2]In ProB the user chooses the tab *Animate* → *View* → *History to Current State*

[3]That is, if the user has chosen to find deadlocks.

1.4 Discussion: Model Checking versus Mathematical Proof

Model checking is a completely automated method to verify the correctness of a model. However, is it really the same as a mathematical proof in B? The answer is no, as the following example shows:

MACHINE *Unreachable*
VARIABLES x
INVARIANT $x \in \mathbb{N}$
INITIALISATION $x := 1$
OPERATIONS
Increment $\widehat{=}$
$x := x + 1;$
Decrement $\widehat{=}$
PRE
$x \neq 1$
THEN
$x := x - 1$
END
END

The machine has one variable x, which is a natural number[4] and two operations, one for incrementing and one for decrementing x. When we run this machine in the ProB model checker, then there will be no errors found. Yet we cannot prove the correctness using B's proof schema performing an induction! More precisely, we cannot verify that the decrement operation always holds the invariant. This is because the precondition of the operation is $x \neq 1$. So, if we had a state with $x = 0$, then the decrement operation could be executed and we would end in the state where $x = -1$, which violates the invariant $x \in \mathbb{N}$. That means that, from a mathematical point of view, the model is not correct. On the other hand the error state with $x = -1$ can never be reached, because the machine is initialised with $x = 1$. Consequently, if a counter of a rocket launcher was specified with this model, nothing would go wrong as long it was initialised in the same way.

The question here is, if the initialisation belongs to the model, or not. More generally, we have to ask how unreachable states should be treated. When proving the correctness mathematically, there is no distinction made between reachable and unreachable states. This is why the proof of the intuitively correct model, above, failed. If we want to make this distinction and decide that only reachable states concern us, then the unreachable states can get in the way of a mathematical proof.

[4]The notation $\mathbb{N}$ means in B the set of non-negative integers

Model checking works better in this situation, because it only deals with the reachable state space. Of course, we then have to regard the initialisation as part of the model, and any implementation has to initialise its variables in a respective manner.[5]

[5]The question above has a bit of a philosophical nature, and could explain why theses with a related topic are written to obtain a doctor degree in philosophy in many countries.

Chapter 2

Symmetry

2.1 Motivation

Model checking is a very friendly way of verifying a specification. A system designer only has to press a button, and the model checker does all the work for him. The remaining question is how long that might take. Indeed, the size of the state space grows exponentially with the number of state variables. Since a standard model checker has to go though the entire state space, the time for model checking also increases exponentially. This is known as the *state space explosion problem* in model checking. Consequently, there is a great incentive to reduce the state space somehow, and there have been various approaches developed in order to do so. Those methods include partial order reduction [21], symbolic model checking [44] or symmetry reduction [28]. For a nice summary of the most popular methods, see [55].

In this thesis, we want to concentrate on symmetry reduction, because symmetries arise naturally in B models. We will explain shortly why that is, but first we want to explain intuitively what we mean by symmetry with regard to model checking. When we look again at the state space of the security door example from Section 1.1 in Figure 2.1, then we see, by drawing a vertical axis through state s_1, that the state space is symmetric. This type of symmetry, though, is not very useful, because we have to develop the entire state space in order to find such symmetries, which is what we just wanted to avoid.

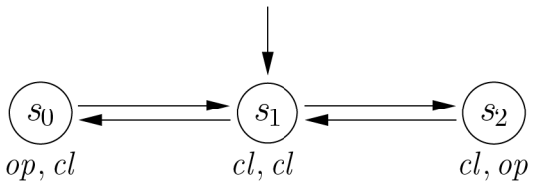


Figure 2.1: State graph from security door example

Let's have a closer look at the states, instead. We see that the two states $s_0 = (d1 = op, d2 = cl)$ and $s_2 = (d1 = cl, d2 = op)$ are somewhat similar. Swapping around the numbering of the doors in s_0 leads directly to state s_2 and vice versa. In fact, we don't need to know which door is the first or second one to reason about the system. All we wanted to specify is a two door system such that, at most, one door is open at all times. Since there is just a set of doors and no distinction between individual doors, we have to verify the requirement only for one of the two states. For the other state it follows by swapping $d1$ with $d2$. This kind of symmetry is far more useful, because it can be applied during the exploration of the state space. When encountering a new state s, it is checked first, if that state emerges from an already explored state by exchanging interchangeable elements. If this is the case, then s no longer needs to be verified.

We now want to explain how symmetry arises in B. Let's take for example the following B-machine, modelling a very simple database of members of a club, with two operations for persons to join or leave the club.

MACHINE *Club*

SETS
$Person$

VARIABLES
$member$

INVARIANT
$member \subseteq Person$

INITIALISATION
$member := \emptyset$

OPERATIONS
$join(p) \mathrel{\widehat{=}}$
PRE
$p \in Person \wedge$
$p \notin member$
THEN
$member := member \cup \{p\}$
END;

$leave(p) \mathrel{\widehat{=}}$
PRE
$p \in member$
THEN
$member := member - \{p\}$
END
END

The machine contains one deferred set *Person*. That means, the cardinality of the set is not specified, and its elements are not enumerated (and are given no name). Consequently, neither within the invariant nor within the operations, can the model refer to a specific element of that set. We assign a cardinality to the deferred set, and the set is instantiated, e.g. $Person = \{p1, p2, p3\}$. This is done because we need to make the state space finite in order to be able to decide via model checking, if the model is correct. For this instantiation, the state space of the model would look as depicted in Figure 2.2.

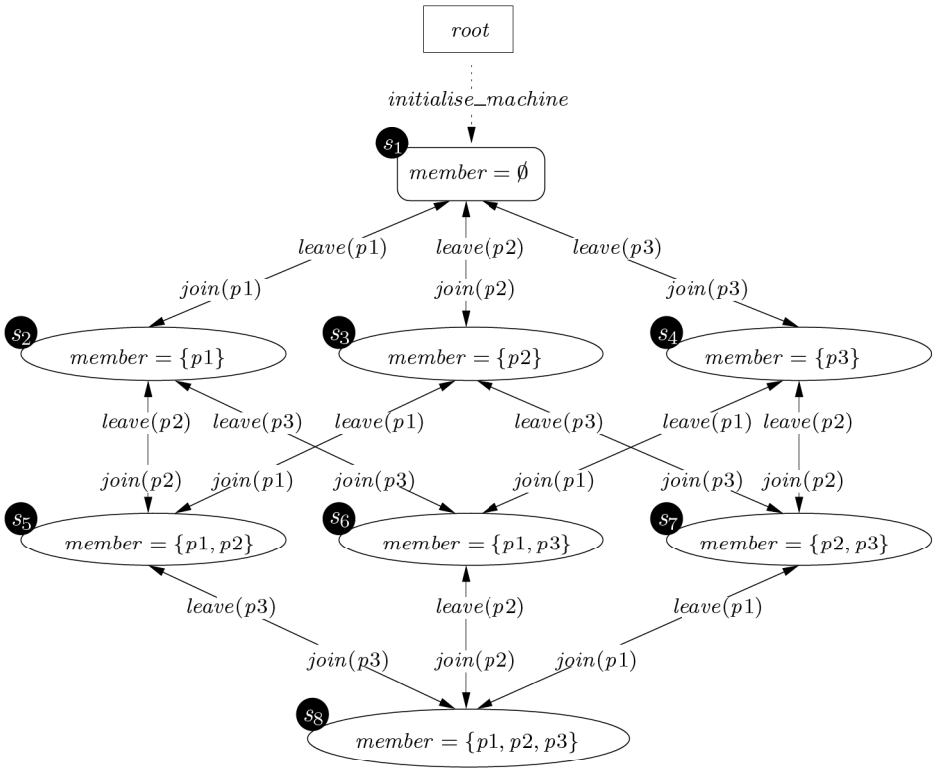


Figure 2.2: State space of the club machine

We can see that the states s_2, s_3 and s_4 are symmetric, in that

1. the states can be transformed into each other by permuting the elements of the set *Person*;

2. if one of the states satisfies (respectively violates) the invariant, then any of the other states must also satisfy (respectively violate) the invariant;

3. if one of the states can perform a sequence of operations, then any other state can perform a similar sequence of transitions, possibly substituting operation

arguments in the same way that the state values were permuted. For example, state s_2 can perform $leave(p1)$, state 3 can be obtained from state s_2 by replacing $p1$ with $p2$ and indeed, state s_3 can perform $leave(p2)$.

The same is true for the states s_5, s_6 and s_7. Consequently, the state space can be reduced as in Figure 2.3.

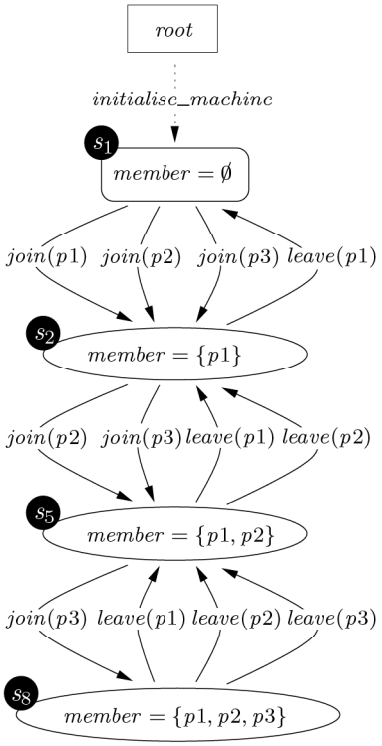


Figure 2.3: State space of the club machine with symmetry

Let's have a closer look at the new state space. The state space is now reduced to a set of representative states, which are the states $1, 2, 4$ and 8 from the original state graph. We can see that, for example the transitions $join(p1)$ and $join(p2)$ performed directly after the initialisation, lead to state $s2 = (member = \{p1\})$. This is because the elements of the set *Person* are interchangeable, so that both transitions lead to the same representative state.

Even for this small example, we have already halved the number of states that need to be considered. The number of transitions is indirectly reduced through the reduction of the number of states. Note that in Figure 2.2 we had each edge representing a transition in each direction. We will explain in Section 2.3 why the suggested state space reduction is sound. Beforehand, we need some mathematical background described in the next section.

2.2 Mathematical Background to Symmetry

Symmetry and group theory are very closely related. Indeed, mathematical groups are often used to express symmetries. Because of this close relationship, we need to introduce a few definitions and notations from group theory. We've taken the following mathematical definitions from [4].

Definition 2.1 A *group* is a set G together with a multiplication $\circ$ on G, which satisfies three axioms:

1. The multiplication is associative, that is, $(x \circ y) \circ z = x \circ (y \circ z)$ for any three elements x, y, z in G.
2. There is an element e in G, such that $x \circ e = x = e \circ x$ for every x in G. The element e is called identity element.
3. Each element x of G has a so called inverse x^{-1}, which belongs to the set G and satisfies $x^{-1} \circ x = e = x \circ x^{-1}$.

Definition 2.2 A *subgroup* H of a group G, is a subset of G, which itself forms a group under the multiplication of G, we write $H \leq G$.

Definition 2.3 Let $X = \{x_1, \ldots, x_n\}$ be a subset of a group G then, $\langle x_1, \ldots, x_n \rangle$ denotes the smallest subgroup of G containing $\{x_1, \ldots, x_n\}$. The subgroup is *generated by* X. If the subgroup equals to G itself then X is called a *set of generators for* G.

For this work we are particularly interested in permutation groups. A permutation of a finite set X is a bijection from X onto itself, $\alpha : X \to X$. The set of all permutations of X form a group S_X under the composition of functions. This group has as identity element e the permutation, that does not exchange any element of X.

Example 2.4 Let's take the set of integers $S = \{1, 2, 3, 4, 5, 6\}$. The function $\alpha : \{1, 2, 3, 4, 5\} \to \{1, 2, 3, 4, 5\}$ with $\{1 \mapsto 4, 2 \mapsto 2, 3 \mapsto 5, 4 \mapsto 1, 5 \mapsto 6, 6 \mapsto 3\}$ is a permutation of S. It is

$$\alpha = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 2 & 5 & 1 & 6 & 3 \end{bmatrix}.$$

This notation from Example 2.4 can be shortened by using *cyclic notation*. The permutation α in cyclic notation is written as

$$\alpha = (14)(356).$$

Inside each pair of brackets, an element is sent to the element following it, and the final element is sent to the first. Which means, 1 is sent to 4 and 4 is sent to 1, 3 is sent to 5, 5 to 6, and 6 to 3. Elements that don't occur in the cyclic notation remain unchanged. A permutation, that can be described inside a single pair of brackets, is called a *cyclic permutation*. Any permutation can be written as a set of cyclic permutations. We say that the permutation is in *cyclic notation*.

Definition 2.5 The permutation group over the set of the first n positive integers is called the *symmetric group* of degree n, and denoted S_n.

Example 2.6 The group $S_{\{1,2,3\}} = S_3$ has the identity element

$$e = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix},$$

and the following group elements:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}.$$

In cyclic notation we have $S_3 = \{e, (12), (23), (132), (13), (123)\}$

Permutation groups can be used to describe symmetries on geometric objects.

Example 2.7 The group

$$G = \left\{ \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix} \right\} = \{e, (132), (123)\}$$

is a subgroup of S_3, since e is in G and the elements $\alpha = (132)$ and $\beta = (123)$ are inverse of each other. The concatenation $\alpha \circ \beta = (132) \circ (123)$, where β is applied first, gives, 1 is sent to 2 by β and 2 is sent back to 1 by α, 2 is sent to 3 by β and 3 is sent back to 2 by α, 3 is sent to 1 by β and 1 is sent back to 3 by α, so $\alpha \circ \beta = e$ is indeed the identity. All other concatenations of two elements in G are in G, too, so G is a group.

If we consider the set $X = \{1, 2, 3\}$ as numbering for the edges of a regular triangle, then the group G is exactly the set of permutations of X, that describes all symmetric triangles under rotation around the triangles midpoint M, see Figure 2.4.

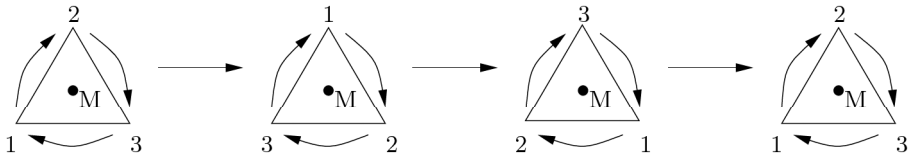


Figure 2.4: Rotational symmetry of a triangle

The reflections on a line between the midpoint M and any triangle vertex can be described with the transpositions $(12), (13)$ and (23). Those transpositions together with the elements of the group G build the group S_3. So we see that S_3 describes all symmetries in a triangle. That is to say, all elements in S_3 preserve the structure of a triangle.

The concept of symmetry described with permutation groups can also be applied to Kripke structures. Definitions in the following are adapted from Section 14 of [13].

Definition 2.8 Let $K = (S, S_0, R, AP, L)$ be a Kripke structure. A permutation α of S is called an *automorphism* of the Kripke structure K, if and only if α keeps the transition relation $R \subseteq S \times S$. That means, α meets the following condition:

$$(s_1, s_2) \in R \Rightarrow (\alpha(s_1), \alpha(s_2)) \in R \ \forall\, s_1, s_2 \in S.$$

Let G be a permutation group on S, then G is an *automorphism group* of K, if and only if every permutation $\alpha \in G$ is an automorphism of K.

Definition 2.8 does not refer to the labelling function L of the Kripke structure, but only to the transition relation R. Automorphism groups that also leave the labelling unchanged are called *invariance groups*.

Definition 2.9 Let G be an automorphism group of the Kripke structure $K = (S, S_0, R, AP, L)$. Then G is an invariance group for an atomic proposition $p \in AP$, if and only if the following condition holds:

$$(\forall\, \alpha \in G)(\forall\, s \in S)(p \in L(s) \Leftrightarrow p \in L(\alpha(s)))$$

We say that p is *invariant* under G. We call G an *invariance group*, if every atomic proposition in AP is invariant under G.

Example 2.10 Let's continue Example 1.2. There we had the Kripke structure $K = (S, S_0, R, AP, L)$ describing a security door, with:

$$S = \{s_0, s_1, s_2\}$$
$$S_0 = \{s_1\}$$
$$R = \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1)\}$$
$$AP = \{d1 = op, d1 = cl, d2 = op, d2 = cl\}$$
$$L(s_0) = \{op, cl\}, L(s_1) = \{cl, cl\}, L(s_2) = \{cl, op\}$$

Let α be a permutation on the set of states S, that exchanges s_0 with s_2 and leaves s_1 fixed. If we look at the graphical representation of the Kripke structure K in Figure 1.1 from Example 1.2, we see intuitively that α is an automorphism. To prove this intuition, we have to show, that for each transition $(s_i, s_j) \in R$, with $i, j \in 0, \ldots, 2$, there is a transition $(\alpha(s_i), \alpha(s_j)) \in R$, with $i, j \in 0, \ldots, 2$. We are going to take the transition (s_0, s_1) as example, all other transitions can be examined in a similar way. For the transition (s_0, s_1), we have $(\alpha(s_0), \alpha(s_1)) = (s_2, s_1)$, which is a transition in R. The same can be observed for all transitions in R, so α is an automorphism of K. Since applying α twice results in the identity, we have the automorphism group $G = (e, \alpha)$ for K. This group is also an invariance group, since both states s_0 and s_2 are labelled with op and cl.

We haven't explained yet how the symmetry within Kripke structures can help to reduce the effort in model checking. Before we do this, we need one more concept from group theory:

Definition 2.11 Let X be a set $x \in X$, and G a permutation group on X. The set of all images $\alpha(x)$, as α varies through G, is called the *orbit* of x and written Gx:

$$Gx = \{\alpha(x) \mid \alpha \in G\}$$

The set of orbits are equivalence classes under the relation

$$x \sim y, \text{ if } \exists \alpha \in G \text{ such that } \alpha(x) = y.$$

Each equivalence class Gx can be represented by one representative $rep(Gx)$ of Gx.

Applying this concept to Kripke structures, we can reduce the model to its quotient model, flattening all states in one orbit to their representative state.

Definition 2.12 Let $K = (S, S_0, R, AP, L)$ be a Kripke structure and G an invariance group on S. The quotient model of K is a tuple $Q_G = (S, S_G^0, R_G, AP, L_G)$ where

1. $S_G = \{Gx \mid x \in S\}$, is the set of orbits of S,
2. $S_G^0 = \{Gx \mid x \in S^0\}$ is the set of orbits, that have an initial state as representative,
3. $R_G = \{(Gx, Gy) \mid (x, y \in R)\}$ is the transition relation,
4. L_G is the labelling function and given by $L_G(Gx) = L(rep(Gx))$.

Since G is an invariance group, R_G and L_G are independent of the chosen representative.

Example 2.13 We want to take once again our security door example. We've already shown that the group $G = (e, \alpha)$, where α exchanges the states s_0 and s_2, is an invariance group on K. Now, the orbits are $\{s_0, s_2\}$ and $\{s_1\}$. We take s_0 and s_1 as the representatives and obtain the quotient model, see Figure 2.5.

Example 2.14 Let's now look at the slightly larger *Club* example. We assume again that the deferred set *Person* is instantiated as a set with three elements. Consequently, we have a finite set of states, and we can represent the machine with the following Kripke structure K:

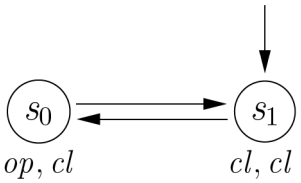


Figure 2.5: Quotient model of security door example

$$
\begin{aligned}
&S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\},\\
&S_0 = \{s_1\},\\
&R = \{\\
&\qquad (s_1, s_2), (s_1, s_3), (s_1, s_4),\\
&\qquad (s_2, s_1), (s_2, s_5), (s_2, s_6),\\
&\qquad (s_3, s_1), (s_3, s_5), (s_3, s_7),\\
&\qquad (s_4, s_1), (s_4, s_6), (s_4, s_7),\\
&\qquad (s_5, s_2), (s_5, s_3), (s_5, s_8),\\
&\qquad (s_6, s_2), (s_6, s_4), (s_6, s_8),\\
&\qquad (s_7, s_3), (s_7, s_4), (s_7, s_(),\\
&\qquad (s_8, s_5), (s_8, s_6), (s_8, s_7)\\
&\quad \},\\
&AP = \{\\
&\qquad card(member) = 0,\\
&\qquad card(member) = 1,\\
&\qquad card(member) = 2,\\
&\qquad card(member) = 3\\
&\quad \},\\
&L(s_1) = \{\, card(member) = 0 \,\},\ L(s_2) = \{\, card(member) = 1 \,\},\\
&L(s_3) = \{\, card(member) = 1 \,\},\ L(s_4) = \{\, card(member) = 1 \,\},\\
&L(s_5) = \{\, card(member) = 2 \,\},\ L(s_6) = \{\, card(member) = 2 \,\},\\
&L(s_7) = \{\, card(member) = 2 \,\},\ L(s_8) = \{\, card(member) = 3 \,\}
\end{aligned}
$$

In order to apply symmetry reduction, we are only interested in those automorphisms that also leave the labelling invariant. That means, we need to find the invariance group G of K. We discussed earlier how elements of deferred sets can be exchanged with one another. This is where we want to start looking for automorphisms.

In the Kripke structure, see Figure 2.6, there is no reference to any concrete element. The Kripke structure only reflects how many elements are in the set-variable *member*. It seems that we can just swap the states s_2 and s_3, but this does not lead to an automorphism.

Let α be the permutation that swaps s_2 with s_3, and leaves all other states fixed. This permutation is not an automorphism of the Kripke structure K, since the

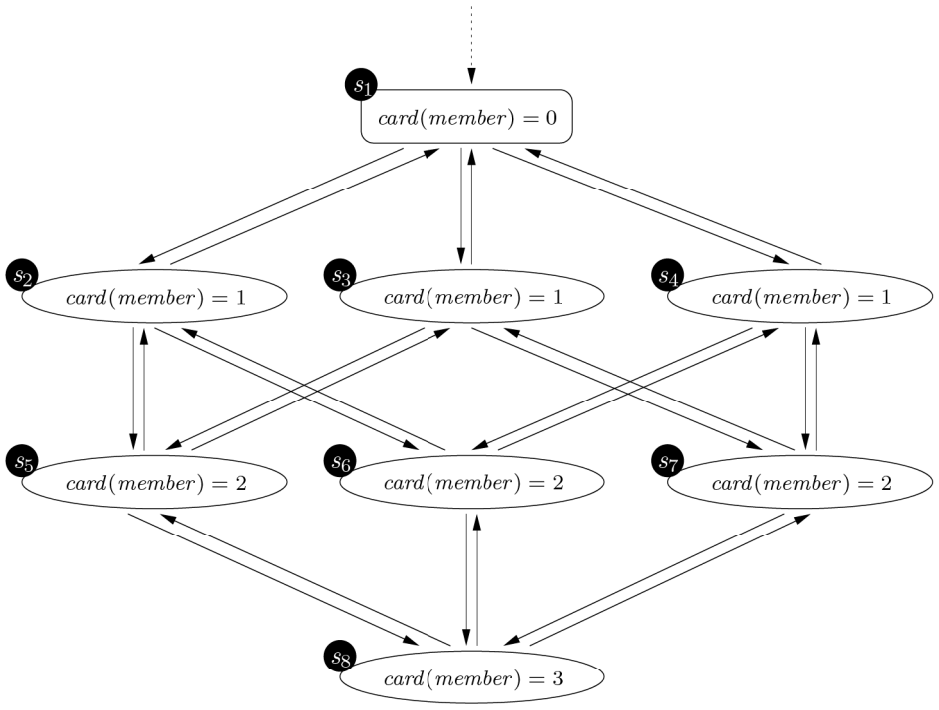


Figure 2.6: Kripke structure of the club machine

transitions

$$(\alpha(s_2), \alpha(s_6)) = (s_3, s_6) \text{ and } (\alpha(s_2), \alpha(s_6)) = (s_2, s_7)$$

are not part of the relation R of K.

However, if we exchange the states s_6 and s_7 as well, then we do indeed get an automorphism. When we look at the labelling of those two states, we see that they are the same. So the exchange of two elements of the deferred set did lead us to an automorphism, after all. Since the other states are not permuted, this automorphism is already an element of our invariance group G. In the same fashion, we get another two automorphisms, one by swapping $s2$ with $s4$, and $s5$ with $s7$, and another by swapping $s3$ with $s4$, and $s5$ with $s6$. So, apart from the identity, we have now the following three automorphisms:

$$\alpha = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 3 & 2 & 4 & 5 & 7 & 6 & 8 \end{bmatrix} = (23)(67)$$

$$\beta = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 4 & 3 & 2 & 7 & 6 & 5 & 8 \end{bmatrix} = (24)(57)$$

$$\gamma = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 4 & 3 & 6 & 5 & 7 & 8 \end{bmatrix} = (34)(56)$$

There are more automorphisms of the Kripke structure as we can infer from the symmetric structure of the state space, see Figure 2.6, but none of them leaves the labelling invariant. So the group

$$G = \{e, \alpha, \beta, \gamma\}$$

is our invariance group.

Now let's calculate the orbit for each state. For the state s_1, the orbit contains only the element s_1 itself, since all automorphisms in G map s_1 onto itself. In the case of s_2, the situation is more interesting. The state s_2 is mapped to s_3 by α, to s_4 by β, and to $s2$ itself by γ, so the orbit of s_2 is the set $\{s_2, s_3, s_4\}$. Below we list the orbit for each state:

$$\begin{array}{ll} Gs_1: & s_1 \\ Gs_2: & s_2, s_3, s_4 \\ Gs_3: & s_2, s_3, s_4 \\ Gs_4: & s_2, s_3, s_4 \\ Gs_5: & s_5, s_6, s_7 \\ Gs_6: & s_5, s_6, s_7 \\ Gs_7: & s_5, s_6, s_7 \\ Gs_8: & s_8 \end{array}$$

The listing shows that we have four equivalence classes, namely: $\{s_1\}$, $\{s_2, s_3, s_4\}$, $\{s_5, s_6, s_7\}$, and $\{s_8\}$. Choosing exactly one representative of each equivalence class leads to a reduced state space, see Figure 2.7.

The quotient model of K is $Q_G = (S_G, S_G^0, R_G, AP, L_G)$ where:

$$\begin{array}{l} S_G = \{Gs_1, Gs_2, Gs_5, Gs_8\}, \\ S_G^0 = \{Gs_1\}, \\ R = \{ \\ \qquad (Gs_1, Gs_2), \\ \qquad (Gs_2, Gs_1), (Gs_2, Gs_5), \\ \qquad (Gs_5, Gs_2), (Gs_5, Gs_8), \\ \qquad (Gs_8, Gs_5) \\ \quad \}, \\ AP = \{ \\ \qquad card(member) = 0, \\ \qquad card(member) = 1, \\ \qquad card(member) = 2, \\ \qquad card(member) = 3 \\ \quad \}, \\ L_G(Gs_1) = L(s_1) = \{\ card(member) = 0\ \}, \\ L_G(Gs_2) = L(s_2) = \{\ card(member) = 1\ \}, \\ L_G(Gs_5) = L(s_5) = \{\ card(member) = 2\ \}, \\ L_G(Gs_8) = L(s_8) = \{\ card(member) = 3\ \}, \end{array}$$

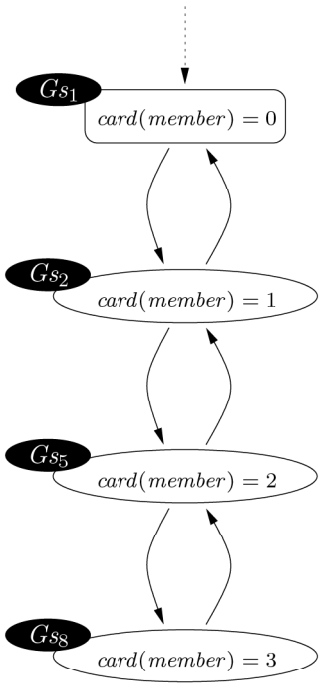


Figure 2.7: Quotient model of the Kripke structure K

Unlike our little examples, the reduction from the model to the quotient model can be quite respectable, so that model checking of the quotient model needs far less resources. The following Theorem states that model checking of a model is equivalent to model checking of the quotient model.

Theorem 2.15 Let $K = (S, S_0, R, AP, L)$ be a Kripke structure, G be an automorphism group of K, and f be a CTL^* formula. If G is an invariance group for all the atomic propositions p occurring in f, then

$$K, s \models f \Leftrightarrow K_G, Gx \models f$$

where K_G is the quotient model corresponding to K.

For a proof of this theorem, see [13] chapter 14.

We haven't yet discussed how easy or difficult it is to apply symmetry during model checking. The modification of the model checking algorithm is fairly straight forward. Instead of exploring every single state s, each state is mapped by a function ϕ to the unique representative state $\phi(s)$ of the orbit it belongs to, see [28]. So we need to calculate the orbit relation to determine ϕ. This leads us to the so called *orbit problem*, which asks in our case, if two states belong to the same orbit. We give a formal definition of this problem.

Definition 2.16 Let G be a group acting on the set $\{1, 2, \ldots, n\}$, which has a finite set of generators. Let D be a set and $x, y \in D^n$ two n-dimensional vectors. The question, whether there exists a permutation $\alpha \in G$ such that $y = \alpha(x)$, is called the *orbit problem*.

Another related mathematical problem is the *Graph Isomorphism problem*. We will reduce the orbit problem of symmetry reduction to the Graph Isomorphism problem in Section 2.4. Therefore we want to introduce here some definitions from graph theory adapted from [18].

Definition 2.17 A graph is an ordered pair $G = (V, E)$, that consists of a finite set V of vertices and a finite set $E = V \times V$ of edges. A graph is called a (un-)directed graph, if the pairs of vertices in E are (un-)ordered pairs.

Unless stated otherwise, a graph is always a directed graph in the following.

Definition 2.18 Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a bijection $f : V_1 \to V_2$ such that

$$(x, y) \in E_1 \text{ if and only if } (f(x), f(y)) \in E_2,$$

where (x, y) and $(f(x), f(y))$ are ordered pairs. The mapping f is said to be an *isomorphism* between G_1 and G_2. We will also use the notation $G_1 \cong G_2$ for isomorphic graphs. Furthermore, if $G_1 = G_2$ then f is called an automorphism of G_1.

The set of automorphisms of a graph G form a group under the operation of function composition, which we will denote $Aut(G)$. Below we present an example for two isomorphic graphs.

Example 2.19

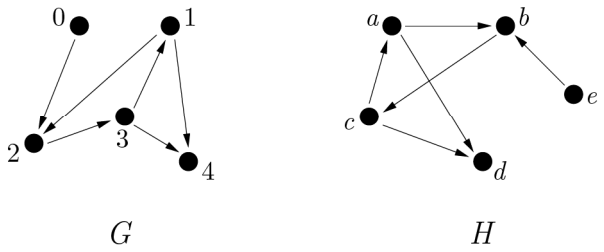


Figure 2.8: Two isomophic graphs

The function $f = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ e & a & b & c & d \end{pmatrix}$ is an isomorphism between the graphs G and H in Figure 2.8.

The question if two given graphs are isomorphic is known as the *Graph Isomorphism problem*. The complexity of the orbit problem and the Graph Isomorphism problem are related by the following Theorem.

Theorem 2.20 The orbit problem is as hard as the Graph Isomorphism problem.

A proof is given in [13] chapter 14. The Graph Isomorphism problem is in the NP complexity class, but not known to be NP-complete. So the same is true for the orbit problem and it is therefore just as mathematically challenging. Consequently, the computational effort required to find symmetries within large state spaces can be quite high.

2.3 Soundness of State Space Reduction

Now we want to outline why this state space reduction is sound. Let f be a permutation function over the deferred sets of a machine such that only elements of the same deferred set are permuted. We have to show, essentially, that for any state s and its permuted state $f(s)$, the truth value of a predicate P in s is the same as in $f(s)$.

Let's now formalise the definition of f and the above statement about f. In order to do so, we will adapt some notations and definitions from [39] and conclude with the Theorem and Corollary presented there. Variables and constants in B-expressions and predicates, have as values some basic or nested data structures. There are only three types of basic data structures. First there are atoms, i.e. elements of deferred or enumerated sets, which includes integers or Boolean values. Then we also have pairs of values or sets of values. Any other data structure can be constructed out of these basic types. For example, a relation, which is very often used in B-models, is just a set of pairs. In the following, we want to represent the constants and variables of a machine by a vector of variables $V := \langle v_1, \ldots, v_n \rangle$. So, constants are simply regarded as variables that don't change their value.

We will also use some notation taken from [37]. Any B operation operating on the variables V with input values x and output values y has a normal form characterised by a predicate $P(x, V, V', y)$, where V' are the variables after the execution of the operation. The normal form assumes that an operation has only guards instead of predicates. Guards allow the execution of an operation only when they are satisfied. As a reminder for the reader, an operation with preconditions can always be executed, but there is no guaranteed behaviour of a machine if one of its operations is called outside its preconditions. Guarded operations are also called events. Any B operation of the form $X \longleftarrow op(Y)$ with input values x and output values y corresponds to an event denoted by $op.x.y$. Characterising a B operation as a predicate gives rise to a labelled transition relation on states. Two states s and s' of a machine M are related by an event $op.x.y$, when the predicate $P(a, s, s', b)$ holds. We then write $s \rightarrow^{M}_{op.x.y} s'$.

We continue now with the definition of a permutation function over a set of disjoint sets:

Definition 2.21 Let DS be a set of disjoint sets. A permutation f over DS is a total bijection from $\cup_{S \in DS} S$ to $\cup_{S \in DS} S$ such that $\forall\, S \in DS$ we have $\{f(s) \mid s \in S\} = S$.

The deferred sets in a B machine are disjoint, so we can use this definition to define a permutation function for a B machine.

Definition 2.22 Let M be a B machine with deferred sets $DS_1, \ldots, DS_k$ and enumerated sets $ES_1, \ldots, ES_m$. A function f is called a permutation for M if and only if it is a permutation over $DS_1, \ldots, DS_k$ and leaves any other basic datatypes, such as integers, Boolean values or values of the enumerated sets $ES_1, \ldots, ES_m$ fixed, i.e.

$$f(x) = x \quad \text{if} \quad x \in \mathbb{Z} \text{ or } x \in BOOL \text{ or } x \in ES_j \text{ for some } j \in \{1, \ldots, m\}.$$

We extend the definition of f by lifting f to pairs of values and sets of values as follows:

$$f(x \mapsto y) = f(x) \mapsto f(y)$$
$$f(\{x_1, \ldots, x_n\}) = \{f(x_1), \ldots, f(x_n)\}$$

We further lift f to state vectors by defining

$$f(\langle v_1, \ldots, v_k \rangle) = \langle f(v_1), \ldots, f(v_k) \rangle$$

Example 2.23 Let's consider the *Club* machine with its deferred set $Person = \{p_1, p_2, p_3\}$ and a permutation function f with $f = \{p_2 \mapsto p_3, p_3 \mapsto p_2\}$. Applying f to the state $s_5 = (member = \{p_1, p2\})$ we have:

$$f(\langle member \rangle) = \langle f(\{p_1, p_2\}) \rangle = \langle \{f(p_1), f(p_2)\} \rangle = \langle \{p_1, p_3\} \rangle.$$

We've now defined a permutation function, and we need to prove that it preserves the evaluation of any expression or predicate. Before we state the respective theorem, we need a few more notations. In the following, we represent a state by a substitution of variables with respective values. Let $V = \langle v_1, \ldots, v_n \rangle$ be the vector of variables and $C = \langle c_1, \ldots, c_n \rangle$ the vector of the respective values in some B expression or predicate, then we represent the state as a substitution of the form

$$[V := C] = [v_1, \ldots, v_n := c_1, \ldots, c_n].$$

The variables in V can be state variables, machine constants, quantified variables and local operation variables. To denote the value of an expression E in a state $[V := C]$, we write

$$E[V := C].$$

The type of this value is constructed from the sets of the machine. In a similar way we express the Boolean value of a predicate P in a state $[V := C]$ with

$$P[V := C].$$

The B-language has various set operators, but most of them are defined with more basic operators and/or set comprehension. For example, the subset operator can be written as

$$S \subseteq T \iff \forall x.(x \in S \Rightarrow x \in T).$$

Consequently, we can concentrate on the core expression and predicate syntax as it is defined in [1]. This core syntax is shown in Figure 2.9 and Figure 2.10.

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

Figure 2.9: Core syntax for expressions

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

Figure 2.10: Core syntax for predicates

Theorem 2.24 For any expression E, predicate P, state $[V := C]$ and permutation function f:

$$f(E[V := C]) = E[V := f(C)]$$
$$P[V := C] \Leftrightarrow P[V := f(C)]$$

Proof 2.25 The proof of the theorem can be done by induction over expression and predicate terms. The induction is mutual, since expressions may contain predicates, and vice versa. We want to present a few cases of the induction. Those cases not shown are proven in a similar way. First we consider the base case where E is an enumerated value ev:

$$\begin{array}{lll} & f(ev[V := C]) & \\ = & f(ev) & ev \textit{ has no free variables} \\ = & ev & f \textit{ leaves enumerated values fixed} \\ = & ev[V := f(C)] & ev \textit{ has no free variables} \end{array}$$

The case of a membership predicate makes use of the injectivity of f:

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

The case of set comprehension makes use of the mutual induction:

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

The case of a predicate with universal quantification

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

Corollary 2.26 From Theorem 2.24 we can conclude that every state permutation f for a B machine M satisfies

$$\begin{array}{l} \forall\, s \in S : s \models I \text{ if and only if } f(s) \models I \\ \forall\, s_1 \in S,\ \forall\, s_2 \in S : s_1 \rightarrow^{M}_{op.a.b} s_2 \ \Leftrightarrow\ f(s_1) \rightarrow^{M}_{op.f(a).f(b)} f(s_2), \end{array}$$

where S is the set of states, and I the invariant of M.

We explained so far what kind of symmetries we are looking for, and showed that reducing the state space using those symmetries is sound. We can now start to describe how we approach this. In the next section we want to introduce an essential part of our method to find symmetries, the graphical representation of B-states.

2.4 Viewing States as Graphs

Since deferred sets are used frequently in B models, we also have symmetries arising very frequently. The question, now, is: How do we detect those symmetries during model checking? We have developed a method to do so, in [56] and improved the idea in [51]. An essential point of our approach is the translation of individual states of a B machine into *state graphs*. This is what we are going to explain in this section.

Indeed, binary relations are at the heart of the B method, and are used to represent more complicated data structures (functions, sequences). Binary relations can be translated into directed coloured graphs in a natural way, thus translating the orbit problem of symmetry reduction into the Graph Isomorphism problem (see also [14, 45] or Section 14.4.1 of [13]). The idea does not sound so beneficial, because we already know from Section 2.2 that both problems have the same complexity. However, the graphs constructed from states are much smaller than a complete state graph of a model, so the computational effort is much lower in practice.

Any state consists of the models set of variables associated with certain values. Such values in B are either elements of basic sets (including Boolean values and

integers), sets of values, ordered pairs, relations or combinations of those data types. Let us first discuss the simple data types.

For a variable v whose value s_0 is an element of a set S, so $s_0 \in S$, we have the graph in Figure 2.11. The value of the variable is described by a vertex, and

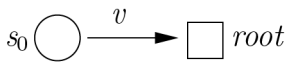


Figure 2.11: Graph for an atom

the variable itself is represented by an edge labelled with the name of the variable pointing to a special root vertex.

Is the value of the variable a set $\{s_0, \ldots, s_n\} \subseteq S$, then we have the graph as shown in Figure 2.12. Here, we have, for each element of the set $\{s_0, \ldots, s_n\}$, a

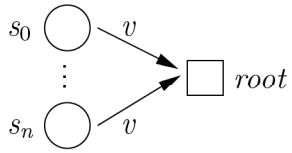


Figure 2.12: Graph for a set

vertex in the graphical representation. From each vertex there is an edge, labelled with the name of the variable, pointing to the *root* vertex. In both state graphs, the edges indicate the connection between the variable and its value in a particular state.

For a relation $v \in S \leftrightarrow T$, the graphical representation is slightly different, in that the root vertex is not truly needed. The graph for a state where the value of the variable v is a set of pairs $\{(s_0, t_0), \ldots, (s_n, t_m)\} \subseteq S \times T$ is depicted in Figure 2.13 below:

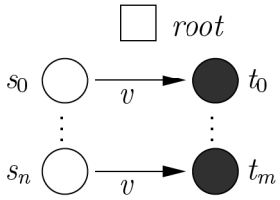


Figure 2.13: Graph for a relation

In this graph, the value of the relation, v, is represented by edges that indicate specific ordered pairs, whose edge labels denote the variable they encode. Note the colouring of the vertices: Each deferred set is assigned its own unique colour.

Furthermore, all elements of enumerated sets, as well as all other non-symmetric elementary datavalues (integers, booleans), get their own unique colour. An ordered pair can be regarded as a relation with a single pair, so there is not need for another graphical representation. Full details about the translation of basic data structures to graphs have been presented in [56].

For more complicated, nested data types, we have to introduce intermediary vertices into the graph - one for each nested value.

Example 2.27 Let v be a variable whose value is a set containing sets: $\{\{s_0\}, \{s_1, s_2\}\}$. The respective graph is depicted in Figure 2.14.

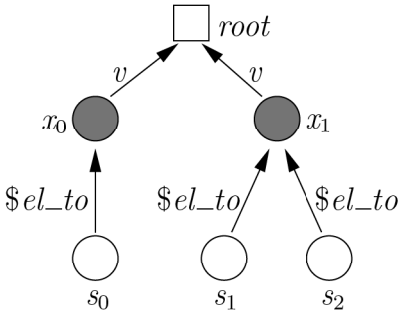


Figure 2.14: Graph for a set of sets

We give another example for a more complex data structure further below, but first we want to discuss the algorithm to construct those graphs. The basic idea is to compose individual graphs that represent each value of a variable in a state, to obtain the *state graph*. The algorithm to construct the state graphs has been presented in [55]. However, the current implementation of the algorithm presented there is slightly different, therefore we want to present here a refined version, which also corrects an old implementation in a few points. The algorithm takes a B-state as input and creates for each variable a subgraph representing its value. Unlike the presentation of the algorithm in [55], in the current implementation those subgraphs share vertices of atomic elements that occur as part of the data structure of several variables, see 2.15.

Example 2.28 Let's have a model with two deferred sets A, B and two variables S and T, where $S \subseteq A$ and $T \in A \leftrightarrow B$. Let $s = (S = \{x_1, x_2, x_3\}, R = \{(x_1, y_1)\})$ be a state of the model. The graphical representation of the state s would look as depicted in Figure 2.15. We can see that the vertex x_1 has been reused for representing the relation R.

Next, we want to describe the algorithm as it is implemented in ProB 1.3.1. There are likely to be some optimisations in future versions. The algorithm is separated into several sub-algorithms according to their function. The main algorithm

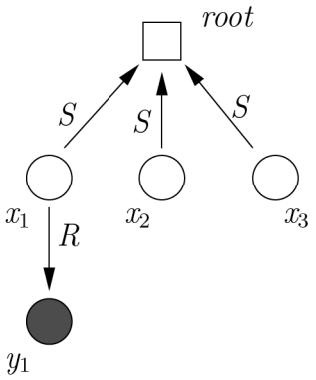


Figure 2.15: Graph for the state $s = (S = \{x_1, x_2, x_3\}, R = \{(x_1, y_1)\})$

state_graph calls *var_graph* with the parameter '*root*' for the parent vertex and a variable-value pair of the state. Then the algorithm *var_graph* calls the algorithm for creating an atom, a set or a relation depending on the data structure of the value. As mentioned earlier, all data structures in B can be constructed from atoms, sets or relations.

Algorithm 2.29[*state_graph(state)*]

Input: A State value *state*

1. // Setup vertex colours using global variables
2. **global** $dsets := \{DS_0, \ldots, DS_n\}$; // deferred sets used in the machine
3. **global** $esets := \{ES_0, \ldots, ES_m\}$; // enumerated sets used in the 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. **global** $element\_rep := \{\}$; // relation of elements to their vertex in the graph
   // representation
8. // Draw graph
9. **for all** variable-value pairs **such that** $\langle v, var \rangle$ in *state* **do**
   var_graph('*root*', v, *val*) ;
   end for

In order to remember the atomic elements (and their representative) that have been added to the graph already, we have introduced the variable *element_rep* to hold pairs consisting of atomic elements and their corresponding vertex in the graph. The denotation of the other global variables we took from [55]. The variables *dsets* and *esets* denote the sets of deferred- and enumerated sets in a model, respectively. Further we have an injective function *dcol*, giving each deferred set a unique colour. The variable *xcol* is given some unused colour to be reserved for vertices in X. The set of used colours is hold in the variable *used_colours*.

Algorithm 2.30[*assign_colour(val, vertex)*]

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

1. **if** val is empty set **then**
   assign *vertex* with colour $c \in Colours \setminus used\_col$;
2. **else if** $\exists\, S_p$ such that $S_p \in dsets \wedge val \in S_p$ **then**
   assign *vertex* with colour $dcol(p)$;
3. **else if** $\exists\, S_p$ such that $S_p \in esets \wedge val \in S_p$ **then**
   assign *vertex* with colour $c \in Colours \setminus used\_col$;
   $used\_col := used\_col \cup \{c\}$;
4. **else** // *val* is nested, so use colour for vertices in X
   assign *vertex* with colour *xcol*;
5. **end if**

The algorithm $assign\_colour(val, vertex)$ takes a B-value and the corresponding vertex in the graphical representation as input, and assigns the vertex the respective colour. If the B-value is an element of a deferred set, then its vertex gets the colour of that deferred set. In case of an element of an enumerated set, the vertex gets a new colour that hasn't been used yet. For nested values the predefined colour *xcol* is used.

Algorithm 2.31[$var\_graph(V_{parent}, v, val)$]

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

1. **if** *val* is a set **then**
   $set(V_{parent}, v, val)$;
2. **else if** *val* is an atom **then**
   $atom(V_{parent}, v, val)$;
3. **else if** *val* is a relation **then**
   $relation(V_{parent}, v, val)$;

4. **else**
   // *val* is a pair
   $val := \{val\}$; // new value has same graph
   $relation(V_{parent}, v, val)$
5. **end if**

We want to describe next, the implementation of the sub-algorithms that construct the graphical representation of atoms, sets and relations. Each algorithm takes the parent vertex *parent*, a variable v and its value *val* as input parameters. In case of an atom, such as $v = val_0$ the algorithm is fairly simple. Depending on whether the atomic element val_0 has been represented before in the graph, i.e. $val_0 \in dom(element\_rep)$, during the construction of some other data structure, the algorithm either draws an edge from an existing vertex to the parent vertex, or creates a new vertex V_{val_0}, assigns it a colour and draws an edge from V_{val_0} to the vertex *parent*, see also Figure 2.11.

Algorithm 2.32[$atom(V_{parent}, v, val)$]

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

1. **if** val_0 is in the domain of *element_rep* **then**
   Draw edge, $element\_rep(val_0) \overset{v}{\mapsto} V_{parent}$;
2. **else**
   Create vertex V_{val_0};
   $assign\_colour(val_0, V_{val_0})$;
   Draw edge, $V_{val_0} \overset{v}{\mapsto} V_{parent}$;
3. **end if**

For creating a set, the algorithm *set* does consider several cases. A special case is when the set is empty and the parent is *root*. This case often occurs when a set variable is initialised. Step 1 of the algorithm treats that case, by drawing an edge, labelled with the name of the variable, from *root* back to *root*. Note, that step 1 handles only the case when the parent vertex of an empty set is the *root* vertex. When the empty set is part of a nested data structure, it is handled like an atom within such a structure in step 3. Step 2 to 6 take each element val_i of *val* and create a new vertex V_{val_i} for that element, unless it already has a representation. A newly created vertex gets a colour by calling the algorithm *assign_colour*. If the parent vertex of V_{val_i} is the root vertex, then an edge, labelled with the name of the variable v, is drawn from V_{val_i} to *root*. Is *root* not the parent vertex of V_{val_i} then the edge from V_{val_i} to its parent vertex gets the label $\$to\_el$. This special label is only used when the value of the variable v is a nested data structure and the algorithm *set* has been called recursively to construct that nested data structure. When an

element val_i of val is not an atom or the empty set, then the algorithm *var_graph* is called recursively in step 5, with an V_{val_i} as parent vertex, the variable v, and the set element val_i as its value.

Algorithm 2.33[*set*(V_{parent}, v, val)]

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

1. **if** val is empty and *root* is parent **then**
   Draw edge, $root \overset{v}{\mapsto} root$; // self loop
2. **else**
   for all $0 \leq i \leq n$ **do**
3. **if** val_i is not in the domain of *element_rep* **then**
   Create vertex V_{val_i}, such that if V_{val_i} is not an atom or the empty set, then $V_{val_i} \in X$;
   assign_colour(val_i, V_{val_i});
   add pair (val_i, V_{val_i}) to *element_rep*
   else
   Set $V_{val_i} = element\_rep(val_i)$;
   end if
4. **if** $V_{parent} = root$ **then**
   Draw edge, $V_{val_i} \overset{v}{\mapsto} V_{parent}$;
   else // different edge label for nested data structure
   Draw edge, $V_{val_i} \overset{\$to\_el}{\longmapsto} V_{parent}$;
   end if
5. **if** val_i is not an atom or the empty set **then**
   var_graph(V_{val_i}, v, val_i);
   end if
6. **end for**
7. **end if**

When creating the graphical representation for a relation, there are again several cases to consider: The case when the relation is empty is treated the same way as we discussed for an empty set. Now, each element of a pair in a relation can either be an atom or not, and if it is an atom, it might have been represented in the graph through another data structure already. The algorithm *relation* starts with a loop going through all pairs (val_i, val_j) of the relation. Steps 3 and 4 decide independent from each other, if there needs to be a vertex created for the elements val_i and val_j respectively, or if an existing vertex can be taken for the remaining

steps. The next step draws an edge from the vertex representing val_i to the vertex representing val_j, if the parent vertex is the *root* vertex. If the parent vertex is not *root*, then there are edges constructed from the vertices val_i and val_j to the parent vertex *parent*. The edge from val_i to the parent vertex has the label $\$from$ and the edge from val_j to the parent vertex has the label $\$to\_el$. The different labels are used to distinguish between the first and second element of a pair. This construction is necessary if a relation is part of a nested data structure, to avoid the occurrence of false symmetries. Note that we have also used the label $\$to\_el$ for sets earlier. We did this to reduce the number of different edge labels. When we explain the translation from state graphs to vertex-coloured graphs in Section 3.4, we will see why reducing the number of edge labels is an advantage. Also, note that for a relation the *root* vertex is not used. Similar as for drawing a set, for any element of a pair of the relation that is not an atom or the empty set, the algorithm *var_graph* is called recursively for that element.

Algorithm 2.34[*relation*(V_{parent}, v, val)]

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

1. **if** *val* is empty **then**
   if empty set is not in the domain of *element_rep* **then**
   Create vertex $V_{val_{\{\}}}$
   assign_colour(val, $V_{val_{\{\}}}$);
   add pair (val_i, $V_{val_{\{\}}}$) to *element_rep*
   Draw edge, $V_{val_{\{\}}} \overset{v}{\mapsto} V_{parent}$;
   else
   Draw edge, *element_rep*($\{\}$) $\overset{v}{\mapsto}$ V_{parent};
   end if
   end if

2. **for all** $(val_i, val_j) \in val$ **do**

3. **if** val_i is not in the domain of *element_rep* **then**
   Create vertex V_{val_i}, such that if V_{val_i} is not an atom or the empty set, then $V_{val_i} \in X$;
   assign_colour(val_i, V_{val_i});
   add pair (val_i, V_{val_i}) to *element_rep*
   else
   Set V_{val_i} = *element_rep*(val_i);
   end if

4. **if** val_j is not in the domain of *element_rep* **then**
   Create vertex V_{val_j}, such that if V_{val_i} is not an atom or the empty set,

then $V_{val_i} \in X$;
$assign\_colour(val_j, V_{val_j})$;
add pair (val_j, V_{val_j}) to $element\_rep$
else
Set $V_{val_j} = element\ \ rep(val_j)$;
end if

5. **if** $V_{parent} = root$ **then**
Draw edge, $V_{val_i} \overset{v}{\mapsto} V_{val_j}$;
else // V_{parent} is not $'root'$
Draw edges, $V_{val_i} \overset{\$from}{\longmapsto} V_{parent}$, $V_{val_j} \overset{\$el\_to}{\longmapsto} V_{parent}$;
end if

6. **if** val_i is not an atom or the empty set **then**
$var\_graph(V_{val_i}, v, val_i)$;
end if

7. **if** val_j is not an atom or the empty set **then**
$var\_graph(V_{val_j}, v, val_j)$;
end if

8. **end for**

The following Theorem, taken from [55], states the correctness of the algorithm $state\_graph(state)$ presented there. For a proof refer to [55]. We have the algorithm refined here in favour of clarity, but the correctness proof still applies to concept. In order to proof the correctness of the refinement and therefore the implementation used by ProB we intend to formal specify the algorithm in B in future work.

Theorem 2.35 For any two states, s and s', s is symmetric to s' if and only if $state\_graph(s)$ is isomorphic to $state\_graph(s')$.

Note that the algorithm to create a graph from a state is just to have a working concept. There might be other representations that are easier to create, or construct smaller graphs. We now want to go through the algorithm with an example state, and construct stepwise the respective graph.

Example 2.36 Let's take a model M with two deferred sets A and B, and variables $Q \in \mathbb{P}(A)$ and $R \in A \leftrightarrow B$. Let s be a valid state of M, where

$$s = (Q = \{\{\}, \{a_1\}, \{a_2, a_4\}\}, R = \{(a_1, b_1), (a_2, b_2)\})$$

First, the algorithm $state\_graph(s)$ with state s as the parameter is called. This assigns a colour the deferred sets A and B, and initialises the remaining global variables accordingly. Then the algorithm calls the sub-algorithm var_graph for

each variable-value pair of the state and *root* as the parent vertex. Let's start with the variable-value pair $(v, val) = (Q, \{\{\}, \{a_1\}, \{a_2, a_4\}\})$, then the function call is

$$var\_graph(`root`, Q, \{\{\}, \{a_1\}, \{a_2, a_4\}\});$$

Since Q is a set, the algorithm calls the function

$$set(`root`, Q, \{\{\}, \{a_1\}, \{a_2, a_4\}\});$$

The set Q is not empty, and there have been no vertices created in the graph yet, except for *root*, so the algorithm creates a vertex for the first element in Q. The first element $val_0 = \{\}$ of Q is the empty set, therefore the algorithm creates a special vertex for the empty set and assigns it an unused colour. Then an edge labelled with Q is drawn from the new vertex to the parent vertex *root*, see Figure 2.16.

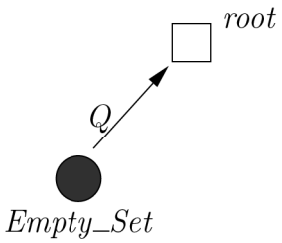


Figure 2.16: Graph for $Q = \{\{\}\}$

The next element $val_1 = \{a_1\}$ is constructed as follows. Since $\{a_1\}$ is not the empty set or an atom, an intermediary vertex V_{val_1} is created, coloured with the reserved colour *xcol*. Step 3 of the *set* algorithm draws an edge labelled with Q from that vertex to its parent vertex *root*. The last step then calls *var_graph* recursively:

$$var\_graph(V_{val_1}, val_1, \{a_1\}\});$$

Since val_1 is a set, the algorithm calls the function

$$set(V_{val_1}, val_1, \{a_1\}\});$$

Now we have to construct the graph for a simple set. For each element in that set, here only a_1, there is a vertex created, coloured with the colour assigned to the deferred set A. The edge going from the vertex representing x_1 to the parent vertex V_{val_1} gets the label $\$to\_el$, since the parent vertex is not the root node. Similar steps are repeated for the last element $val_2 = \{a_2, a_4\}$ in Q. The graphical representation for the set Q is depicted in Figure 2.17

The algorithm continues with constructing the graphical representation of the relation R and calls

$$var\_graph('root', R = \{(a_1, b_1), (a_2, b_2)\}\});$$

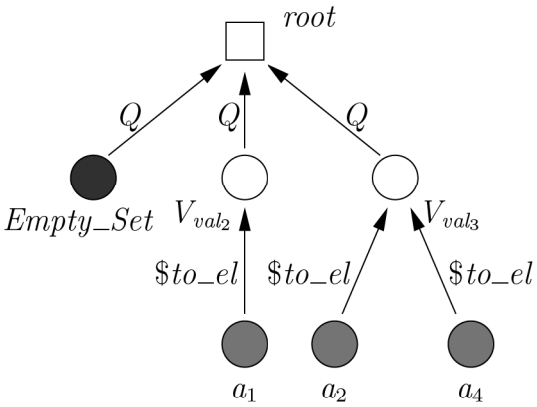


Figure 2.17: Graph for $Q = \{\{\}, \{a_1\}, \{a_2, a_4\}\}$

Since R is a relation, the algorithm now calls

$$relation('root', R = \{(a_1, b_1), (a_2, b_2)\}\});$$

Starting with the pair a_1, b_1, the algorithm first decides, if the atoms a_1 and b_1 have been represented in the graph before or not. Since a_1 already has been created, there is no new vertex needed for that element. For the element b_1 though, there is a new vertex created, which is assigned the colour of the deferred set B. Now the algorithm draws an edge labelled with R, from the vertex representing a_1 to the new vertex representing b_1. The same steps are done for the second pair (a_2, b_2), and we finally obtain the graphical representation for the state s as depicted in Figure 2.18.

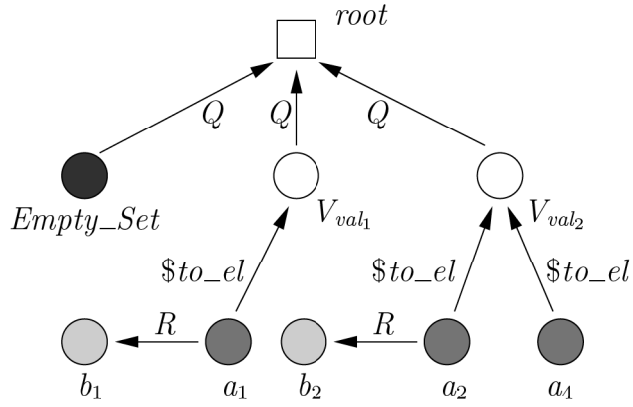


Figure 2.18: Graph for $s = (Q = \{\{\}, \{a_1\}, \{a_2, a_4\}\}, R = \{(a_1, b_1), (a_2, b_2)\})$

The algorithm translates states into graphs, such that symmetric states are represented by isomorphic graphs, and states that are not symmetric are represented

by non-isomorphic graphs.

We now want to show intuitively how symmetric states can be detected through their graphical representation. Let's go back to the example *club model* to make this concept more clear. This machine has a variable *member* that contains elements of the deferred set *Person*. We compare the graphical representation of the symmetric states

$$s1 = (member = \{p1, p2\}) \quad and$$
$$s2 = (member = \{p1, p3\}).$$

In both cases, we have to graphically represent a simple set with two elements. The respective graphs for the states $s1$ and $s2$ are depicted in Figure 2.19.

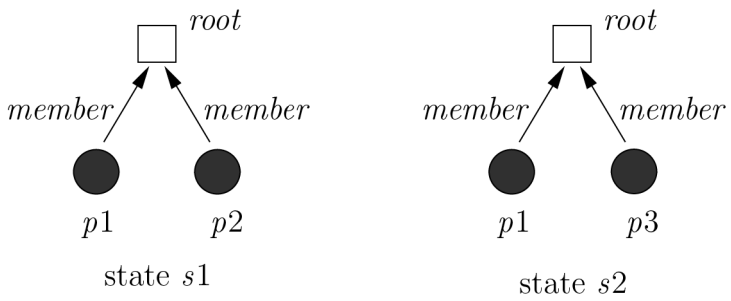


Figure 2.19: Graphical representation of two symmetric states

The labellings on the vertices are only for clarity in the picture, but what matters is the colouring of the vertices here. Since the elements of the set-variable *member* are elements of a deferred set, the respective vertices all get the same colour. We can see that the two graphs are isomorphic. Indeed exchanging the deferred set element $p2$ with $p3$ gives an isomorphism between the states $s1$ and $s2$. In this example, we have only one type of label on the edges, so it does not affect the isomorphism relation. Generally, we have to consider different labels on edges as differently coloured edges in the graph. We will come back to this later in Section 3.4.

Let's now take two non-symmetric states and compare their graphical representations. We have

$$s1 = (member = \{p1, p2\}) \quad and$$
$$s3 = (member = \{p1, p2, p3\}).$$

with their graphical representations in Figure 2.20.

These two graphs are clearly not isomorphic, since the graph to state $s3$ has an edge and a vertex more than the graph to state $s1$.

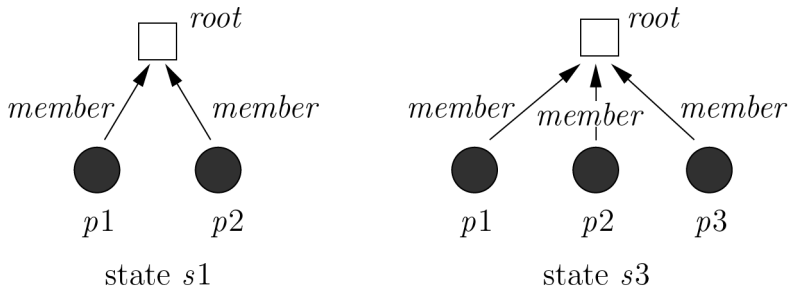


Figure 2.20: Graphical representation of two non-symmetric states

Chapter 3

Using NAUTY to detect Symmetry for B

We've seen so far how states can be translated into state graphs. In this chapter we will introduce NAUTY, which we use to calculate a so-called canonical form for any state graph. We then show how the model checking algorithm is modified to make use of that information. Finally, we will describe how the tools ProB and NAUTY work together in practice.

3.1 The NAUTY Toolset

We want to give an introduction into the NAUTY toolset, explain what it is, what it can do and how we are going to use it. For further details see the NAUTY *User's Guide* [42]. For the mathematical terminology used in this section review Section 2.2. The name *nauty* is short for the phrase ***n**o **aut**omorphisms, **y**es?* This phrase already gives an idea what NAUTY has been developed for. NAUTYs developer Brendan D. McKay wrote a tool for determining the automorphism group of a vertex-coloured graph. It computes the size of the automorphism group, a set of generators and the orbits of that group. However, we are making use of a powerful side feature of NAUTY, which is its ability to test graphs for isomorphism by calculating a canonical form. The idea, in order to decide if two graphs are isomorphic, is to calculate an isomorph graph, called canonical form, for each of them, and see if those are the same. The respective isomorphic graphs are represented in a way so that this is easy to decide. This representation is called a certificate, which is unique for each isomorphism class of graphs. We will explain how the canonical form and its respective certificate are calculated in Section 3.3.

Now we want to concentrate on how NAUTY works with regard to our interface, and how it represents graphs. NAUTY comes with numerous options to modify its behaviour and performance. Most of them we left at their default values, which is the safest way to work with NAUTY. One of the options we really do need to set is the Boolean variable *digraph*, which tells NAUTY that it has to handle directed

graphs when it is set. NAUTY also uses its own data structures in order to represent graphs. The first is called *setword* and just an unsigned integer with either 16, 32 or 64 bits. The size of *setword* depends on the parameter *WORDSIZE*. This parameter is set at the compile time of NAUTY either manually as a command-line parameter or automatically to the size of a *long int*. Usually, the value for *WORDSIZE* will be 32. The next data structure is called *set*, and is used to represent a subset of a graphs vertices $V = \{0, 1, \ldots, n-1\}$, where n is the number of vertices in the graph. A *set* is written as an array of *setwords*, and the bits are numbered from 0 to $n-1$. Unnumbered bits always have the value 0. Any numbered bit $i \in 0, \ldots, n-1$ is set to 1, if and only if vertex i is element of that set.

Example 3.1 Let's say we have a graph with 40 vertices, and the predefined size of a setword is 32. We want to describe the set of vertices adjacent to vertex $v = 0$, which is, say, $\{2, 33, 39\}$. In order to represent all 40 vertices, we need two *setwords* arranged as array to build the *set*.

$$\begin{array}{cccccccccccccccccc}
[& 0 & 0 & 1 & 0 & \cdots & 0 & | & 1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 &] \\
 & 0 & 1 & 2 & 3 & \cdots & 32 & | & 33 & 34 & \cdots & 38 & 39 & 40 & & &
\end{array}$$

first setword — second setword

Generally, the smallest number m of *setwords* in the array is chosen such that

$$WORDSIZE * m \geq n.$$

Indeed in the example we have $32 * 2 \geq 40$.

NAUTY has two different ways of representing graphs: The first is called *sparse form* and uses the type *sparsegraph*, and the second is called *packed form* and uses the type *graph*. We decided to use the packed form for representing graphs, because its generally more space efficient for our state graphs. The space needed for the packed form depends only on the number of vertices in a graph, whereas for the sparse form it also depends on the number of edges. The sparse representation is more efficient for large sparse graphs, which we usually don't encounter, see Appendix C. Another advantage of the packed form is that the number of bytes used to store a graph does not depend on the size of a long integer. If the size of a long integer is larger, then there are simply less setwords needed to represent a graph. This pays off for architectures that use 64 bits for representing a long integer, instead of 32 bits.

We want to explain just the packed form here. This representation is quite intuitive, when we consider a graphs adjacency matrix. Each row in an adjacency matrix can be written as an array of m *setwords* building one set. The whole matrix can then be stored in n *sets* or equivalent $m * n$ *setwords*. Let's take a graph with 40 vertices and its adjacency matrix given as:

$$\left|\begin{array}{cccc}
a_{1,1} & a_{1,2} & \cdots & a_{1,40} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,40} \\
\cdots & \cdots & \cdots & \cdots \\
a_{40,1} & a_{40,2} & \cdots & a_{40,40}
\end{array}\right|$$

The information of that graph is stored in the following array of setwords, where setwords are separated by vertical bars for visibility:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,32} & | & a_{1,33} & \cdots & a_{1,40} & 0 & \cdots & 0 & | \\ a_{2,1} & a_{2,2} & \cdots & a_{2,32} & | & a_{2,33} & \cdots & a_{2,40} & 0 & \cdots & 0 & | \\ \cdots & \cdots & \cdots & \cdots & \vdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\ a_{40,1} & a_{40,2} & \cdots & a_{40,32} & | & a_{40,33} & \cdots & a_{40,40} & 0 & \cdots & 0 &] \end{bmatrix}$$

As we can see, the above matrix is stored in $2 * 40 = 80$ setwords.

3.2 Notations from Graph Theory

Before we explain in detail how the canonical form is calculated, we need a few more notations and definitions, which we take from McKay's article *Practical graph isomorphism* [43].

If not stated otherwise, V denotes the set $\{0, 1, 2, \ldots, n\}$. We denote with $\mathcal{G}(V)$ the set of vertex-coloured directed graphs with vertex set V.

Definition 3.2 Le $G \in \mathcal{G}(V)$. A *partition* of the set V is a set of disjoint non-empty subsets of V whose union is v. An *ordered partition* of V is a sequence $[V_1, V_2, ..., V_r]$, such that $\{V_1, V_2, ..., V_r\}$ is a partition of V.

If not stated otherwise in the following, a partition is always an ordered partition. The set of (ordered) partitions of V will be denoted by $\Pi(V)$. The elements of a partition $\pi \in \Pi(V)$ are called its *cells*. A cell of cardinality one, is a *trivial* cell. If every cell of a partition π is trivial, then π is said to be a *discrete* partition, while if there is only one cell, π is the *unit* partition.

Definition 3.3 Let $\pi, \psi \in \Pi(V)$ be two ordered partitions. We say, ψ is *finer* than π if:

1. every cell $\psi[i]$ of ψ is subset of some cell $\pi[i]$ of π, and
2. if $u \in \pi[i_1]$ and $v \in \pi[j_1]$ with $i_1 \leq j_1$, then $u \in \psi[i_2]$ and $v \in \psi[j_2]$ for some i_2, j_2 with $i_2 \leq j_2$.

We write $\psi \leq \pi$. Under the same conditions, we say that π is *coarser* that ψ.

Example 3.4 Let $G = (V, E)$ be a graph with the vertex set $V = \{0, 1, 2, 3, 4\}$. The partition

$$\psi = [\{0\}, \{2\}, \{1, 3, 4\}]$$

is finer than

$$\pi = [\{0\}, \{1, 2, 3, 4\}],$$

but the partition $[\{2\}, \{1, 3, 4\}, \{0\}]$ is not finer than π, because the blocks are out of order, with respect to π.

The set $\Pi(V)$ forms a lattice under the partial order $\leq$. It follows that for any two partitions $\pi_1, \pi_2 \in \Pi(V)$, there is a unique coarsest partition $\bar{\pi}$ which is finer than π_1 and π_2, and a unique finest partition $\tilde{\pi}$ which is coarser than π_1 and π_2. This fact is needed for the correctness of the algorithm calculating the canonical form, see [43].

Definition 3.5 Let $\pi_1, \pi_2 \in \Pi(V)$ be two partitions, then the coarsest partition $\bar{\pi}$, which is finer than π_1 and π_2, is denoted as $\pi_1 \wedge \pi_2$, and the finest partition $\tilde{\pi}$, which is coarser than π_1 and π_2, is denoted as $\pi_1 \vee \pi_2$.

We will need the following definition for *in-* and *out-degree* of a vertex v, with respect to a subset of V:

Definition 3.6 For $G \in \mathcal{G}(V)$, $v \in V$ and $W \subseteq V$, we define the *in-degree* of v, denoted as $d_{in}(v, W)$, to be the number of vertices in W, which have an edge going to v, and the *out-degree*, denoted by $d_{out}(v, W)$, to be the number of vertices in W, which have an edge going from v to it.

Example 3.7 Let's take the graph G with $V = \{0, 1, 2, 3\}$ in Figure 3.1 as an example. Let $W = \{0, 2\}$ be a subset of V and $v = 1$ then $d_{out}(v, W) = 2$ and

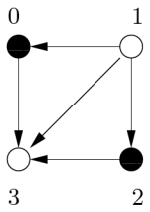


Figure 3.1: A vertex-coloured graph

$d_{in}(v, W) = 0$, since there is one edge going from vertex 1 to vertex 0 and vertex 2 each, but there are no edges from an element in W to v. Looking at $v = 3$, gives us $d_{out}(v, W) = 0$ and $d_{in}(v, W) = 2$, since there is no edge going from v to an element in W, but two edges going from elements in W to v.

For the algorithm that calculates the canonical form, we need the following definition:

Definition 3.8 Let $G \in \mathcal{G}(V)$. A partition $\pi \in \Pi(V)$ is *equitable* with respect to G if, for all cells $W_1, W_2 \in \pi$ and for all $v_1, v_2 \in W_1$ we have

$$d_{in}(v_1, W_2) = d_{in}(v_2, W_2) \text{ and } d_{out}(v_1, W_2) = d_{out}(v_2, W_2).$$

This definition says that all vertices in any given cell of an equitable partition π cannot be distinguished by the in- or out-degree with respect to any cell in π. On the other hand, if a partition is not equitable, then it contains a cell whose vertices can be distinguished in that fashion. The algorithm, which we will describe later in this chapter, uses this distinction to break up the cells of a partition into smaller cells. By sorting the cells in a certain way the algorithm defines stepwise a unique ordering on the vertices. This ordering defines an automorphism of the respective graph, such that by applying this automorphism to the graph we obtain its canonical form. The canonical form is a unique representative of an isomorphism class of graphs. Before we get to the details of the algorithm, we want to give an example of an equitable partition in order to make the concept more clear.

Example 3.9 We take again the graph G from Example 3.7. Let $\pi_1 = [\{0,2\}\{1,3\}]$ be a partition with the cells $W_1 = \{0,2\}$ and $W_2 = \{1,3\}$ containing the black and white vertices, respectively. If π_1 is equitable, then we would have the following cases

$$\begin{aligned} &d_{in}(v, W_1) \quad \text{is the same for all vertices} \quad v \in W_1 \\ &d_{in}(v, W_1) \quad \text{is the same for all vertices} \quad v \in W_2 \\ &d_{in}(v, W_2) \quad \text{is the same for all vertices} \quad v \in W_1 \\ &d_{in}(v, W_2) \quad \text{is the same for all vertices} \quad v \in W_2 \end{aligned}$$

for d_{in} and respectively for d_{out}. We have $d_{in}(0, W_1) = d_{in}(2, W_1) = 0$ for the first case, but $d_{in}(1, W_1) = 0 \neq 2 = d_{in}(3, W_1)$ for the second case, and therefore we already know that π_1 is not equitable, without looking at the other cases.

Let's take as another example $\pi_2 = [\{0,2\},\{1\},\{3\}]$ as a partition of V. Let $W_1 = \{0,2\}$, $W_2 = \{1\}$ and $W_3 = \{3\}$, We consider the non-trivial cell $W_1 = \{0,2\}$ first, and test for the property in Definition 3.8 for all cells in π_2, including W_1 itself. We have:

$$\begin{aligned} d_{in}(0, W_1) &= d_{in}(2, W_1) = 0 \\ d_{in}(0, W_2) &= d_{in}(2, W_2) = 1 \\ d_{in}(0, W_3) &= d_{in}(2, W_3) = 0 \end{aligned}$$

and

$$\begin{aligned} d_{out}(0, W_1) &= d_{out}(2, W_1) = 0 \\ d_{out}(0, W_2) &= d_{out}(2, W_2) = 0 \\ d_{out}(0, W_3) &= d_{out}(2, W_3) = 1. \end{aligned}$$

So for all V_i, $i \in \{1,2,3\}$ the property in Definition 3.8 holds. For trivial cells, like $V_2 = \{1\}$, the property in Definition 3.8 is obviously true for each cell $V_i \in \pi_1$. Therefore the partition π_2 is equitable.

In the next section, we want to describe the algorithm for calculating the canonical form, and its mathematical background.

3.3 The Canonical Form

The graph isomorphism problem is the problem of determining whether two finite graphs are isomorphic. It is not known if this problem is solvable in polynomial time, or if it belongs to the NP-complete class of problems. There are many graph classes with a polynomial-time algorithm to solve the graph isomorphism problem, for example, trees [3, 33] and planar graphs [27]. This suggests that the graph isomorphism problem might be polynomial in general. However, so far, Babai and Luks have developed the best algorithm, which has a runtime of $2^{O(\sqrt{(n \ log\, n)})}$ for general graphs with n vertices [31]. In practice, though, an algorithm developed by Brendan D. McKay, which he described in his article *Practical Graph Isomorphism* [43] is widely used, and has been implemented in the NAUTY package [42].

The goal of a graph isomorphism algorithm is to classify graphs such that two graphs belong to the same class, if and only if the two graphs are isomorphic. It is easy to find properties that two isomorphic graphs need to have. For example, two isomorphic graphs must obviously have the same number of vertices. The challenging part is to find properties that are sufficient to determine if two graphs are isomorphic, that are also easy to compute with computer aid.

McKay's graph isomorphism algorithm computes a canonical form to determine isomorphisms. As mentioned earlier, this canonical form is a unique representative of an isomorphism class of graphs. That means, by computing the canonical form for two graphs, say G_1 and G_2, respectively, we can decide whether G_1 and G_2 belong to the same isomorphism class, i.e. they are isomorphic or not.

The algorithm by McKay therefore implements a function that calculates to a given vertex-coloured graph its canonical form. We want to give a formal definition of such a function, which is called a canonical label.

Definition 3.10 A *canonical label* is a map $C : \mathcal{G}(V) \times \Pi(V) \to \mathcal{G}(V)$, such that for any $G \in \mathcal{G}(V)$, $\pi \in \Pi(V)$ and $\gamma \in S_n$ we have:

$(C1)$ $C(G, \pi) \cong G$

$(C2)$ $C(G^\gamma, \pi^\gamma) = C(G, \pi)$

$(C3)$ If $C(G, \pi^\gamma) = C(G, \pi)$, then $\pi^\gamma = \pi^\delta$ for some $\delta \in Aut(G)$.

This definition takes some partition π of the vertices V into account. The partition π corresponds to the vertex-colouring of the graph. That means, vertices are in the same cell of π, if and only if they have the same colour. The following Theorem states that we can use the definition of a canonical label to solve the graph isomorphism problem on vertex-coloured graphs.

Theorem 3.11 Let $G_1, G_2 \in \mathcal{G}(V)$, $\pi \in \Pi(V)$ and $\gamma \in S_n$. Then it is

$$C(G_1, \pi) = C(G_2, \pi^\gamma)$$

if and only if there is a permutation $\delta \in S_n$ such that $G_2 = G_1^\delta$ and $\pi^\gamma = \pi^\delta$.

For a proof of this Theorem, see [43].

A very simple way to define a canonical label is to take a function that reorders the vertices of a graph, such that reading the adjacency matrix row by row[1] gives the smallest number. For vertex-coloured graphs, we rearrange only vertices within cells. The resulting binary number would be a certificate for an isomorphism class of graphs. This certificate though is not very practical, because the number of possible vertex-orderings grows exponentially with the number of vertices in a graph. McKay has developed an algorithm that uses the structural properties of a graph to calculate a canonical form. Because of this, his algorithm has to consider only a subset of the vertex-orderings.

In the following, we describe a simplified version of the algorithm by McKay, see also [34]. We call it the *canonical labelling algorithm*, since the algorithm defines a relabelling on the vertices to obtain the canonical form. For more details and proofs of correctness, refer to [43]. First we want to look at the part of the algorithm that uses the structure of the graph to reorder the vertices. Given a graph G and an initial partition π, the procedure tries to distinguish vertices of one cell by the number of ingoing/outgoing edges from/to another cell. In other words, the procedure calculates an equitable partition, which is finer than π. The algorithm for this procedure we call *refine* in the following.

The *refine*-algorithm takes a graph $G \in \mathcal{G}(V)$, a partition $\pi \in \Pi(V)$ and returns the unique coarsest equitable partition $\xi(\pi)$ finer than π:

Algorithm 3.12 *refine*(G, π)
Input: $G \in \mathcal{G}(V)$, $\pi \in \Pi(V)$
Output: $\xi(\pi)$

1. Set ξ equal to π
2. Let $\mathcal{S}$ be a list containing the cells of ξ.
3. **while** $\mathcal{S} \neq \emptyset$ **do**
4. remove a cell T from the list $\mathcal{S}$;
5. **for all** cells $\xi[i]$ of ξ do
6. create a list L and set $L[j] = \{v \in \xi[i] : d_{in}(v, T) = j\}$ for each j;
7. **if** there is more than one non-empty set in L, **then**
8. replace the cell $\xi[i]$ with the non-empty sets in L, in the order of the index $j, j = 0, 1, ..., n-1$;
9. add the non-empty sets on L to the end of the list $\mathcal{S}$.

[1] reading column by column is also feasible

10. **end if**
11. **end for**
12. **for all** cells $\xi[i]$ of ξ do
13. create a list L and set $L[j] = \{v \in \xi[i] : d_{out}(v, T) = j\}$ for each j;
14. repeat steps 7 to 10;
15. **end for**
16. **end while**
17. Return ξ.

We want to go through the algorithm with an example.

Example 3.13 Let's pick the graph $G = (V, E)$ with $V = \{0, 1, 2, 3\}$ and the partition $\pi = [\{0, 2\}\{1, 3\}]$, such that it appears as depicted in Figure 3.2.

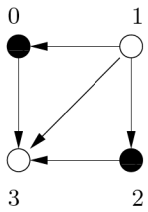


Figure 3.2: Graph with initial partition $\pi = [\{0, 2\}\{1, 3\}]$

1. The first step of the algorithm sets $\xi = \pi$.
2. The set $\mathcal{S}$ is initialised with the cells of ξ: $\mathcal{S} = [\{0, 2\}\{1, 3\}]$.
3. The first cell $T = \{0, 2\}$ is taken from $\mathcal{S}$. For the first element $\xi[0] = \{0, 2\}$ in ξ, a list L, as in step 6, is created. The list L contains for each index $j, j \in 0, ..., 3$ the set of vertices in $\xi[0]$, which have exactly j edges going from any vertex in T to them. Since for both vertices in $\xi[0]$ there are no edges from any vertex in T to either of them, the list L looks as follows:

   $$L = [\{0, 2\}, \{\}, \{\}, \{\}].$$

   There is only one non-empty set in the list, so the algorithm goes directly to the next element $\xi[1] = \{1, 3\}$ in ξ. Here we have $d_{in}(1, T) = 0$ and $d_{in}(3, T) = 2$, so

   $$L = [\{1\}, \{\}, \{3\}, \{\}].$$

Now there are two non-empty sets in L, and therefore the cell $\{1,3\}$ in ξ is replaced with the two non-empty sets in the order they occur in L. The cells $\{1\}$ and $\{3\}$ are also added to the end of the list $\mathcal{S}$. We have

$$\xi = [\{0,2\},\{3\},\{1\}] \text{ and}$$
$$\mathcal{S} = [\{1,3\},\{1\},\{3\}]$$

The same procedure is now repeated with the new ξ, while the list created in step 13 contains now for each index $j, j \in 0,...,3$ the set of vertices in $\xi[i]$, which have exactly j edges going to any vertex in T. Since there are no cells split up in this step, the algorithm continues with the next element in $\mathcal{S}$.

4. The next element in $\mathcal{S}$ is $T = \{1,3\}$. For the first element $\xi[0] = \{0,2\}$ in ξ we get in step 6

   $$L = [\{\},\{0,2\},\{\},\{\}].$$

   So the cell $\xi[0]$ is not split up further by T. The two remaining cells in ξ are already trivial cells and cannot be split up further. In the algorithm, the list L has only one non-empty element for these cells. Therefore, ξ and the set $\mathcal{S}$ are not changed in steps 5 to 11. Steps 12 to 15 give no changes again.

5. Continuing with the last two elements in $\mathcal{S}$ gives no further splits, and the algorithm finishes with the output of the equitable partition

   $$\xi(\pi) = [\{0,2\},\{3\},\{1\}].$$

After refining a partition once, we will usually not obtain a discrete partition. That means the vertex-ordering is not uniquely determined yet. The canonical labelling algorithm now takes the first non-discrete cell c of the refined partition and splits it into two cells. One cell containing a single vertex and the other the remaining vertices. The resulting partition is then refined again. This procedure is done for all vertices in c.

Given a graph G with a partition π, we will use the notation $Num_\pi(G)$ to denote the binary number obtained by reading the adjacency matrix of G row by row, and we write $Cert(G)$ for the minimum number returned by the canonical labelling algorithm. We present the canonical labelling algorithm below:

Algorithm 3.14 *label*(G,π)
Input: $G \in \mathcal{G}(V)$, $\pi \in \Pi(V)$
Output: $Cert(G)$

1. Refine π to an equitable partition: $\pi_{eq} = refine(\pi)$.

2. **if** π_{eq} is discrete, **then**

3. compute $Num_{\pi_{eq}}(G)$

4. **if** $Num_{\pi_{eq}}(G)$ is smallest number so far, **then**

   $Cert(G) = Num_{\pi_{eq}}(G)$

   end if

5. **else**

6. take first non-discrete cell V_j in $\pi_{eq} = [V_1, \ldots, V_{j-1}, V_j, V_{j+1}, \ldots, V_r]$

7. **for all** vertices $v \in V_j$ **do**
   create a new partition

   $\pi_v = [V_1, \ldots, V_{j-1}, \{v\}, V_j \backslash \{v\}, V_{j+1}, \ldots, V_r]$

   $label(G, \pi_v)$.
   end for

8. **end if**

9. Return $Cert(G)$.

Example 3.15 Let's take the Graph G from Example 3.13 and the initial partition $\pi = [\{0, 2\}, \{1, 3\}]$ as input for Algorithm 3.14:

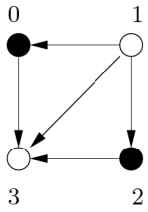


Figure 3.3: Graph with initial partition $\pi = [\{0, 2\}\{1, 3\}]$

1. In the first step, the partition π is refined to an equitable partition. From Example 3.13 we already know that the refined partition is

   $\pi_{eq} = [\{0, 2\}, \{3\}, \{1\}]$.

2. The refined partition is not discrete, so the first non-discrete cell is taken, which is $V_1 = \{0, 2\}$.

3. Now, for each vertex in V_1 a new partition is created:

$$\pi_0 = [\{0\}, \{2\}, \{1\}, \{3\}]$$
$$\pi_2 = [\{2\}, \{0\}, \{1\}, \{3\}],$$

and the algorithm *label* is called recursively for both partitions

$$label(G, \pi_0),$$
$$label(G, \pi_2).$$

Since both partitions are discrete, the numbers $Num_{\pi_0}(G)$ and $Num_{\pi_2}(G)$ are computed, from their respective adjacency matrices, which are:

π_0	0	2	1	3
0	0	0	0	1
2	0	0	0	1
1	1	1	0	1
3	0	0	0	0

π_2	2	0	1	3
2	0	0	0	1
0	0	0	0	1
1	1	1	0	1
3	0	0	0	0

Reading each matrix row by row and we obtain:

$$Num_{\pi_0}(G) = 0001000111010000 \text{ and}$$
$$Num_{\pi_2}(G) = 0001000111010000$$

Both numbers are the same in this case and the algorithm returns the certificate

$$Cert(G) = Num_{\pi_0}(G) = 0001000111010000.$$

Note that the algorithm does not truly know about the vertex coloring of a graph. This information is given by the initial partition, and therefore the sequence of colours must be the same for each graph.

We've seen in this section how the canonical form, i.e certificate, can be calculated for vertex-coloured graphs. The state graphs though, that we computed in Section 2.4 are vertex- and edge-coloured graphs. Therefore, what we need to do, is to transform those graphs into vertex-coloured graphs. That is our aim in the next section.

3.4 Transforming Graphs for NAUTY

To decide whether two graphs are isomorphic, we have implemented in [56] a procedure to compute a canonical form. This implementation was done in Prolog, and was an extension of the core algorithm of [43] for vertex- and edge-coloured graphs.

Now we use the implementation of the canonical labelling algorithm from the NAUTY package [42] directly. Since NAUTY can handle only vertex-coloured graphs, we need to transform our vertex- and edge-coloured state graphs into graphs with labels only on the vertices. We took inspiration from the NAUTY User's Guide [42]. We will first explain our transformation, and then prove its correctness.

Let's consider the following machine, which describes a simple database for a company's staff. It stores only two sets of information: The set of staff that are also members of the staff council, and the marital status of each member of staff.

MACHINE *Personnel*

SETS
$NAME$;
$MARITAL\_STATUS = \{single, married\}$

VARIABLES
$staff\_council$,
$status$

INVARIANT
$staff\_council \in \mathbb{P}(NAME) \wedge$
$status \in NAME \rightarrow MARITAL\_STATUS$

INITIALISATION
$staff\_council := \emptyset \;||$
$status := NAME \times \{single\}$

OPERATIONS
$add\_to\_council(nn) \mathrel{\widehat{=}}$
PRE $nn \in NAME \wedge nn \notin staff\_council$
THEN $staff\_council := staff\_council \cup \{nn\}$
END;

$del\_from\_council(nn) \mathrel{\widehat{=}}$
PRE $nn \in NAME \wedge nn \in staff\_council$
THEN $staff\_council := staff\_council - \{nn\}$
END;

$change\_status(nn) \mathrel{\widehat{=}}$
PRE $nn : NAME$
THEN
IF $status(nn) = single$ **THEN**
$status(nn) := married$
ELSE
$status(nn) := single$
END
END
END

The machine has a deferred set NAME, an enumerated set MARITAL_STATUS, a set variable, *staff_council*, that stores the members of the staff council, and a relation *status*, which gives the marital status of each employee. The database is initialised with an empty staff council and every employee defined as single, by default. Furthermore, the machine has three operations for adding and deleting members from the staff council, and updating the marital status of an employee. A valid state of this machine is

$$\begin{aligned} &member = \{name2, name3\}, \\ &status = \{(name1 \mapsto single), (name2 \mapsto married), \\ &\qquad (name3 \mapsto single)\} \end{aligned}$$

We have already seen in Section 2.4 how such a state is transformed into a graph with labels on both vertices and edges, yielding the graph in Fig. 3.4.

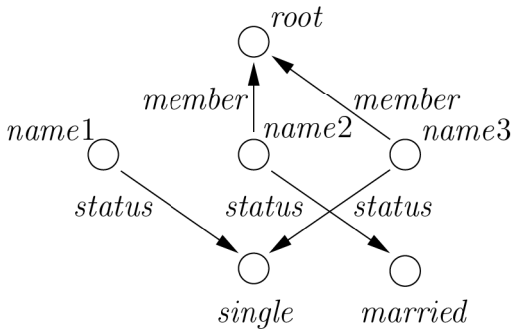


Figure 3.4: State Graph representation

The labels on the vertices and edges are transformed into colours, so that vertices corresponding to the same deferred set get the same colour. Vertices corresponding to an enumerated set in the B-machine get a different colour for each vertex. Considering that the set NAME is deferred, we get the vertex- and edge-coloured graph in Fig. 3.5. We used solid and dashed directed edges to distinguish the set variable *member* from the relation *status*.

Now we need to transform the vertex- and edge-coloured graph g into an only vertex-coloured graph $\hat{g}$, before it can be handed over to NAUTY. The NAUTY User's Guide [42] suggests in Chapter 12, a method describing how an edge-coloured graph can be transformed into a vertex-coloured graph. We implemented a simpler version, because it is easier to prove its correctness. We want to describe our adapted method here now. For each colour (different label) on the edges, there is a layer with all vertices of g constructed. Vertices in $\hat{g}$ that originate from the same vertex in g, are connected with directed edges in each direction. Now, a layer in $\hat{g}$ represents an edge colour in g, and all those edges in g with that colour are added to that layer, connecting the respective vertices. That means, each layer in $\hat{g}$ copies the vertices

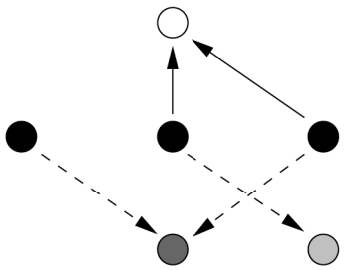


Figure 3.5: State Graph represented as vertex- and edge coloured graph g

of g, and the edges of one particular colour. Vertices in different layers though, have always a different colour. The layered graph $\hat{g}$ is depicted in Figure 3.6.

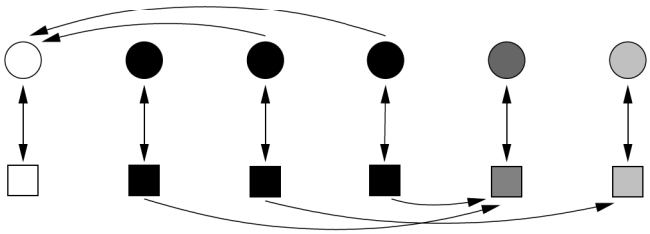


Figure 3.6: State Graph represented as vertex coloured graph $\hat{g}$

We gave the vertices in each layer a different shape, to differ the vertices of the layers, but still showing which vertex in $\hat{g}$ comes from which vertex in g.

We are providing now a more formal description of this transformation. Let $g = \langle V, C_N, C_E, L, E\rangle$ the original vertex- and edge-coloured graph, with C_N and C_E the colours of the vertices and edges respectively, $L : N \to C$ a vertex labelling function, and $E \subseteq C_E \times N \times N$. Then $\hat{g} = \langle \widehat{V}, \widehat{C}_N, \widehat{E}, \widehat{L}\rangle$ is the layered graph, with respective sets of vertices, colours on the vertices and edges, and labelling function. Let n_E be the number of colours on the edges of g. For each $u \in V$ there are $u_i \in \widehat{V}$, where $i = 1, \ldots, n_E$, such that $\{(u_i, u_{i+1}), (u_{i+1}, u_i) \mid i = 1, \ldots, n_E\} \subseteq \widehat{E}$ holds.

In each layer of the graph $\hat{g}$ there is an $u_i \in \widehat{V}, i \in 1, \ldots, n_E$, being a duplicate of $u \in V$. Every such u_i gets a colour depending on the colour of $u \in V$ in the original graph and i, the number of the layer in $\hat{g}$.

There is now one layer for each colour on the edges of g. The edges with colour one are inserted in layer one, edges with colour two in layer two, and so on, connecting the respective vertices of the original graph g. For example, let $e = (u, v) \in E$ have colour 02, then the corresponding edge is inserted in $\hat{g}$ in layer 2, from vertex u_2 to vertex v_2. So we have $(u_2, v_2) \in \widehat{E}$.

In general, for every $e = (u, v) \in E$ with colour $i, i \in 1, \ldots, n_E$ there is an edge $(u_i, v_i) \in \widehat{E}$, where u_i is the corresponding vertex to u and v_i is the corresponding vertex to v in layer i. The edges in $\hat{g}$ are no longer coloured as the corresponding

ones in g. Finally we get a vertex, but no longer edge-coloured graph, so it can now be handled by NAUTY.

Correctness of Encoding Edge Colours as Vertex Colours

We now want to formalise the transformation and prove its correctness. We only formalise and prove the transformation of an edge-coloured graph into a vertex-coloured graph. The extension to a vertex- and edge-coloured graph is straightforward.

Definition 3.16

- An edge-coloured graph is a tuple $\langle V, C, E\rangle$ where V is a set of vertices, C a set of colours, and $E \subseteq C \times V \times V$.
- A vertex-coloured graph is a tuple $\langle V, C, L, E\rangle$ where V is a set of vertices, C a set of colours, $L : V \to C$ a vertex labeling function, and $E \subseteq V \times V$.

Given an edge-coloured (respective vertex-coloured) graph g, we denote by $vert(g)$ the set of vertices of g.

Definition 3.17 Let V and V' be two sets of vertices and π a bijection between V and V'. The bijection π can be applied to an edge-coloured graph $g = \langle V, C, E\rangle$ as follows:

$$\pi(g) = \langle V', C, E'\rangle, \quad \text{where} \quad \begin{array}{l} V' = \{\pi(v) \mid v \in V\} \quad \text{and} \\ E' = \{(c, \pi(v_1), \pi(v_2)) \mid (c, v_1, v_2) \subset E\}. \end{array}$$

The bijection π can be applied to a vertex-coloured graph $g = \langle V, C, L, E\rangle$ as follows:

$$\pi(g) = \langle V', C, L', E'\rangle, \quad \text{where} \quad \begin{array}{l} V' = \{\pi(v) \mid v \in V\}, \\ \forall\, v \in V : L'(\pi(v)) = L(v) \quad \text{and} \\ E' = \{(\pi(v_1), \pi(v_2)) \mid (v_1, v_2) \in E\}. \end{array}$$

Two edge-coloured (resp. vertex-coloured) graphs g_1, g_2 are said to be isomorphic if and only if there exists a bijective function π between $vert(g_1)$ and $vert(g_2)$ such that $\pi(g_1) = g_2$.

We now show how to formally translate an edge-coloured graph into a vertex-coloured graph, encoding every edge-colour as a level in the vertex-coloured graph. We use the following definition:

Definition 3.18 Let $g = \langle V, C, E\rangle$ be an edge-coloured graph. We denote by $level(g)$ the vertex-coloured graph $\langle V \times C, C, L^{\times}, E^{\times}\rangle$, where

- $L^{\times}((v, c)) = c$ for all $(v, c) \in V \times C$,

- $E^{\times} = \{((v,c),(v,c')) \mid v \in V \wedge c, c' \in C \wedge c \neq c'\} \cup \{((v_1,c),(v_2,c)) \mid v_1, v_2 \in V \wedge c \in C \wedge (c, v_1, v_2) \in E\}$

Note that for a vertex- and edge-coloured graph g, the colour of a vertex in $level(g)$ indeed also depends on the colour of the respective vertex v in g.

Proposition 3.19 Let g, g' be two edge-coloured graphs. If g and g' are isomorphic then $level(g)$ and $level(g')$ are isomorphic.

Proof 3.20 First, obviously g and g' must have the same set of colours. Let $g = \langle V, C, E\rangle$ and $g' = \langle V', C, E'\rangle$. Let π be a bijection between V and V' such that $\pi(g) = g'$. We now construct π_l such that

$$\pi_l((v,c)) = (\pi(v), c) \text{ for all } (v,c) \in V \times C.$$

This function is a bijection between $V \times C$ and $V' \times C$. In order to see this, we only need to show that π_l is injective, since it is a function between finite sets. Let $level(g) = \langle V \times C, C, L^{\times}, E^{\times}\rangle$ and (v_1, c_1), (v_2, c_2) be two vertices in $level(g)$. We assume that the images under π_l hold $\pi_l((v_1,c_1)) = \pi_l((v_2,c_2))$ and show then that the preimages must be the same, too. It is

$$\begin{array}{lll} & \pi_l((v_1,c_1)) = \pi_l((v_2,c_2)) & \\ \Rightarrow & (\pi(v_1), c_1) = (\pi(v_2), c_2) & \text{(definition of } \pi_l) \\ \Rightarrow & \pi(v_1) = \pi(v_2) \text{ and } c_1 = c_2 & \\ \Rightarrow & v_1 = v_2 \text{ and } c_1 = c_2 & (\pi \text{ is bijection}) \\ \Rightarrow & (v_1, c_1) = (v_2, c_2). & \end{array}$$

It follows that π_l is injective and therefore a bijective function. We now claim

$$\pi_l(level(g)) = level(g').$$

Let $\pi_l(level(g)) = \langle V' \times C, C, L_2^{\times}, E_2^{\times}\rangle$, and $level(\pi(g)) = \langle V' \times C, C, L'^{\times}, E'^{\times}\rangle$. We need to show $L'^{\times} = L_2^{\times}$ and $E_2^{\times} = E'^{\times}$.

We start with proving that $L'^{\times} = L_2^{\times}$:
For all $v' \in V$ and $c \in C$ we have

$$L'^{\times}((v',c)) \;=\; c \qquad \text{(definition of } L'^{\times})$$

and

$$\begin{array}{lll} L_2^{\times}((v',c)) & = & L_2^{\times}((\pi(v),c)) \quad \text{for some } v \\ & = & L_2^{\times}(\pi_l((v,c))) \qquad\qquad \text{(definition of } \pi_l) \\ & = & L^{\times}((v,c)) \qquad\qquad \text{(Definition 3.17)} \\ & = & c \qquad\qquad \text{(Definition 3.18).} \end{array}$$

Finally we need to show $E_2^{\times} = E'^{\times}$:
We have

$$\begin{aligned} E_2^{\times} &= \{(\pi_l((v,c)), \pi_l((v,c'))) \mid v \in V \ \wedge \ c, c' \in C \ \wedge \ c \neq c'\} \cup \\ &\quad \{(\pi_l((v_1,c)), \pi_l((v_2,c))) \mid v_1, v_2 \in V \ \wedge \ c \in C \ \wedge \ (c, v_1, v_2) \in E\} \\ &= \{((\pi(v),c), (\pi(v),c')) \mid v \in V \ \wedge \ c, c' \in C \wedge \ c \neq c'\} \cup \\ &\quad \{((\pi(v_1),c), (\pi(v_2),c)) \mid v_1, v_2 \in V \wedge c \in C \ \wedge (c, v_1, v_2) \in E\} \\ E'^{\times} &= \{((v,c),(v,c')) \mid v \in V' \wedge c, c' \in C\} \cup \\ &\quad \{((v_1',c),(v_2',c)) \mid v_1', v_2' \in V' \wedge c \in C \wedge (c, v_1', v_2') \in E'\}. \end{aligned}$$

Since π is a bijection, we have

$$\begin{aligned} &\{((\pi(v),c),(\pi(v),c')) \mid v \in V \wedge c, c' \in C \wedge \ c \neq c'\} \\ = \ &\{((v,c),(v,c')) \mid v \in V' \wedge c, c' \in C \wedge \ c \neq c'\} \end{aligned}$$

and furthermore, because of $\pi(g) = g'$, we have $(c, v_1', v_2') \in E'$ implies that for some v_1, v_2 it is $(c, v_1, v_2) \in E$ and $v_1' = \pi(v_1)$ and $v_2' = \pi(v_2)$. Hence $E_2^{\times} = E'^{\times}$. □

For showing the opposite direction, we need the following Lemma which says that if a vertex (v, c) is mapped onto a vertex (v', c) under π, then for any colour $c' \in C$, the corresponding vertex (v, c') is mapped to the corresponding vertex (v', c') with the same v'.

Lemma 3.21 Let π be a permutation such that $\pi(\mathit{level}(g)) = \mathit{level}(g')$. Let $\mathit{level}(g) = \langle V \times C, C, L, E\rangle$ and $\mathit{level}(g') = \langle V' \times C, C, L', E'\rangle$. Then for any $v \in V, v' \in V$ and $c \in C$, we have:

$$\pi((v,c)) = (v',c) \Longrightarrow \forall\, c'.(c' \in C \Rightarrow \pi((v,c')) = (v',c')).$$

Proof 3.22 If we have only one colour then $c' = c$, and the property is obvious. Let's have more than one colour, and $c' \in C$ any colour with $c' \neq c$. By Definition 3.18 we know that for any c': $((v',c),(v',c')) \in E'$ as well as $((v,c),(v,c')) \in E$. We also know by Definition 3.18 that (v', c') is the only successor s of (v', c) with colour c'. Because Definition 3.17 we have that

$$(\pi((v,c)), \pi((n,c'))) \in E'$$

and also

$$L'(\pi((v,c'))) = L((v,c')) = c'.$$

Our assumption was that $\pi((v,c)) = (v',c)$ and we just figured that s is the only successor of (v', c) with colour c'. Hence $\pi((v,c'))$ as another successor of the vertex $\pi((v,c) = (v',c)$ with colour c' must be identical to s. □

Proposition 3.23 Let g, g' be two edge-coloured graphs. The graphs g and g' are isomorphic if $level(g)$ and $level(g')$ are isomorphic.

Proof 3.24 Let $level(g) = \langle V \times C, C, L^{\times}, E^{\times} \rangle$ and $level(g') = \langle V' \times C, C, L'^{\times}, E'^{\times} \rangle$. Let π_l be a bijection between $V \times C$ and $V' \times C$.

First, by Definition 3.17 we know that $L'(\pi_l((v, c))) = L((v, c)) = c$, that means, $\pi_l((v, c)) = (v', c)$ for some $v' \in V'$. We define a bijection π between V and V' by $\pi(v) = v'$, $v \in V$. This definition is well defined, because it is $\pi_l((v, c)) = (v', c)$, and with Lemma 3.21 it follows that

$$\pi_l((v, c')) = (v', c') \quad \text{for all } c' \in C.$$

We want to prove that $\pi(g) = g'$. We already have a bijection π between the sets of vertices of g and g' defined. So all we need to show is that for any edge between two vertices v_1, v_2 in E with colour c, there is an edge with colour c between the respective vertices $\pi(v_1)$ and $\pi(v_2)$ in E', and vice versa. We have by Definition 3.18 that for any $(c, v_1, v_2) \in E$, there is an edge $((v_1, c), (v_2, c)) \in E^{\times}$. Since $level(g)$ and $level(g')$ are isomorphic, the edge $(\pi_l(v_1, c), \pi_l(v_2, c))$ is in $E'^{\times}$. We know that

$$(\pi_l(v_1, c), \pi_l(v_2, c)) = ((v_1', c), (v_2', c)) \quad \text{for some } v_1', v_2' \in V'. \tag{3.1}$$

Since we defined π such that $v_1' = \pi(v_1)$ and $v_2' = \pi(v_2)$, we get by applying Definition 3.18, that $(c, \pi(v_1), \pi(v_2)) \in E'$.

The reverse statement follows with similar arguments. Indeed,

$$(c, \pi(v_1), \pi(v_2)) \in E' \Rightarrow ((\pi(v_1), c)(\pi(v_2), c)) \in E'^{\times} \quad \text{(Def. 3.18)}$$

With the definition of π and the properties of π_l following from Definition 3.17, we get

$$\begin{aligned} ((\pi(v_1), c)(\pi(v_2), c)) &= ((v_1', c), (v_2', c)) \\ &= (\pi_l(v_1, c), \pi_l(v_2, c)) \quad \text{for some } v_1, v_2 \in V. \end{aligned}$$

Again, since π_l is an isomorphism, we have $((v_1, c), (v_2, c)) \in E^{\times}$. It follows that $(c, v_1, v_2) \in E$ from Definition 3.17. □

We've shown and proven correct the translation of vertex- and edge-coloured graphs to vertex-coloured graphs. NAUTY can now calculate the canonical form of those graphs and find those that are isomorphic. Isomorphic graphs correspond to symmetric B states, so that with this information, only one state of each symmetry class needs to be explored. In the next section, we describe how the graph canonicalisation is integrated into the model checking algorithm, in order to achieve a reduction of the state space.

3.5 The Model Checking Algorithm

We now formalise our modified model checking algorithm and show how symmetry detection via graph isomorphism has been integrated into the model checking. Algorithm 3.25 has been adapted from [56].

Algorithm 3.25[*Model Checking with Symmetry Reduction*]

Input: An abstract machine M

1. $Queue := \{root\}$; $Canon := \{\}$; $SGraph := \{\}$
2. **while** *Queue* is not empty **do**
3. **if** random(1) $< \alpha$ **then**
   state := pop_from_front(*Queue*); /* depth-first */
   else
   state := pop_from_end(*Queue*); /* breadth-first */
   end if
4. **if** *error*(*state*) **then**
   return counter-example trace in *SGraph* from *root* to *state*
5. **else**
   for all *succ*, *Op* **such that** $state \rightarrow^{M}_{Op} succ$ **do**
   $sg := nauty\_canon(\mathcal{G}(succ))$
   if $\exists\, s$ such that $(sg, s) \in Canon$ **then**
   $SGraph := SGraph \cup \{state \rightarrow^{M}_{Op} s\}$
   else
   add *succ* to front of *Queue*
   $Canon := Canon \cup \{(sg, succ)\}$
   $SGraph := SGraph \cup \{state \rightarrow^{M}_{Op} succ\}$
   end if
   end for
   end if
6. **od**
7. **return** ok

Step 1 initialises the variables. The variable *Queue* stores the states with transitions yet to be explored, and is initialised with *root*. The *root* node is not a real state of the machine, and only there as starting point. *Canon* records canonical forms of states already reached, along with the corresponding state. It is an empty set, initially. The variable *SGraph* stores the visited state space, so that traces to occurring error states can be produced. This variable is also initialised with *root*. The steps 2 to

4 are the same as for the ordinary model checking algorithm in Section 1.3. Step 5 calculates to all successor states of a state their canonical form. The function $\mathcal{G}$ converts a state of a B machine into a labelled, directed graph, as explained in Section 2.4. The function *nauty_canon* computes a canonical form for such a graph using NAUTY, as explained earlier. If the canonical form of a state graph is already in *Canon* then the respective state is just added to *SGraph*. Otherwise that state is also added to the *Queue* and its canonical form to the set *Canon*. This is done for all successor states. As long as the *Queue* is not empty, the algorithm continues searching the state space.

Example 3.26 We want to take the machine *Personnel* from Section 3.4 as an example, to explain the model checking algorithm in more detail with respect to the canonicalisation. We assume that the cardinality of deferred sets is three. After step 1, the variables *Queue* and *SGraph* store the *root* node, while the variable *Canon* is empty.

The algorithm enters the while loop, since *Queue* is not empty, and takes the only node, the *root* node, from the Queue of unexplored states. The *root* node is not an error state, so the algorithm now computes all successor states of the *root* node, which in this case is only the state after the initialisation of the machine. Then the canonical form of the state graph of the initial state is computed, with the function *nauty_canon*. Since *Canon* is empty so far, this canonical form is not in *Canon*, and therefore the initial state is added to *Queue*, its canonical form is stored in *Canon* together with the initial state. The *root* node together with a directed edge to the initial state, is added to *SGraph*.

Now there is the initial state in *Queue*, so the algorithm continues with exploring the initial state. The algorithm checks if the initial state satisfies the invariant with the *error*() function. We have

$$\begin{aligned} &staff\_council = \emptyset \in I\!P(NAME) \text{ and} \\ &status = \{(NAME1, single), (NAME2, single), (NAME3, single)\} \\ &\in NAME \rightarrow MARITAL\_STATUS, \end{aligned}$$

so the invariant is true.

Again, the algorithm looks at all successor states. The successor states are all those states, that can be reached with the execution of an operation exactly once. The operation *add_to_staff_council*(nn) can be executed for each element in the deferred set $NAME$. Since $card(NAME) = 3$, we have three possible operation calls. The operation *del_from_staff_council*(nn) cannot be executed in the initial state, since the variable *staff_council* is empty in this state, therefore the precondition of the operation is not satisfied. The operation *change_status* has again three possible executions. Altogether, this gives us six successor states from the initial state.

For each successor state, the canonical form is computed, and compared with the already existing canonical forms in the variable *Canon*. Let's look at the state

graph for the state

$$s1 = (staff\_council = \{NAME1\},$$
$$status = \{(NAME1, single), (NAME2, single), (NAME3, single)\})$$

in comparison with the graph for the initial state

$$init = (staff\_council = \{\},$$
$$status = \{(NAME1, single), (NAME2, single), (NAME3, single)\})$$

and the respective corresponding vertex-coloured graph, that is fed to NAUTY.

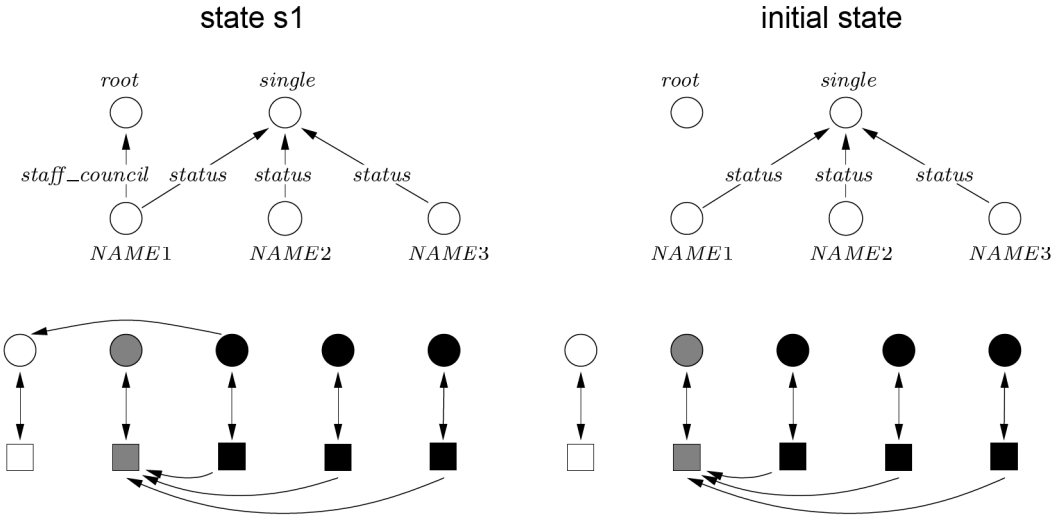


Figure 3.7: State graphs of not isomorphic state $s1$ and initial state and corresponding vertex-coloured graphs

We can see that the graphs are not isomorphic, and therefore the canonical forms will be different. Consequently, the state $s1$ is added to *Queue*, its canonical form together with $s1$ is stored in *Canon*, and a new node for $s1$ and an edge going from the initial state to $s1$, is added to *SGraph*.

Now the algorithm proceeds to the next state, e.g.

$$s2 = (staff\_council = \{NAME2\},$$
$$status = \{(NAME1, single), (NAME2, single), (NAME3, single)\}).$$

This state is reached from the initial state by executing $add\_to\_staff\_council(NAME2)$. Again, the canonical form is computed and compared with each canonical form stored in *Canon*. We have now the canonical forms of the initial state *init* and $s1$ in *Canon*. Let's compare the corresponding graphs of $s1$ and $s2$ as in Fig. 3.8.

These two graphs are isomorphic and will produce the same canonical form. The algorithm adds only a node for state $s2$ and an edge from *init* to $s2$ to *SGraph*, but

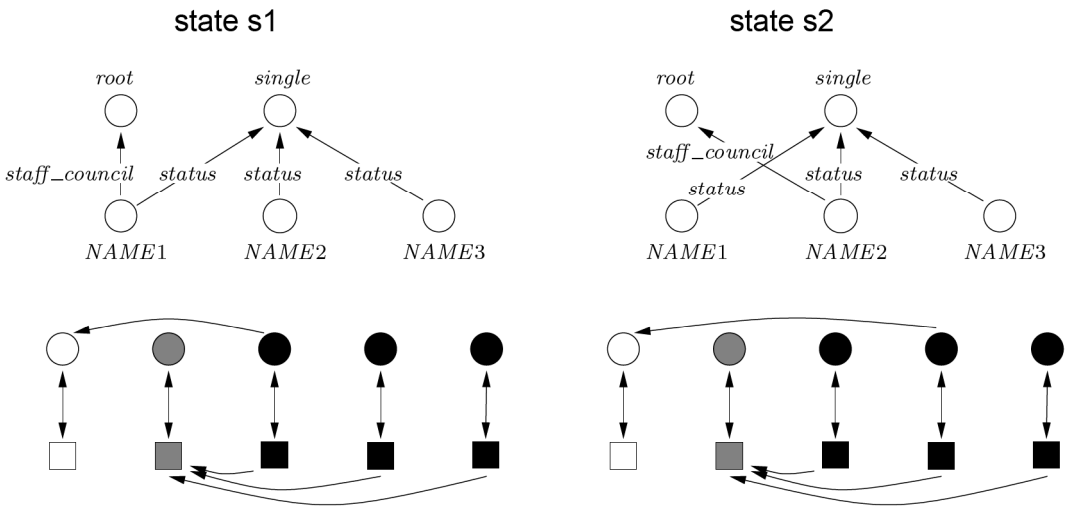


Figure 3.8: State graphs of isomorphic states $s1$ and $s2$ and corresponding vertex-coloured graphs

no element to the variables *Canon* and *Queue*. The algorithm continues with the next state $s3$, following the execution of $add\_to\_staff\_council(NAME3)$. This state is again isomorphic to $s1$ and therefore not added to *Queue*.

Another set of isomorphic states is produced by executing *change_status* after the initialisation for each element in the deferred set *NAME*. The state $s4$ belongs to this new isomorphism class:

$$s4 = (staff\_council = \emptyset,$$
$$status = \{(NAME1, married), (NAME2, single), (NAME3, single)\}).$$

The graphical representation for the state is depicted in Figure 3.9.

This graph is not isomorphic to any of the previous graphs, and therefore the new canonical form - together with the corresponding state $s4$, is added to *Canon*, and $s4$ is added to *Queue*.

After considering all successor states of the *init* state, the graph of the so far explored model, *SGraph*, is shown in Figure 3.10 and we have the states $s1$ and $s4$ in the variable *Queue*.

If we used the model checking algorithm without symmetry, then at this stage we would obtain the graph of the explored model as in Figure 3.11 and six successor states in the queue of states to be explored rather than only two.

The algorithm continues now with model checking of one of the two states in *Queue*, depending on the value of the random function and the value of α. If the state is an error state, then the algorithm returns the trace from the *root* node to that state, and continues with adding the successor states to *SGraph* otherwise.

We have seen in the example, how easily the state space is reduced by using symmetry. We have implemented this algorithm within ProB, and we provide

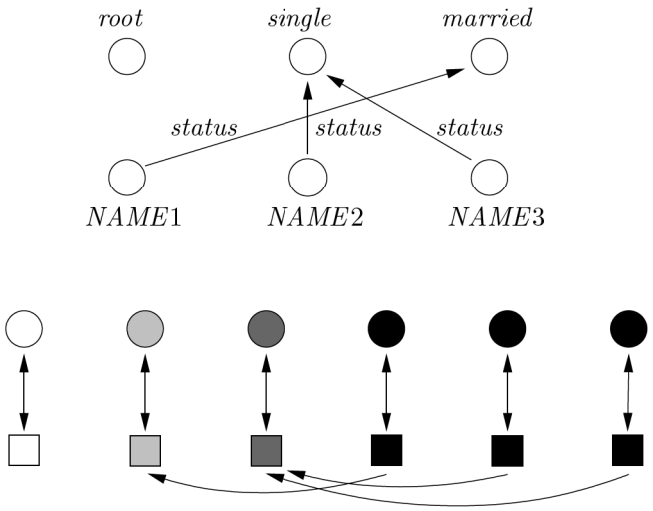


Figure 3.9: State graph of state $s4$ and corresponding vertex-coloured graph

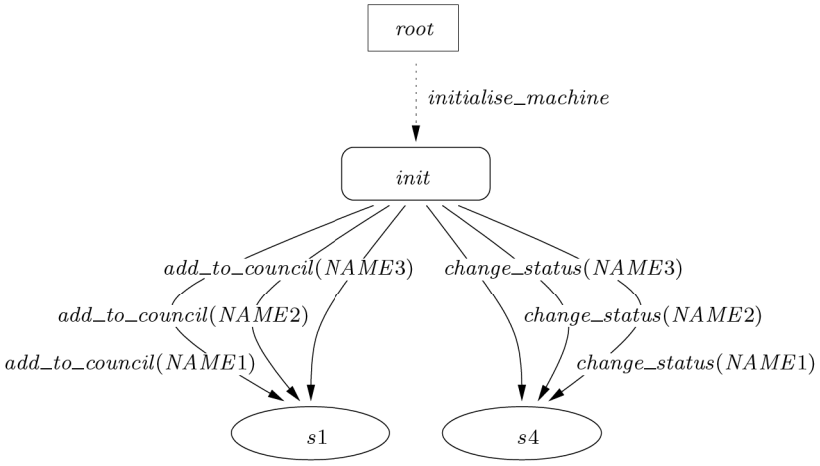


Figure 3.10: Explored state space after initialisation with symmetry

empirical results later in Chapter 4.

3.6 The Interface between NAUTY and PROB

We have explained so far how states can be translated to vertex-coloured graphs, so that NAUTY can calculate a canonical form to detect symmetric states. We also discussed how symmetry is applied in the model checking algorithm. In this section we want to explain how NAUTY has been integrated into PROB from a technical point of view. PROB does the translation from states to vertex-coloured graphs. NAUTY takes a vertex-coloured graph and computes the canonical form for that

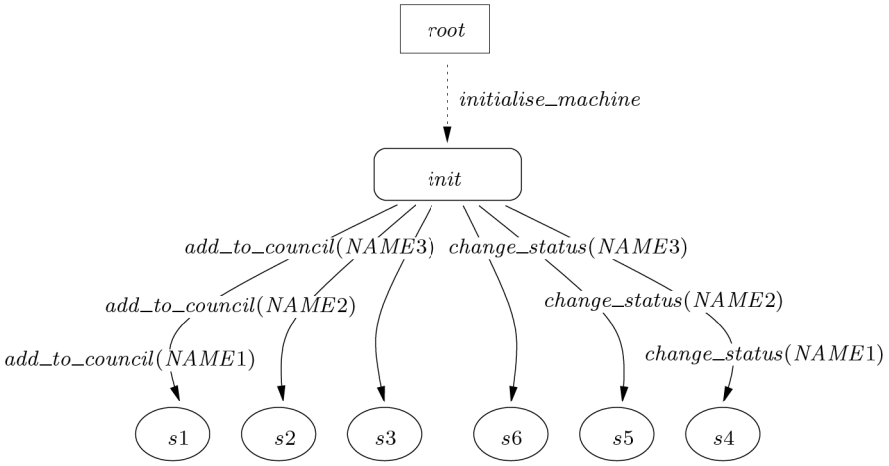


Figure 3.11: Explored state space after initialisation without symmetry

graph. However, NAUTY has very specific and complex data structures to handle graphs, so that ProB cannot communicate its translated graphs directly to NAUTY and NAUTY cannot just hand back the canonical forms to ProB. What we need is an interface that can build a graph in NAUTY's format from ProB's input and handle each canonical form that NAUTY calculates for each graph. The idea is, that ProB first transfers the data of a graph, piece by piece, with simple function calls to the interface. Then the interface calls NAUTY to calculate the canonical form to that graph, and compares it to previously stored canonical forms. All that the interface needs to tell ProB, is whether that canonical form has been encountered before, or not. Figure 3.12 graphically shows the function of the interface we have implemented.

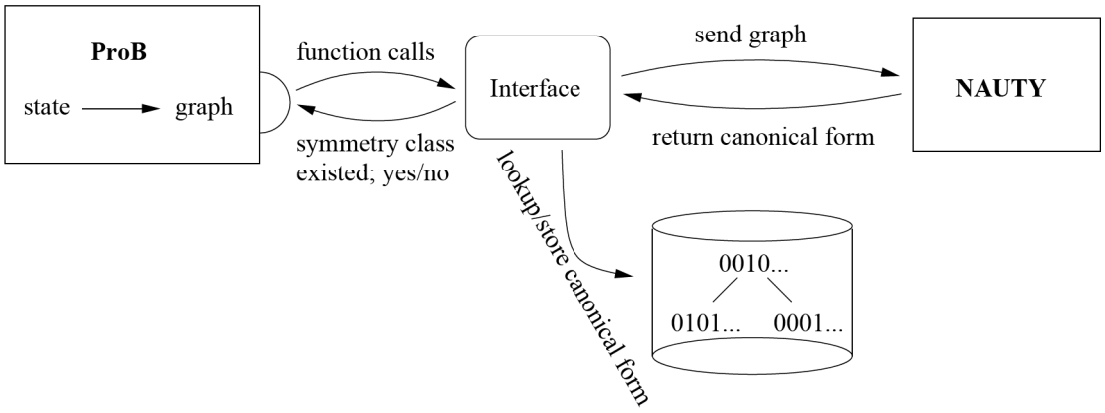


Figure 3.12: Functionality of the Interface

We want to describe the interface now, in a bit more detail. The interface has a set of C-functions, that can be called by PROB to transfer information. Those functions are listed below:

void prob_init(void);
void prob_set_number_of_colours(int number_of_colours);
void prob_start_graph(int number_of_nodes);
void prob_add_edge(int from, int to);
void prob_set_colour_of_node(int node, int colour);
int prob_exists_graph(void);
void prob_free_storage(void);

The prefix *prob_* in each function name indicates that this function is called from PROB. Those functions that take a parameter, call another function with the same name without the prefix *prob_* . This separation was done, because it is easier to send only simple data structures, such as interger values, from PROB to the interface and vice versa. Consequently, the parameters and output values of those functions called by PROB need to be basic data structures or void. In order to store the information, though, the internal functions take more complex data structures as parameters. The source code for each of these functions can be found in the file *internal_functions.c* in Appendix A.3. The source code for the functions called by PROB are in the file *interface.c* in Appendix A.1.

During a model checking process, the first two functions, *prob_init(void)* and *prob_set_number_of_colours(int number_of_colours)*, are called only once. The function *prob_init(void)* allocates memory and initialises a set of variables used throughout the program. PROB then transfers information about how many colours are going to be needed during the model checking of a machine, by calling the function *prob_set_number_of_colours(int number_of_colours)*. Then, the function *prob_start_graph(int number_of_nodes)* is called. This function takes the number of vertices, also called nodes in the source code, as input from PROB. It allocates memory and initialises variables for working with a particular graph G. The most important variables initialised here are those for storing the graph G, called *global_g* in the source code, and its canonical form, *global_canong*. The canonical form is just a different representation of the graph and therefore has the same data format as the graph. We described this format, which is used by NAUTY, in Section 3.1. Also important is the encoding of the initial partition. The variable *cell_list* stores for each colour the set of vertices with that colour, and therefore contains the information for the initial partition. Though strictly not necessary, the sizes of each cell are stored for easier reference in another variable, called *cell_sizes*. Note that the functions *prob_init(void)* and *prob_set_number_of_colours(int number_of_colours)* need to be called before *prob_start_graph(int number_of_nodes)* in order for the initialisations to take place.

When all related variables are initialised, ProB can start transferring the data for edges and vertex colours of the graph to the interface, by using the functions *prob_add_edge(int from, int to)* and *prob_set_colour_of_node(int node, int colour)*. All information is encoded in the form of integer values. The latter function takes a vertex and its colour, and stores it according to its colour, in the appropriate cell within *cell_list*. The variable *cell_list* is just one long array of integers, and the cells are separated by an offset. After the colour of a vertex has been set, all outgoing edges from that vertex are added, one by one, to the graph with the function *prob_add_edge*(*int from*, *int to*). This function takes two vertices, that are joined by an edge, as input parameters. It calls the internal function *add_edge*(...), which uses some functions from NAUTY to make the respective changes to the variable *global_g*.

Once all the data for a graph has been transferred and stored in a NAUTY friendly format, ProB wants to know, if the symmetry class of the graph G has been encountered before, or not. In order to answer this question, ProB calls the only function with a return value, *int prob_exists_graph(void)*. Now, before NAUTY is called within this function to calculate the canonical form, there are various NAUTY options and parameters set. Most important for our purpose are the NAUTY options *options.digraph* and *options.getcanon*. Those options need to be set to *TRUE*, since our graphs are directed graphs and we want NAUTY to calculate the canonical form.

We also implemented a function to check if all vertices in G have been given a colour. In cases where a vertex has no colour assigned, it gets the default colour 0. This makes sure that NAUTY works correctly.[2] When all vertices have been given a colour, the initial partition can be set. NAUTY has two parameters, $*lab$ and $*ptn$, to represent this information. The parameter $*lab$ stores the vertices of G in the order of the cells, while $*ptn$ indicates where the end of one cell is. An internal function named *set_label* assigns those variables the correct values according to the colouring information stored in the variable *cell_list*. With all parameters set, NAUTY can be called to calculate the canonical form. The function call looks as follows:

nauty(global_g, lab, ptn, NULL, orbits,&options,&stats, workspace,
*100*MAXM, m, global_n, global_canong);*

For more information on the individual parameters, see NAUTY User's Guide [42]. The resulting canonical form is stored in the variable *global_canong*. Previously calculated canonical forms have been stored in a binary tree.[3] Now the interface tries to insert the canonical form in the binary search tree, by calling its function *insert_canon*, found in the file *graph_tree.c*, see Appendix A.4. If the same canonical form is found in the tree, then that means that the respective symmetry class of G has been encountered before, and the function just returns with a positive value. Otherwise, the new canonical form is inserted in the tree, and returns with a negative

[2] If vertices are not given a colour through ProB, then this can lead to a large initial partition, which in turn can cause a long runtime.

[3] Canonical labels are binary strings, so defining a lexicographic ordering is straightforward.

value. So, a positive value tells PROB, that the symmetry class of G is already evaluated and therefore the state represented by G does not need to be checked anymore. A negative value indicates that PROB has some more work to do on the respective state. When PROB has finished model checking a machine, then it calls a tidying routine to clean up allocated memory. The following example shows a listing of function calls as they could be called by PROB:

Example 3.27

$$
\begin{aligned}
&\mathit{prob\_init}();\\
&\mathit{prob\_set\_number\_of\_colours}(230);\\
&\\
&\mathit{prob\_start\_graph}(2);\\
&\mathit{prob\_set\_colour\_of\_node}(0,6);\\
&\mathit{prob\_add\_edge}(0,1);\\
&\mathit{prob\_set\_colour\_of\_node}(1,6);\\
&\mathit{prob\_exists\_graph}();\\
&\qquad\vdots\\
&\mathit{prob\_free\_storage}();
\end{aligned}
$$

Note that the canonical form is stored together with the number of vertices and the sizes of each cell for each colour, where empty cells have the size 0. This additional information to the canonical form is necessary, because the canonical form depends on the initial partition. We discussed in Section 3.3, that the ordering of the colours must be the same, when graphs are tested for isomorphism. Now, even though the sequence of colours is fixed at the beginning for all graphs, we can have two graphs that use different colours, but still end up with the same initial partition and the same adjacency matrix. In order to distinguish such graphs we need to compare for each colour the number of vertices with that colour, i.e. the information stored in the variable *cell_sizes*.

Example 3.28 Let's consider the two graphs depicted in Figure 3.13 and assume the sequence of colours is [‘*black*‘, ‘*grey*‘, ‘*white*‘].

Figure 3.13: Non isomorphic graphs with same canonical form

Clearly both graphs have the same initial partition and the same adjacency matrix:

$g_1 : \{0\}, \{1,2\}$ $\quad$ $g_2 : \{0\}, \{1,2\}$

g_1	0	1	2
0	0	0	1
1	0	0	1
2	0	0	0

g_2	0	1	2
0	0	0	1
1	0	0	1
2	0	0	0

This would lead to the same canonical form, and we would falsely report that these graphs are isomorphic. Those two graphs can only be distinguished, when we apply a generalised definition of the initial partition. Namely, by including the empty cells. Then we have the partitions

$g_1 : \{0\}, \{1,2\}, \{\}$ $\quad$ $g_2 : \{0\}, \{\}, \{1,2\}$

When we now compare the size of each cell of g_1 with the size of the respective cell in g_2, we see that cell 2 of g_1 contains two vertices while cell 2 of g_2 is empty. Consequently g_1 and g_2 are discovered as non-isomorphic.

This comparison has been implemented in the function *compare_to_internal* in the file *graph_tree.c*, see Appendix A.4. Generally, a new graph G is compared to an already stored graph, first by the number of vertices, then by the sizes of the cells in the generalised initial partition and lastly by its canonical form. If all comparisons turn out to be equal, then the symmetry class of the graph G has been encountered before. The interface indicates this to ProB with a respective return value.

3.7 Related Work

3.7.1 Murφ

Protocols are very important tools used for communication over networks. They provide a set of rules, such as *"If the line is free, a device can start sending packets"*, to allow a controlled and successful interaction between devices. However, designing such a protocol is somewhat difficult, since the designer has to consider numerous dependencies within the protocol and unusual conditions in which the protocol can be used. Errors are made easily during design and a simulation of the protocol can filter out only certain types of errors effectively. Those errors, that occur in some odd circumstances and non-deterministically are not reliably found by simulation. This is where formal verification could help, and the reason why Murφ [19], a protocol description language and verifier, has been developed in 1992 and enhanced since.

Murφ was designed to be simple, while supporting non-deterministic scalable descriptions. It uses a set of integrated guarded commands to describe a system, which is inspired by Misra's Unity language [12]. A guarded command consists of a condition, i.e. Boolean expression, and a sequence of actions. A Murφ description contains a set of invariants separately. Those invariants are Boolean expressions

constraining the variables. Each invariant needs to be true in every state. The description is compiled by the Murφ complier to a C++ verifier program, which is then used to verify the description by searching its state space. The verification is similar to model checking in B in that the state space is searched with breadth-first or depth-first search for states that violate any invariant, or are a deadlock state. For any error found, there is an error message, and the trace of transitions to that erroneous state is generated.

Symmetry has been introduced to the Murφ description language by a data type called *scalarset* [28]. User defined data types are each represented by a range of integers. Whenever the integer values of a particular range don't matter throughout the Murφ description, that range can be replaced by a scalarset. Therefore elements of a scalarset can be permuted, without changing the behaviour of the Murφ description. For example, consider a three node cluster as part of a specification for a data centre. Then it's of no importance which individual machine responds to a data request, what matters is that one does.

Scalarsets are restricted to those operations that don't refer to a particular element of a scalarset and also don't imply an ordering on the scalarset. That means elements of a scalarset can be compared for equality or inequality, but not with the binary operators '>' or '<'. A loop that iterates over all elements of a scalarset is feasible, but not a conditional statement that depends on a particular element of a scalarset. The Murφ compiler notifies the user when a symmetry breaking operation is used. So the user gets a feedback, if he introduced a scalarset in the wrong place. Similar to B, the state space can be reduced to its quotient model and therefore achieve savings of the state space of over 90%, see the Section "Practical Results" in [19].

3.7.2 The Model Checker Spin

Spin is a very efficient and widely-used model checker developed by G. J. Holzmann [24]. It takes specifications written in the Process Meta Language *Promela* [22], and verifies correctness claims expressed in LTL [48]. The algorithm used for model checking is an improved depth-first search algorithm [26]. Spin uses various methods to optimise the model checking process by reducing the state space and memory consumption. Those methods include partial order reduction [46], state compression [25] and bitstate hashing [23].

The extension of Spin with the symmetry reduction package SymmSpin [9] adds to the list of optimisation methods in Spin. The idea by Ip and Dill [28] of introducing scalarsets as a new data type to the Murφ description language, has been picked up and adjusted for the integration into Spin. In fact, the SymmSpin package does not use a new data type for scalarsets but the data type *byte*. So, in SymmSpin scalarsets are represented by a range of integers $0..n-1$, where $n < 256$. This is done in order to avoid modifications of Spin's Promela parser. Symmetry is also not described within a Promela specification itself, but in an additional file supplied by

the user.

However, in order to find representatives for symmetry classes, SymmSpin uses some heuristics, as described in Bosnacki et al. [10], which leads to four different strategies we will mention later in this section. Any state can be represented by a state vector, which is just an array containing the values of all state variables. By introducing a lexicographic ordering on the state vectors, one can define representatives, as done in [28]. The state vector is separated in two parts. On the first part, the minimal representative, i.e. canonical form according to the lexicographical ordering, is calculated. Of all permutations that lead to the canonical form, there is one arbitrary permutation picked to induce a permutation on the second part of the state vector. That means the second part is *normalised* after the first part. Since the permutation is chosen arbitrarily, there might be several representatives of one symmetry class stored throughout the model checking process.

The first of the four strategies works essentially as described above, but with one modification: The variables of the state vector are heuristically sorted in a way that reduce the number of permutations, after canonicalising the first part of the state vector. This strategy is also called the *sorted strategy*. The second strategy, called *segmented strategy*, considers all permutations after canonicalising the first part, and takes the permutation of that set that leads to the smallest state vector. With this permutation, the second part is canonicalised, and we obtain a canonical form. Compared to the sorted strategy, the segmented strategy needs less memory, since only one canonical form needs to be stored, rather than potentially several normalised forms. On the other hand, calculating the canonical form needs more computation time. The other two strategies are variants of the sorted and segmented strategies.

Rather than splitting the state vector, it is also possible to apply a canonical function to the whole state vector. This strategy is called the *full strategy*. Bosnacki et al. used this strategy as reference in their empirical results. They show that each of their strategies can lead to the best reduction, where at least one always outperforms the full strategy in their experiments. Compared to using no symmetry, they achieved significant reductions. This suggests to fully integrate symmetry reduction into the Spin model checker.

3.7.3 RuleBase

RuleBase is a model checker developed by the IBM Haifa Research Laboratory [8]. It has been especially designed for the formal verification of hardware, but it has been successfully applied also in protocol verification. RuleBase uses an enhanced version of the symbolic model checker SMV [44]. The specification language of SMV is CTL, which is difficult to understand by ordinary designers. In order to improve usability, RuleBase comes with its own specification language, called *Sugar*. Sugar is based on CTL, but it is far easier to understand by non-experts. Indeed, RuleBase has been developed with usability and industrial practicality in mind. It

offers various analysis and debugging tools. Noteworthy here is RuleBase's support for finding formulas that hold from a model checking perspective, but are meaningless from the design perspective. An example taken from [8], could be the formula *"if a and then b, c must hold"*. Now, if the sequence *"a and then b"* never happens, then the whole formula passes model checking in every state, but is irrelevant overall. To identify such formulas, RuleBase gives warnings and also allows the user to re-translate a formula to natural language. This gives the user the opportunity to find out, if the formulas express what he intended.

However, RuleBase does not just aim to be an easy to use model checking tool, but also offers various reduction methods to reduce time and memory consumption. Generally, the state space explosion problem is addressed by using binary decision diagrams (BDDs) in symbolic model checking. So RuleBase, being based on the symbolic model checker SMV, makes use of that. For large systems though the use of BDDs is still not sufficient to make model checking feasible. The idea of symmetry reduction, which is very successfully used with other model checkers, is more difficult to apply in symbolic model checking, because of the symbolic representations. Nevertheless Barner and Grumberg [7] managed to integrate symmetry reduction into the RuleBase model checker by using *under-approximation*. That means, after encountering a new state, there is only a subset of successor states explored. So, instead of calculating representatives for each symmetry class, a set of representatives is picked as new states are encountered. This choice is not done randomly, though, but by using symmetry information supplied by the user. The algorithm can prove that a given system does not meet its specification, and therefore provide a falsification. Under certain circumstances it can also verify a model.

Experiments suggest that respectable savings in time, and often memory, are achieved with this method. Unfortunately the savings depend, to a large degree, on the input of symmetry information from the user. This is also true for the use of scalarsets in Murφ, and Spin with the SymmSpin package.

3.7.4 The Alloy Analyser

The language alloy [29] is a very small modelling language based on Z [52]. It was designed to be easy to read and write, but with sufficient expressiveness and the possibility of automated analysis. The idea of automatic verification is inspired by model checking. However, the respective tool for alloy, called alloy analyser [30], is based on propositional satisfiability (SAT) solving. An alloy model is translated to a Boolean formula written in conjunctive normal form (CNF). The Boolean formula is passed to the alloy analyser's integrated SAT solver to find variable values, such that the formula evaluates to `true`. Such a set of variable values is said to be an *instance* of the model. An instance can either be a proof that a certain property of the model holds, or it can serve as a counter-example and show that a property does not hold. Based on the formula, the instance is translated back to values of variables and constants of the model. The SAT problem is

an NP-complete problem, so the scope for each variable and constant in a model should be fairly small.

Applying symmetry during the verification of an ALLOY model works a little different to the methods described in the previous sections. There, we had more or less complex methods to find canonical- or normal forms for a symmetry class of states. Now, the most expensive step the ALLOY ANALYSER has to perform, is to find solutions to a Boolean formula. So, the idea here is to avoid searching for symmetric solutions. This can be done by adding *symmetry-breaking predicates* [15] to the Boolean formula, before it is given to the SAT solver. The ALLOY language allows to infer symmetry information from the data structures used in a model. Consequently there is no need for the user to supply any symmetry information. This information is then used to construct *partial symmetry-breaking predicates*. Partial symmetry-breaking predicates are chosen, such that at least one instance for each symmetry class is found, but maybe more. The idea is to pick those predicates, that lead to a small number of instances, without adding too much computation effort for finding solutions to the Boolean formula.

There are various case studies showing the successful application of this method. Two examples are "The design and implementation of an intentional naming system" [2] and "Tsafe: Building a trusted computing base for air traffic control" [17].

3.7.5 More Symmetry Reduction for B

In this subsection, we want to discuss related work towards symmetry reduction for B. There have been three different attempts to use symmetry to combat the state space explosion problem of B models in particular. We have seen in Section 2.4 how B-states can be represented as vertex- and edge-coloured graphs. The first attempt to use symmetry for B models was an implementation of the canonical labelling algorithm, as described in Section 3.3, for vertex- and edge-coloured graphs in Prolog. The details of this work are described in the article "*Symmetry Reduced Model Checking for B*", [56]. We will compare this work with our new approach using NAUTY in Chapter 4.

Symmetry Reduction by Permutation Flooding

The Permutation Flooding approach works differently from classical symmetry reduction, in that there is no need for the calculation of a representative for each symmetry class. Instead, whenever a new state is encountered during model checking, a set of symmetric states is calculated, by permuting the elements of deferred sets[4], and then added to the set of already evaluated states. Therefore, only the one new encountered state needs to be model checked, but not all its symmetric states as well.

[4]Under certain circumstances, elements of enumerated sets can also be permuted, see [39].

Example 3.29 Let's take again the personnel machine from Section 3.4. The execution of the operation $add\_to\_council(NAME1)$ after the initialization of the machine leads to the state:

$$s1 = (staff\_council = \{NAME1\},\\ status = \{(NAME1, single), (NAME2, single), (NAME3, single)\}).$$

Permuting the elements of the deferred set NAME, which shall have a cardinality of three, we get two symmetric states:

$$s2 = (staff\_council = \{NAME2\},\\ status = \{(NAME1, single), (NAME2, single), (NAME3, single)\})$$

and

$$s3 = (staff\_council = \{NAME3\},\\ status = \{(NAME1, single), (NAME2, single), (NAME3, single)\}).$$

In a similar manner, the execution the operation $change\_status(NAME1)$ after the initialization leads to:

$$s4 = (staff\_council = \{\},\\ status = \{(NAME1, married), (NAME2, single), (NAME3, single)\}),$$

where permuting the elements of the deferred set NAME again gives the symmetric states:

$$s2 = (staff\_council = \{\},\\ status = \{(NAME1, single), (NAME2, married), (NAME3, single)\})$$

and

$$s3 = (staff\_council = \{\},\\ status = \{(NAME1, single), (NAME2, single), (NAME3, married)\}).$$

Looking at the resulting state space in Figure 3.14, at first glance it seems that we only saved the execution of some transitions which have been replaced by a permutation operation. However, the states found though permutation of elements of the deferred set *NAME* are considered as already visited, that means they don't have to be model checked and they don't have their subsequent transitions computed, which leads to a big saving in computation time.

Generally, this approach has the following advantages over model checking without symmetry:

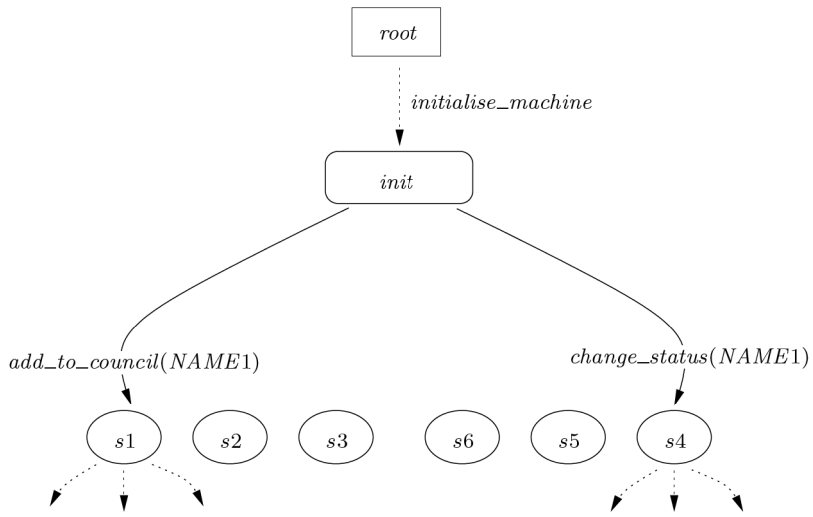


Figure 3.14: Explored state space

- The states calculated by permuting elements of deferred sets don't have to be model checked.
- There is no need for the calculation of successor states for permuted states.
- The calculation of the permutation function is simple, even for more complex data structures, see [39] for more details.

The obvious disadvantage is that there can be many permutations of elements of deferred sets creating a rather large number of symmetric states. Compared to exhaustive model checking, where every state is encountered anyway, there is nothing lost, but compared to the method described in this thesis, the 'flooding' of the state space is quite an overhead. We will compare both methods in more detail later in Chapter 4.

Symmetry Reduction using Symmetry Markers

The main problem in Symmetry Reduction is to find a function that always decides correctly, whether two states are symmetric or not, and is also easy to compute. The Symmetry Marker approach [40] slightly weakens the correctness requirement in favour of speed. The idea is to define a hash function, that assigns a hash value to each state, such that symmetric states are guaranteed to get the same hash value and non-symmetric states get different hash values with a very high likelihood. Before a state is analysed, there is a hash value computed for that state, so that if a previously encountered state had the same hash value then the current state does not need to be analysed further. The assumption is that non-symmetric states get different hash values. Since this is not necessarily the case in practice, the method is an

approximate verification method. That means, it cannot guarantee the correctness of a model but allows a falsification.

The idea of approximation has been successfully used by Holzmann's bistate hashing technique [23]. The Symmetry Marker approach takes this idea, but replaces the hash values by a symmetry marker for each encountered state. The structure of a marker is more complicated than a hash value, because it also integrates the notation of symmetry. The modified model checking algorithm then stores the symmetry markers instead of the state. Any newly encountered state is only checked, if its marker has not been stored before. Similar to hash values, two symmetric states have always the same hash marker, but it is possible that non-symmetric states falsely get the same marker, too. In the event of such a collision, some of the state space may not be checked.

The main source for symmetries in B are deferred sets. Consequently, they play an important role in the definition of the marking function. Indeed, the main idea of the marking function is to replace the deferred set elements of an encountered state by so-called *vertex invariants*. Vertex invariants are known in graph theory as functions, that label the vertices of a graph, such that symmetrical vertices obtain the same value. Simple examples for vertex invariants on graphs are the in-degree and out-degree functions. Generalising the idea of those two function leads to a vertex invariant for deferred set elements in B. A formal description of how this idea is implemented can be found in [40]. We just want to give an overview here. The following notations are taken from [40]. We describe a state s as a vector $\langle c_1, \ldots, c_n \rangle$ containing the values of its variables and constants $v_1, \ldots, v_n$ in a fixed order. A sequence is described by $\langle \ldots \rangle$ and we use the notation $\{| \ldots |\}$ for multisets. The data structures of all variables in a state are analysed, and for each deferred set element, all possible paths are calculated that lead to that element within those data structures. We obtain a multiset of paths. Now, the marker is the set of variable-value pairs, where each occurrence of a deferred set element in a value is replaced by the respective multiset of paths.

Example 3.30 We describe here an example taken from [40]. Let's have a machine with the deferred set $D = \{d_1, d_2\}$ and variables c and r, where $c \in D$ and $r \subseteq D \times D$. We take the state $s = \langle d_1, \{(d_1, d_2)\} \rangle$ and calculate its symmetry marker. The graphical representation of s is depicted in Figure 3.15. We have the variable value pairs $c \mapsto d_1$ and $r \mapsto \{(d_1, d_2)\}$. Let m be a function, that maps, depending on the state, to each element of a deferred set its multiset of paths. Then we have:

$$m_s(d_1) = \{| \langle c \rangle, \langle r, el, \mathit{left} \rangle |\}$$
$$m_s(d_2) = \{| \langle r, el, \mathit{right} \rangle |\}.$$

Replacing each deferred set element in s by its multiset of paths, gives us the symmetry marker for the state s:

$$\langle\ \{| \langle c \rangle, \langle r, el, \mathit{left} \rangle |\}\ ,\{|\ (\ \{| \langle c \rangle, \langle r, el, \mathit{left} \rangle |\}, \{| \langle r, el, \mathit{right} \rangle |\}\ )\ |\}\ \rangle.$$

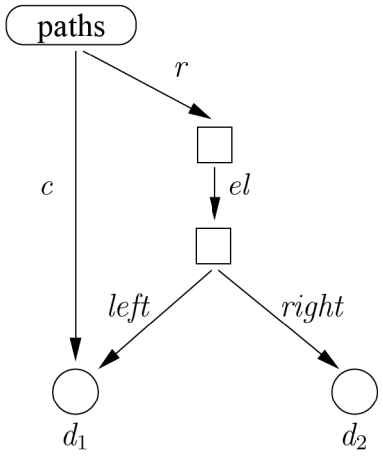


Figure 3.15: Graphical representation of state s

Note that a set, like the set of pairs $\{(d_1, d_2)\}$, is also written as a multiset in the marker. This makes the symmetry marker more precise.

We now consider the state $s_2 = \langle d_2, \{(d_2, d_1)\}\rangle$. This state is symmetric to s, by permuting the elements d_1 and d_2. Let's see if its symmetry marker agrees with that. The graphical representation of s_2 is depicted in Figure 3.16.

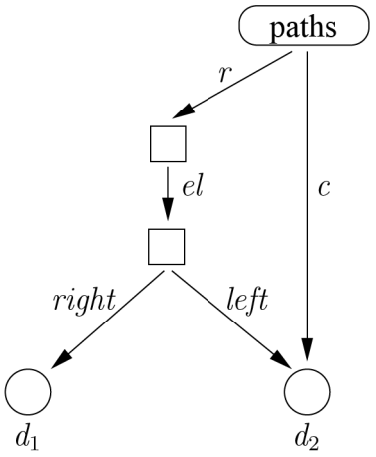


Figure 3.16: Graphical representation of state s_2

We have,

$$m_{s_2}(d_1) = \{|\ \langle r, el, right\rangle\ |\}$$
$$m_{s_2}(d_2) = \{|\ \langle c\rangle, \langle r, el, left\rangle\ |\},$$

and the symmetry marker for s_2 is

$$\langle\ \{|\ \langle c\rangle, \langle r, el, left\rangle\ |\}\ ,\{|\ (\ \{|\ \langle c\rangle, \langle r, el, left\rangle\ |\}, \{|\ \langle r, el, right\rangle\ |\}\ )\ |\}\ \rangle$$

which is the same as for s.

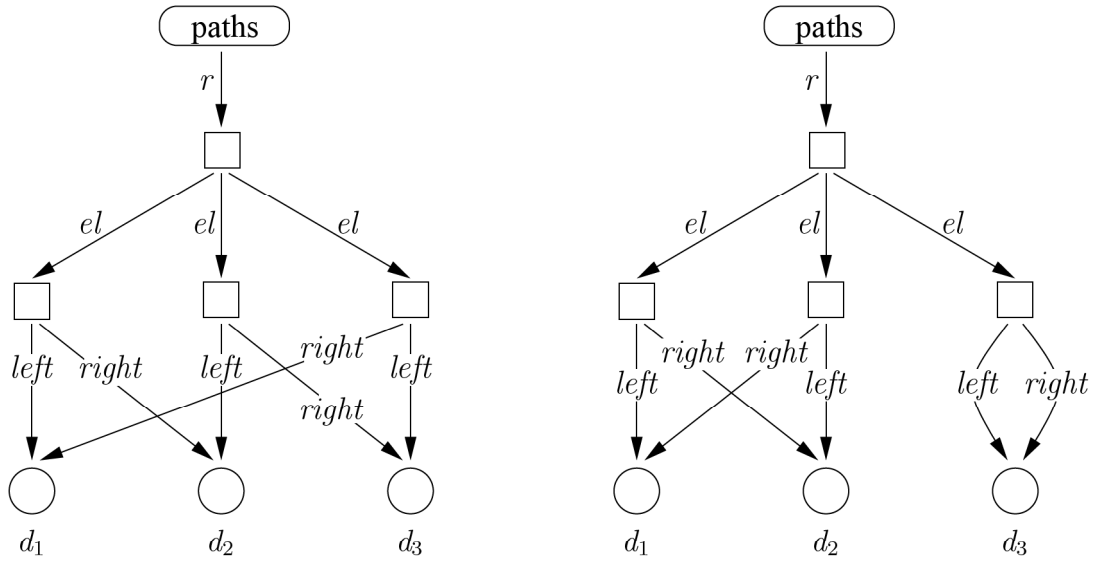
The graphical representation of the state here is only to give an understanding on how the symmetry markers are calculated. The method does not calculate state graphs as described in Section 2.4.

In Example 3.30, the symmetry markers of two states indicated correctly that the states are symmetric. This means that the state encountered second, would not need to be checked anymore and none of its successor states either in order to verify the model. Generally, states that are symmetric are also indicated as symmetric by their symmetry markers. However, it is possible to have non-symmetric states that have the same symmetry marker, as the following example shows.

Example 3.31 Let's have now a deferred set with three elements, $D = \{d_1, d_2, d_3\}$, and a relation $r \in D \times D$. We consider the non-symmetric states

$s_1 = \langle\{(d_1, d_2), (d_2, d_3), (d_3, d_1)\}\rangle$ and
$s_2 = \langle\{(d_1, d_2), (d_2, d_1), (d_3, d_3)\}\rangle$.

Figure 3.17 shows the graphical representation of both states and the multiset of paths for each deferred set element in both states.



$m_{s_1}(d_1) = \{|\ \langle r, el, left\rangle, \langle r, el, right\rangle\ |\}$
$m_{s_2}(d_2) = \{|\ \langle r, el, left\rangle, \langle r, el, right\rangle\ |\}$
$m_{s_3}(d_3) = \{|\ \langle r, el, left\rangle, \langle r, el, right\rangle\ |\}$

$m_{s_1}(d_1) = \{|\ \langle r, el, left\rangle, \langle r, el, right\rangle\ |\}$
$m_{s_2}(d_2) = \{|\ \langle r, el, left\rangle, \langle r, el, right\rangle\ |\}$
$m_{s_3}(d_3) = \{|\ \langle r, el, left\rangle, \langle r, el, right\rangle\ |\}$

Figure 3.17: Non-symmetric states with the same marker

We can see that the multisets are all the same for each deferred set element in both states. Observe that in both states, the relation r has the same number of pairs.

When replacing the deferred set elements with their respective multisets of paths, we yield the same marker for s_1 and s_2.

We've just seen that collisions can already occur in a very small example. A way to get around this problem, is to declare the deferred set as an enumerated set, but then of course there would be less symmetry in the model. There is obviously a tradeoff between speed and precision.

We want to apply now the symmetry marker approach to the personnel machine from Section 3.4.

Example 3.32 Let's look at two states of the personnel machine and compare their symmetry markers. Executing the of the operation $add\_to\_council(NAME1)$ after the initialization of the machine leads to the state:

$$\begin{aligned} s1 = (&staff\_council = \{NAME1\},\\ &status = \{(NAME1, single), (NAME2, single), (NAME3, single)\}), \end{aligned}$$

whereas executing the the operation $add\_to\_council(NAME2)$ leads to the state:

$$\begin{aligned} s2 = (&staff\_council = \{NAME2\},\\ &status = \{(NAME1, single), (NAME2, single), (NAME3, single)\}). \end{aligned}$$

We calculate the symmetry markers for both states. For s_1 we have:

$$\begin{aligned} &m_{s_1}(NAME1) = \{|\ \langle staff\_council, el\rangle, \langle status, el, left\rangle\ |\},\\ &m_{s_1}(NAME2) = \{|\ \langle status, el, left\rangle\ |\},\\ &m_{s_1}(NAME3) = \{|\ \langle status, el, left\rangle\ |\}. \end{aligned}$$

So the symmetry marker for s_1 is:

$$\begin{aligned} \langle\ &\{|\ \langle staff\_council, el\rangle, \langle status, el, left\rangle\ |\},\\ &\{|\ (\{|\ \langle staff\_council, el\rangle, \langle status, el, left\rangle\ |\}, single),\\ &\quad (\{|\ \langle status, el, left\rangle\ |\}, single),\\ &\quad (\{|\ \langle status, el, left\rangle\ |\}, single)\ \ |\}\ \rangle. \end{aligned}$$

For s_2 we have:

$$\begin{aligned} &m_{s_2}(NAME1) = \{|\ \langle status, el, left\rangle\ |\},\\ &m_{s_2}(NAME2) = \{|\ \langle staff\_council, el\rangle, \langle status, el, left\rangle\ |\},\\ &m_{s_2}(NAME3) = \{|\ \langle status, el, left\rangle\ |\}, \end{aligned}$$

and the symmetry marker for s_2 is:

$$\begin{aligned} \langle\ &\{|\ \langle staff\_council, el\rangle, \langle status, el, left\rangle\ |\},\\ &\{|\ (\{|\ \langle status, el, left\rangle\ |\}, single),\\ &\quad (\{|\ \langle staff\_council, el\rangle, \langle status, el, left\rangle\ |\}, single),\\ &\quad (\{|\ \langle status, el, left\rangle\ |\}, single)\ \ |\}\ \rangle. \end{aligned}$$

In a multiset the order of the elements is irrelevant. Consequently, both symmetry markers are the same, and indicate that the two states are symmetric. Therefore only one of those two states is evaluated further in the model checking algorithm.

Leuschel and Massart [40] proved that the method works correctly for falsification, and they also list under which conditions it is precise. Experiments show that there are very few collisions in practice.

Chapter 4

Empirical Results

In the following, we give the results of some runtime experiments. The tests were conducted under Debian Linux on a AMD Dual Core 3800+, 2GHz system with ProB 1.2.7.

We have compared the new approach using nauty for symmetry reduction, abbreviated as *nausym* in the tables and the following, with the other symmetry methods. These are the permutation flooding [39] method, abbreviated *flood*, described in Section 3.7.5 and the approximate symmetry marker method [40], abbreviated *hash*, also described in Section 3.7.5, as well as a former implementation of the canonical labelling algorithm in Prolog [56] abbreviated *canon* in the following. Runtime results presented in the respective articles of the methods may differ, since different computer architectures were used. As baseline we have also conducted experiments with ProB, where symmetry reduction was disabled, abbreviated with *wo*.

We have used a variety of B specifications in our experiments. The machines are mostly the same as in "Efficient approximate verification of B via symmetry markers" [40], in order to give a better comparison to previous work. We want to describe each machine briefly before we present and analyse our results. Each machine is also given in Appendix 5.

1. The machine *scheduler*0 is a specification of a process scheduling system on a single resource taken from [37]. The process identifiers are elements of a deferred set, and the state of each process can be either *idle*, *ready* or *active*. Since a single resource is modelled, the invariant states that there can be at most one active process.

2. *scheduler*1 is a refinement of *scheduler*0 and also taken from [37].

3. *RussianPostalPuzzle* specifies a cryptographic puzzle [20]. The problem of the puzzle is sending a valuable item in a box safely by post. The box is passed by the postal workers to the recipient only if the postal worker does not have a key to open it and knows that there won't be a key sent in the post. The keys and padlocks in the specification are modelled by a deferred set.

4. *USB_4Endpoints* is a specification of a USB transfer protocol designed by ClearSy, a French company that specialises in developing safety-critical systems. The protocol is described as a tuple (host, device endpoint, transfer type), while the invariant allows only one transfer at a time for any endpoint. The endpoints are described by a subset of natural numbers, and a deferred set models the set of all possible transfers.

5. *TokenRing* models a network ring of servers. A server that requests to send needs to get the token first, before it can do so. The token moves around all servers unless one server is in the critical state of sending. The set of servers is modelled with a deferred set.

6. *Dining* is a model of the *Dining Philosophers Problem*. The classical problem has five philosophers on a round table with one fork between two philosophers. In order to eat any philosopher needs two forks. The model describes the philosophers and the forks with deferred sets. Two constant bijections between the philosophers and the forks assign each philosopher a fork to his left, and to his right, respectively. The model generalises the classical problem in that it allows a bigger or smaller table of philosophers (minimum two), or even several round tables that are independent from each other.

7. *Towns* is a model from [49]. It keeps track of motorways being built to link the towns to each other. The set of towns is given as parameter and the roads are described as a relation between the set of towns.

4.1 Analysis of the results

4.1.1 Comparison between Symmetry Reduction Methods for B

We have partitioned the machines into two tables: Table 4.1 contains those experiments where all symmetry reduction methods examined the same number of states, column *states (sym)*, and Table 4.2 contains those experiments where the symmetry reduction methods differed in the number of states examined. Hence, Table 4.1 contains only a single column for the number of states model checked by all symmetry methods, whereas Table 4.2 contains one column per method. In our experiments, we have varied the cardinality of the deferred sets in order to study the effect of symmetry reduction with increasing size of the deferred sets. The cardinality used is shown in the first column labelled with *card*. Runtimes are expressed in seconds; Table 4.1 contains columns for the speedup of Symmetry Reduction using NAUTY compared the other methods, where a value above 1 means that *nausym* is faster.

We also took the machines *scheduler*0, *scheduler*1, *RussianPostalPuzzle* and *TokenRing* and put the runtimes for each method in dependence of the cardinality

Table 4.1: Empirical Results I

			Runtimes of Symmetry Methods					Speedup			
card	states (wo)	states (sym)	wo	flood [39]	hash [40]	canon [56]	nausym	wo	flood [39]	hash [40]	canon [56]
scheduler0											
2	16	10	0,04	0,03	0,03	0,05	0,03	1,3	0,9	0,9	1,4
3	55	17	0,23	0,08	0,08	0,16	0,09	2,5	0,9	0,8	1,8
4	190	26	1,10	0,24	0,17	0,60	0,21	5,4	1,1	0,8	2,9
5	649	37	5,10	0,94	0,33	2,76	0,41	12,4	2,3	0,8	6,7
6	2188	50	23,08	6,11	0,61	17,12	0,76	30,5	8,1	0,8	22,7
7	7291	65	115,01	55,07	1,00	139,05	1,28	89,9	43,0	0,8	108,7
scheduler1											
2	27	14	0,06	0,04	0,03	0,26	0,05	1,2	0,7	0,7	5,12
3	145	29	0,46	0,12	0,10	1,29	0,16	2,0	0,8	0,7	8,08
4	825	51	3,36	0,43	0,25	6,27	0,39	8,6	1,1	0,6	16,03
5	5201	81	27,13	2,28	0,54	35,56	0,84	32,4	2,7	0,7	42,42
6	37009	120	333,82	35,71	0,96	674,26	1,61	207,9	22,2	0,6	419,93
7	-	169	*	*	1,66	*	2,80	-	-	0,6	-
10	-	386	*	*	6,51	*	11,37	-	-	0,6	-
15	-	1041	*	*	35,67	*	67,08	-	-	0,5	-
20	-	2171	*	*	141,96	*	257,16	-	-	0,6	-
RussianPostalPuzzle											
1	15	15	0,03	0,03	0,03	0,07	0,06	0,5	0,5	0,5	1,1
2	81	48	0,21	0,14	0,13	0,42	0,25	0,8	0,6	0,5	1,7
3	441	119	1,40	0,54	0,43	2,20	0,81	1,7	0,7	0,5	2,7
4	2325	248	9,36	2,37	1,14	11,53	2,23	4,2	1,1	0,5	5,2
5	11985	459	64,52	15,91	2,58	63,97	5,57	11,6	2,9	0,5	11,5
USB_4Endpoints											
1	29	29	0,21	0,21	0,23	24,67	1,14	0,2	0,2	0,2	21,7
2	694	355	10,69	5,80	7,74	547,17	21,97	0,5	0,3	0,4	24,9
3	16906	3013	1533,54	265,19	208,40	*	297,43	5,2	0,9	0,7	-

* means test has been cancelled or not done, because of excessive runtime

Table 4.2: Empirical Results II (where the methods calculate different number of states)

	number of states						Runtime of Symmetry Methods			
card	wo	flood [39]	hash [40]	canon [56]	nausym	wo	flood [39]	hash [40]	canon [56]	nausym
Token Ring										
2	35	19	19	35	19	0,07	0,06	0,05	0,11	0,07
3	295	60	60	148	60	0,49	0,19	0,16	0,65	0,22
4	3097	174	141	646	174	6,57	1,44	0,46	6,47	0,86
5	38521	480	278	2248	480	175,61	42,90	0,97	61,98	2,96
6	-	-	495	8460	1252	*	*	2,09	921,79	9,90
7	-	-	816	-	3160	*	*	4,27	*	33,86
Dining										
2	21	8	7	11	8	0,07	0,05	0,04	0,06	0,05
3	337	13	11	29	13	1,50	0,18	0,08	0,27	0,10
4	17713	48	17	165	48	145,53	18,13	0,15	3,39	0,54
Towns										
2	17	11	11	11	11	0,34	0,20	0,21	0,24	0,21
3	513	105	105	105	105	67,78	13,45	13,73	15,68	13,93
4	65537	3045	3011	-	3045	*	1721,73	1748,45	*	1732,03

* means test has been cancelled or not done, because of excessive runtime

of the respective deferred set into diagrams, see Figure 4.1 to Figure 4.4. We chose a logarithmic scale for the runtime.

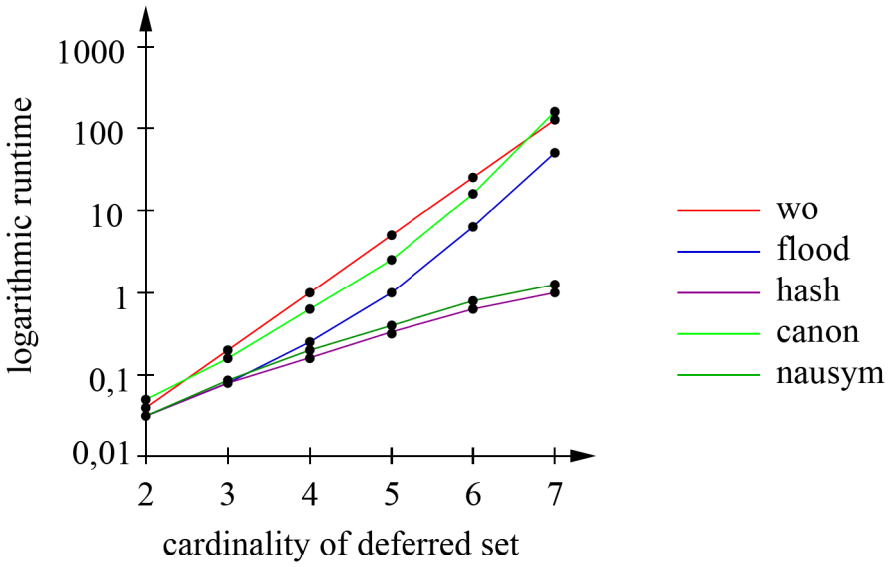


Figure 4.1: Runtimes for scheduler0

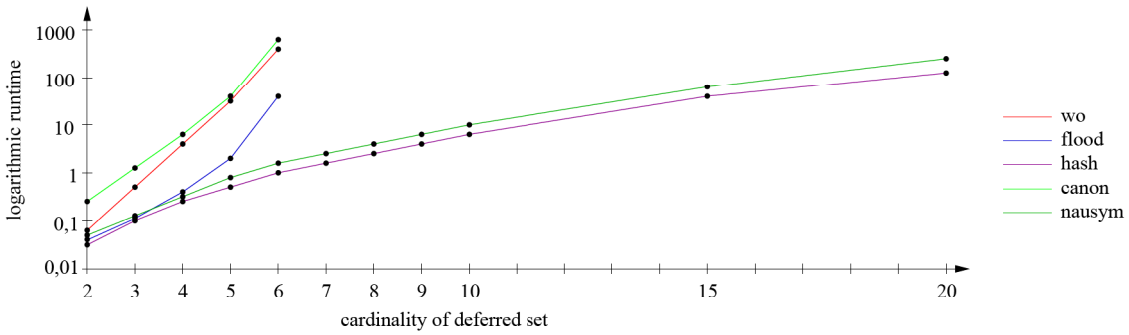


Figure 4.2: Runtimes for scheduler1

nausym* vs. *wo: We can see that for very small cardinalities of the deferred set, the runtimes for each symmetry method or even without symmetry do not differ much in most cases. Except for machines like *USB_4Endpoints*, where the runtime exceeds reasonable timeperiods already for cardinality greater than three. The greater the cardinality of the deferred sets, the greater is also the speedup of *nausym*, compared to using no symmetry (see speedup wo). For a cardinality of five, using *nausym* is already more than ten times faster for all four machines.

nausym* vs. *canon: We can see that in every instance, our new implementation is more efficient than *canon* from [56]. Sometimes the difference is dra-

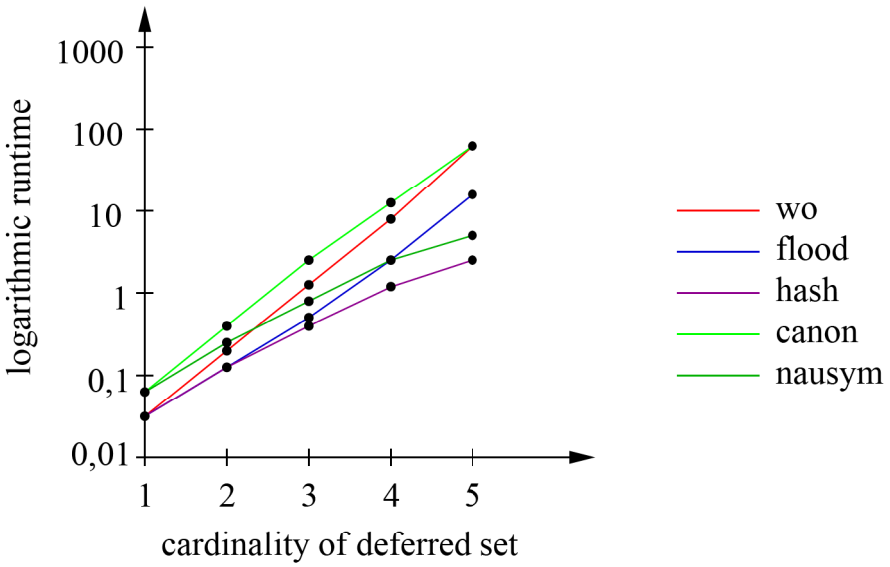


Figure 4.3: Runtimes for RussianPostalPuzzle

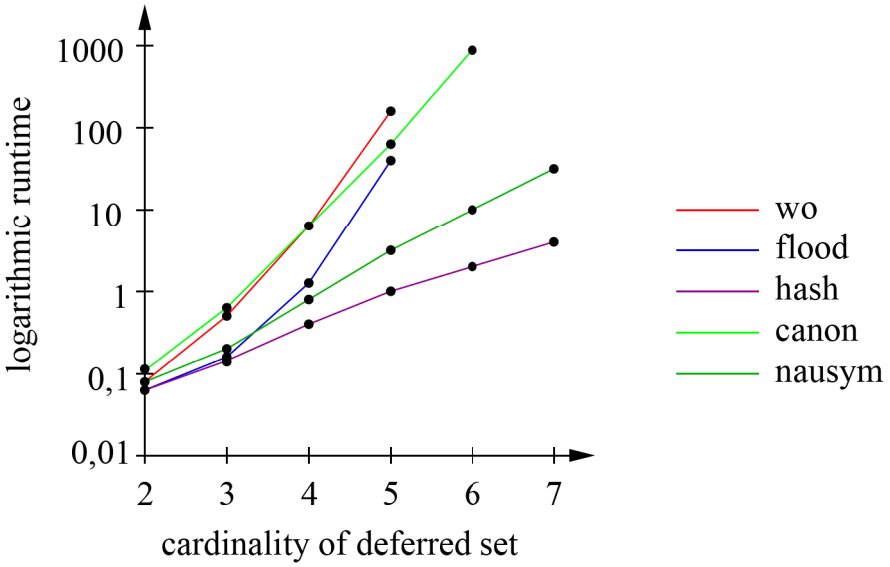


Figure 4.4: Runtimes for TokenRing

matic, exceeding two orders of magnitude. Recall that the method *canon*, has the same mathematical foundation as our new method: The B-states are translated into vertex- and edge-coloured graphs and a canonical form is computed, to decide if the respective state has been computed already or not - see Sections 2.4 and 3.3, or [56]. However, for *canon* the standard canonical labelling algorithm was extended to vertex- and edge-coloured graphs, and implemented in Prolog. For *nausym*, we

transform the state graphs into vertex-coloured graphs as explained in Section 3.4, and then apply NAUTY. When we first applied symmetry reduction with canonical labelling in the approach *canon*, it was disappointingly slow and, therefore, the question was if this was due to the method based on graph isomorphism, or the implementation. Now we can answer this question and say that the disappointing runtime results were due to the implementation. Probably part of the blame goes to the implementation in Prolog, the other part is that the canonical labelling algorithm itself in *canon* did not include many of the optimisations and years of refinement that make NAUTY such an effective tool.

nausym* vs. *flood: The method *flood* employs a different approach to symmetry reduction, called permutation flooding, see Section 3.7.5 and [39]. For small cardinalities, it behaves similar - sometimes even slightly better, than *nausym*. However, for higher cardinalities the flooding of the state space often induces too big an overhead; meaning that the new approach is generally faster and much more scalable, compare, e.g., the runtimes for *scheduler*1 or *TokenRing*. The runtime diagrams in Figure 4.1 to Figure 4.4 state this behaviour graphically.

nausym* vs. *hash: The only method which outruns symmetry reduction using NAUTY is the symmetry marker method (*hash*), see 3.7.5 and [40]. Indeed, it is the only other method that scales well for all examples, see Table 4.1 and Table 4.2. The figures, especially Figure 4.1, show how well both methods scale with increasing cardinality of the deferred set. The runtime curve of *nausym* is just slightly above the runtime graph of *hash*. The two curves run almost parallel, even for higher cardinalities on the logarithmic scale. The method *hash* is about twice as fast as *nausym* in all four examples from Table 4.1 for most cardinalities. Although it is precise for these examples, symmetry reduction with symmetry markers is generally not an exact method; it uses a hash function that does not guarantee that non-isomorphic states are always detected as non-isomorphic. That means that error states of a machine could potentially be missed. This can be seen in Table 4.2: the method *hash* often computes less states than required to exhaustively model check the B machine. In fact, Table 4.2 shows that the number of states computed by each method differs. Permutation flooding and *nausym* inspect the same number of states, see Figure 4.5, which is the minimum number for a correct model checking.[1]

4.1.2 Analysis of the State Space Reduction

We have now compared our approach with the other symmetry reduction methods for B, and, by contrast, using no symmetry reduction at all. We now want to discuss the state space reduction when symmetry is used. In those cases where the symmetry marker method is exact, all symmetry methods have the same number of states to model check. The permutation flooding approach is a special case, because even though the number of states that need to be model checked is the same as for

[1]The Prolog implementation of the canonical labelling algorithm does not detect symmetric states that arise during the constant setup phase, therefore more states need to be model checked.

the other symmetry methods, it generates states through permutation of elements of the deferred sets, so that the state space itself is not reduced. For the *scheduler*0 example we have depicted in Figure 4.5, the size of the state space, and in Figure 4.6 the number of transitions, in dependency of the cardinality of the deferred set. For this example, the symmetry marker approach is exact, so we have the same graph for the three methods denoted as *hash*, *canon* and *nausym*. As explained above, the size of the state space for permutation flooding is the same as for model checking without symmetry, so there is one graph for both cases together. The number of states are presented on a logarithmic scale. We can see clearly that the state space grows exponentially with the size of input, i.e. the cardinality of the deferred set. When using either of the classical symmetry reduction methods, the resulting graph can be approximated by a cubic polynomial. In most cases, the growth of the state space can be approximated by a low degree polynomial function, such as quadratic or cubic. Of course, we have to keep in mind that some models do not allow much symmetry reduction. In those cases, a different approach is recommended.

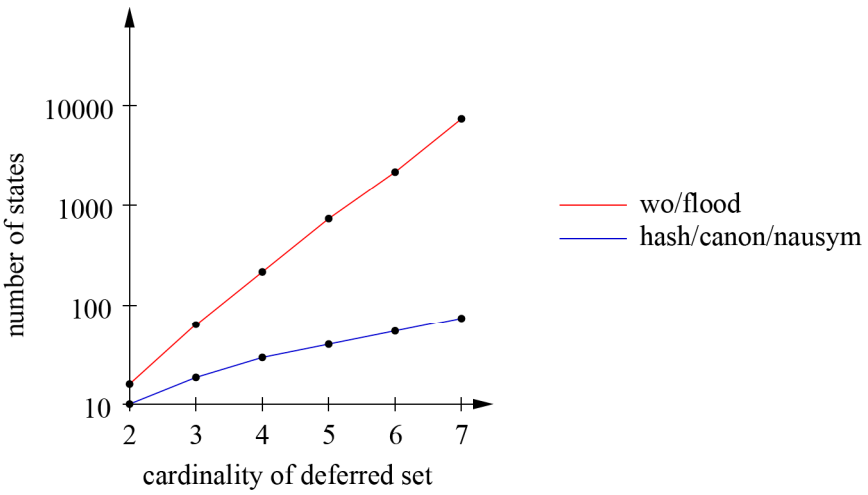


Figure 4.5: Size of State Space for *scheduler0*

4.1.3 Analysis of the Graph Canonicalisation Time

Symmetry reduction using NAUTY showed good results compared to the other symmetry reduction methods in ProB, and we asked ourselves if this is generally the case. After using the method for some time within ProB, we found a model that takes longer to model check using symmetry reduction with NAUTY than using no symmetry at all. This machine models the *TicTacToe* game, see Appendix B.8.

The following two reasons could explain this behaviour: The first cause is that some models do not make much use of deferred sets, which are essential for the occurrence of symmetry. Consequently, the reduction of the state space is not that

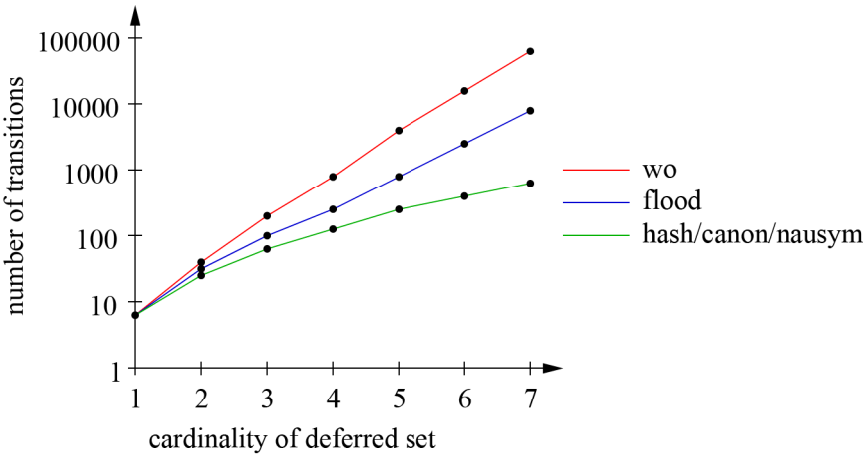


Figure 4.6: Number of transitions of *scheduler0*

great, so that the overhead for the calculation of the canonical form becomes too massive compared to the achieved state space reduction. The second reason for bad performance of *nausym* lies in the size and structure of the graphs constructed from B-states. When calculating the canonical form, the algorithm starts with an initial partition of the vertex set according to the colouring of the vertices. The smaller the cells in the initial partition are, the fewer calculations are needed to find the canonical form. However, if the initial partition contains a very large cell, or several large cells, then the calculation of the canonical form can take a considerable amount of time and space in memory, because of the enlarging search tree, which can slow the computation down even further, depending on the available system memory. Since a canonical form needs to be calculated for every encountered state during model checking, the computation time can add up very quickly to a considerable amount.

In the case for the *TicTacToe* model, we encountered timeouts during the calculation of the successor states for a particular state. This is also when NAUTY is called, so that ProB can decide which of those states needs to be evaluated further. The timeout we had set in the ProB preferences was ten seconds. We set this timeout to make model checking feasible, overall. In Table 4.3[2] we can also see that the other symmetry reduction methods took more than two minutes to model check. Note that we have chosen not to treat deadlocks as errors, because the end of each game is a deadlock - but not an error.

We figure that this model is not so well-suited for symmetry reduction. When we analysed a model of the TicTacToe game that expresses less symmetries of the game in that the rotational symmetries are omitted. The runtime results for model checking of this model are much better, see Table 4.4. We used ProB's *ComputeCoverage*

[2]For computing the results we used the same system as before but with ProB 1.3.1 which has some new optimisations compared to ProB 1.2.7

Table 4.3: Results of the *TicTacToe_Sym* model

	wo	flood [39]	hash [40]	nausym [51]
STATES	5480	5480	1401	*
TIME in sec	518,6	138,3	139,2	TIMEOUT

Table 4.4: Results of the *TicTacToe_SimplerSym* model

	wo	flood [39]	hash [40]	nausym [51]
STATES	5480	5480	1450	1450
TIME in sec	97,6	26,6	26,7	29,1

function for both models to find out the number of states that have been visited. Both models encounter the same number of states when no symmetry is used. So the overall state space of the second model is not smaller, compared to the first one, but the model checking time is greatly reduced for each method. Now the time needed by *nausym* is less that three seconds more than that for *hash*, which is within a similar scale compared to our other experiments. The reason for the difference in the model checking time we found is the complexity of the states. The graphs representing the states of the *TicTacToe_Sym* model turned out to be much more complex than those of the *TicTacToe_SimplerSym* model. An example of such a graph is depicted in Figure 4.7. This graph has still labels, i.e. colours on the edges, so the only vertex-coloured graph, which is handed to NAUTY, is even larger. Surprisingly the model that describes all the symmetries of the TicTacToe game is much slower to model check with any of ProBs symmetry reduction methods. This is due to the complex states, which have to be represented in some kind for every method by the model checker. The example shows very well that the time needed for model checking in general, depends very much on the modelling.

4.1.4 Size of State Graphs

In our experiments, we also evaluated how big the graphs representing states are for our examples, see Appendix C. With state graphs, we mean here those that are handled by NAUTY. That is, the original vertex- and edge-coloured state graph produced by ProB has been transformed to a vertex-coloured graph as described in Section 2.4. In Figure 4.8 in the left graph, we depicted the results for the machine *scheduler0*, for increasing cardinality of the deferred set. It shows the number of graphs in dependency of the number of vertices for an individual graph. The state

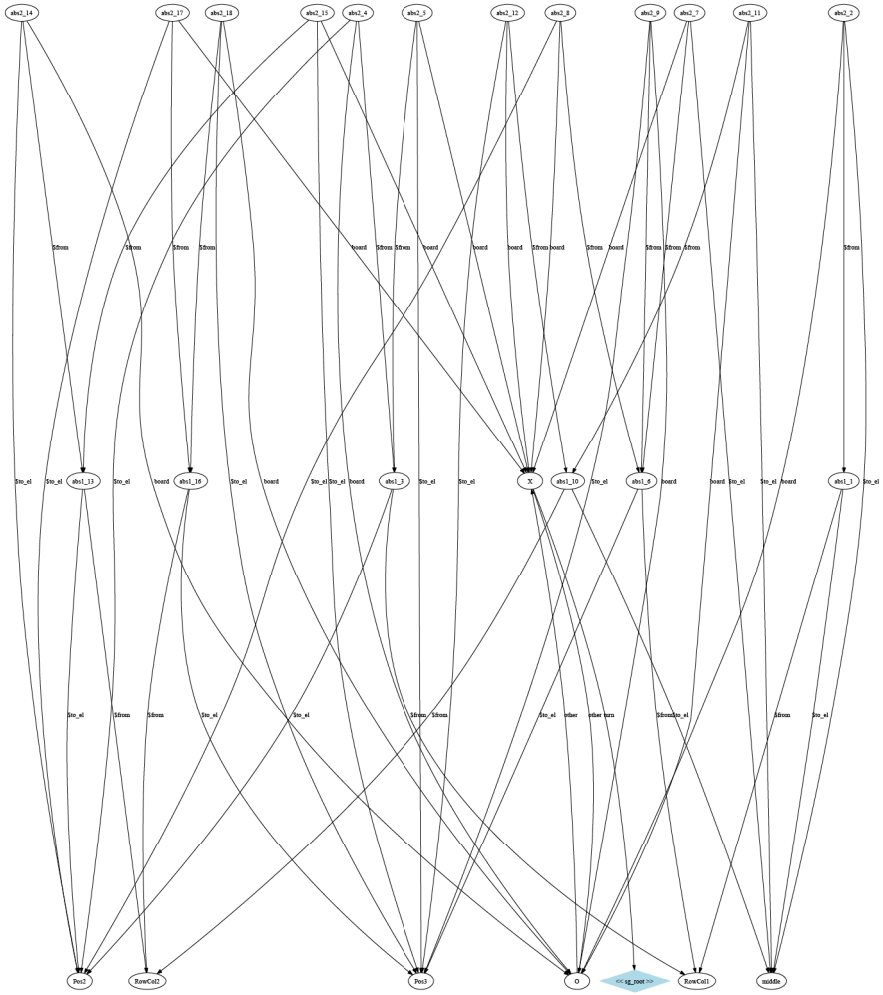


Figure 4.7: One state graph of *TicTacToe_Sym* model produced by ProB

graphs for this example are not very big, and the number of graphs for cardinality seven are still reasonable. Our runtime results from Table 4.1 suggest that nauty handles those graphs very easily. The right graph of Figure 4.8 shows the results for various machines with cardinality three of the deferred set, if not stated otherwise in the caption of the graph. Although the graphs are a bit larger, and many more states are encountered for some machines, the runtimes are still very good. For example, for the *USB4_Endpoints* machine, there are more than 800 graphs with 45 vertices

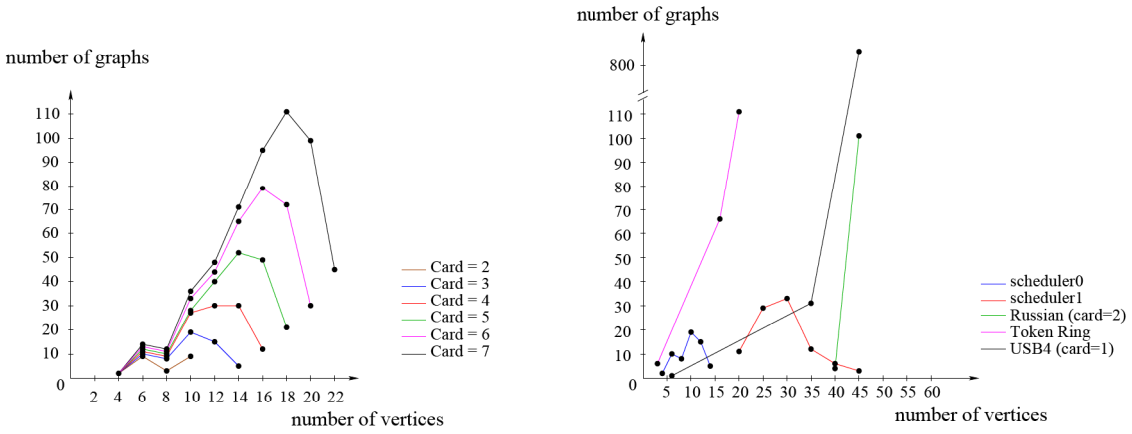


Figure 4.8: Size of state graphs for scheduler0 (with varying cardinality) and for varying B machines

each, but the runtime is still only 1.14 seconds, see Table 4.1. Note that all runtime results also include ProB's model checking and interpretation of the B machine. From our experience, we deduce that most models produce state graphs with less than 100 vertices, which can be handled well by nauty. The model presented in the Section 4.1.3 seems to be a rare exception. The layered structure of the graphs handed to nauty might also help nauty to calculate a canonical form. Indeed, in each layer, the vertices are coloured differently. That means the cells in the initial partition are, at most, as large as the number of vertices in a single layer. The smaller the cells in the initial partition are, the better the performance of nauty.

Chapter 5

Conclusions and Future Work

This thesis continues the work in symmetry reduction via graph canonicalisation, for model checking in B. The pioneering work on this topic was done by E. Turner [55]. His work is a proof of concept and made a further exploration of the method interesting. Now, this thesis takes this foundation and adds an important building block, by using NAUTY [42] for calculating canonical labels of graphs representing B-states. NAUTY has been developed and improved by its authors over years, and its algorithms for graph canonicalisation use optimisations, that haven't been used in the implementation of the algorithm by E. Turner. The use of this efficient tool has led to a breakthrough in terms of practicality. Symmetry reduction via canonical form using NAUTY is now the first choice of precise symmetry reduction methods in the ProB toolset. Empirical results presented in this thesis show the vast improvement to the previous implementation of the canonical labelling algorithm. Consequently, the old implementation is no longer available in release 1.3.0 (or later) of ProB. Our technique is generally much more effective than permutation flooding [39] and therefore the preferred choice. The only symmetry reduction method that is still faster, is the approximate method using symmetry markers by Leuschel and Massart [40]. However our technique scales equally well for most models, while being fully precise. In cases where symmetry reduction using NAUTY for the graph canonicalisation takes too long, we can still choose another symmetry reduction method, or even go the other way round and use the approximate but fast Symmetry Marker method first - and, only in cases where this method found no errors, then use symmetry reduction with NAUTY to make sure that a hash collision did not lead to a wrong result.

We also want to mention here, that since Leuschel and Plagge have developed a translation of Z models into B in [47], and an LTL model checker for ProB in [41], so our method is also applicable to Z and for LTL model checking.

In research, there is always margin for improvement, or some unexplored idea that can be beneficial. We want to discuss here a few ideas to be investigated in future work.

An important part of the symmetry reduction method described in this thesis,

is the translation of vertex- and edge-coloured graphs to vertex-coloured graphs, so that they can be fed to NAUTY. In the following, we describe an idea on how to reduce the size of the resulting vertex-coloured graphs. In Section 3.4 we described how the vertex- and edge-coloured graphs representing states can be transformed to only vertex-coloured graphs, so that they can be fed to NAUTY. Each edge colour has been represented by a layer of duplicated vertices of the original graph. If a model has many data structures, so that the graphical representation of a state has many labels on the edges, then the respective vertex-coloured graph has the same number of layers. This leads to a fairly large vertex-coloured graph, in comparison to the vertex- and edge-coloured graph constructed from a state. In order to reduce the size of those graphs the NAUTY user's guide [42] suggests placing several colours in one layer. We want to explain the idea, together with an example.

Example 5.1 Let's take the following graph with only four vertices, but three different colours on the edges.

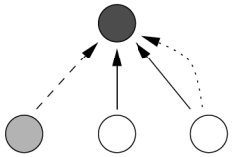


Figure 5.1: A vertex- and edge-coloured graph

With the transformation of Section 3.4, the layered vertex-coloured graph would look like:

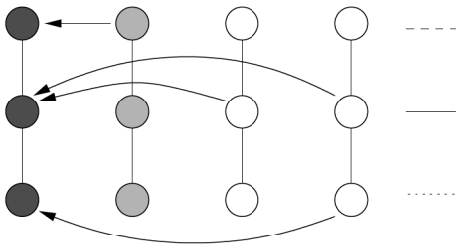


Figure 5.2: The transformed vertex-coloured graph

Each edge-colour has its separate layer, as indicated in Figure 5.2. Now, given that each colour is internally represented by an integer, we take the binary representation of that integer as indication in which layers a colour is represented. The least significant bit is related to the lowest layer, i.e. layer 1, and so on. That means for our example that, colour 1 gets represented in layer 1, colour 2 in layer 2 and colour 3 in layer 1 and 2. Consequently the vertex-coloured graph has now one layer less.

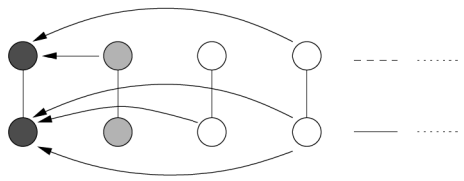


Figure 5.3: Optimised vertex-coloured graph

Generally, if we have k colours, we need $\lfloor log_2 k \rfloor + 1$ layers instead of k layers as before. Implementing this transformation would achieve quite a considerable saving in the size of the graph. On the other hand, we cannot say if it would save any significant computation time, since the graphs are more dense in the individual layers.

Another idea for improvement could be to replace NAUTY with another tool. Related tools are SAUCY [16] and BLISS [32]. SAUCY though does not compute a canonical label, so it does not serve our purpose. BLISS on the other hand does, and it performs particularly well for large sparse graphs. Large sparse graphs are graphs with relatively few "$1's$" in their adjacency matrix. For these graphs BLISS clearly outperforms NAUTY, see [32]. However BLISS can also handle other types of graphs well. Its performance for dense and highly regular graphs, which occur often in combinatorial problems, is similar or better than the performance of NAUTY. We need to investigate how well BLISS would perform with our graphs. The vertex-coloured graphs are usually not very dense, and have some structure due to their layered representation. Both indicates that BLISS would handle our graphs very well.

A very recent idea is to make a slight change to the model checking algorithm, using symmetry reduction, see Section 3.5. The current implementation does calculate the canonical form for all successor states of a state s. Those states that are symmetric to some previously encountered state are not considered further, but also not saved in any way. That means, if another path in the state space leads to such a state again, then its canonical form needs to be calculated again, too. This could be avoided by saving every encountered state in the same way as it is done for the ordinary model checking algorithm without symmetry. There, a hash value is computed and saved for each state. This is done so that the algorithm detects a loop of transitions in the state space and does not run endlessly in such a loop.

When there is a canonical form calculated for each state before it is evaluated further, then this also prevents the model checking algorithm from going into an endless loop. Therefore, saving another hash value for each encountered state isn't necessary. However, if we do it anyway, then we could save calculating the canonical form several times for one state. This means, rather than calculating a canonical form for essentially each transition in the reduced state space, we have to do so only once for each encountered state. The overhead for calculating and saving a hash

value for each state, could pay off for models that have many transitions, compared to their number of states. Indeed, the machine *USB4_Endpoints*, which we used for our empirical results, is such a model. A first experiment showed that model checking of this particular model was five times faster. We will have to evaluate this idea in future work, before we can draw general conclusions.

The last suggestion we wish to make on future work, is on translating states into graphs. In Section 2.4 we explained the currently implemented algorithm. This algorithm is just a proof of concept, and there might be better representations of states as graphs. We want to describe here one possible improvement.

The current algorithm creates intermediary vertices for nested data structures. Those vertices all get the same colour, so far. Considering that NAUTY performs best when the initial partition of a graph has small cells, this is probably not the best solution. Each intermediate vertice comes from a certain data structure. Taking this into account we could group the vertices together accordingly and give each group a different colour. This way, symmetries are still preserved, but the cells of the initial partition are smaller.

Next to improving the algorithm, we also remain to prove that our algorithm is actually correct. That means, our implementation translates states into graphs such that states are symmetric, if and only if the respective state graphs are isomorphic. In order to show this, we intend to specify the algorithm in B and then refine this specification to an implementation. If we prove the correctness of each step of refinement with, for example, Atelier-B [53], then we prove also the correctness of the implementation. This implementation might be very different from our current algorithm.

Appendix A

Code of Interface between NAUTY and ProB

We list the code here that is called by ProB. Note that the code has been edited to fit on the page. Code that was only used for experimenting and debugging has been mostly omitted.

A.1 Functions called by ProB

```
#include "interface.h"

void prob_init(void);

void prob_set_number_of_colours(int num_of_col);

void prob_start_graph(int number_of_nodes);

void prob_add_edge(int from, int to);

void prob_set_colour_of_node(int node, int colour);

int prob_exists_graph(void);

void prob_free_storage(void);

int global_n;
int global_m;

graph *global_g;
graph *global_canong;
int global_number_of_colours;
int *global_cell_sizes;
int *global_is_set;      // to mark which node has got a colour
int *cell_list;
```

```
int tree_node_counter;
struct Node *store_tree; // to initialize with NULL pointer (needed for insert())

/* variables to check the program flow
 */
short is_storage_free = TRUE;
short is_init = FALSE;
short is_set_num_colour = FALSE;
short is_start_graph = FALSE;


void prob_init(void){

   /* start with new state space only if old states (graphs) are
    * removed && is_set_num_colour
    */
   if(is_storage_free){

      /* initialisation of all global variables except
       * global_number_of_colours = 1; set extra with
       * prob_set_number of colours
       */
      global_g = NULL;
      global_canong = NULL;
      global_n = 0;
      global_m = 0;
      cell_list = NULL;
      global_is_set = NULL;
      tree_node_counter = 1;     // start counting number of stored graphs with 1
      store_tree = NULL;

      /* size must be at least n, the maximum number of cells the number of cells
       * of each graph is not used here as array size but MAXN, since the memory
       * for  global_cell_sizes is only once allocated, if this is changed in
       * future versions, also functions that use the size of the array
       * global_cell_sizes must be changed
       */
      global_cell_sizes = malloc(sizeof(int) * MAXN);
      if(global_cell_sizes){     // size of each cell is set 0
         init_cell_sizes(global_cell_sizes, MAXN);
      }
      else{
         printf("Error in prob_init: memory could not be allocated! \n");
         return;
      }

      is_init = TRUE;            // successful initialisation
      is_storage_free = FALSE;
   }
   else{
      printf("ERROR in prob_init: old storage space needs to be freed, before"
             "initialisation! Run function prob_free_storage and"
```

```
            "prob_set_number_of_colours first \n");
   }
}


/* call from ProB and get the number of number of colours, if parameter <= 0 or
 * > MAXN then prints error message and returns
 */
void prob_set_number_of_colours(int num_of_col){

   global_number_of_colours  = get_number_of_colours(num_of_col);
   if(global_number_of_colours > MAXN){
      printf("Error: number of colours is too big!");
      printf("Rerun prob_set_number_of_colours with a number <= %d!", MAXN);
      return;                   // number of colours not set
   }

   is_set_num_colour = TRUE;
}


void prob_start_graph(int number_of_nodes){

   /* all variables need to be initialized before use
    */
   if(is_init && is_set_num_colour){

      /* if MAXN is not set by nauty then n can be up to INT_MAX
       */
      if( (MAXN > 0 && number_of_nodes <= MAXN) ||
          (MAXN == 0 && number_of_nodes <= INT_MAX) ){
         global_n = number_of_nodes;
         global_m = (global_n + WORDSIZE - 1) / WORDSIZE;

         /* free global_g since it may be used before (NULL if not)
          */
         free(global_g);
         global_g = start_graph(global_n);

         /* allocate memory for canong like for g as is will contain an
          * isomorphic graph to g
          */
         free(global_canong);
         global_canong = start_graph(global_n);

         /* size of each cell is set to 0
          */
         init_cell_sizes(global_cell_sizes, MAXN);

         /* initialize cell_list for each graph new -> free cell_list
          * each cell can contain max. n vertices, which are nat. numbers
          */
```

```
        free(cell_list);
        cell_list = malloc( sizeof(int) * global_number_of_colours * global_n );

        if(cell_list){
           init_cell(global_n, global_number_of_colours, cell_list);
        }
        else{
           printf("Error in prob_start_graph: no memory allocated for"
                  "cell_list. Run prob_set_number_of_colours and then"
                  "prob_init. \n");
        }

        free(global_is_set);
        global_is_set = malloc( (sizeof(int)) * global_m);

        int i=0;

        for(i=0; i<global_m; i++){     // all vertices have no colour yet
           global_is_set[i]=0;
        }

        /* entries that exceed the number of vertices are set to 1
         */
        for(i=global_n; i<global_m * WORDSIZE; i++){
           set_node(i, global_is_set);
        }

        is_start_graph = TRUE;         // successful memory allocation for graph
     }
     else{
        printf("Error in prob_start_graph: number_of_nodes %d is too big \n",
               number_of_nodes);
     }
  }
  else{
     printf("ERROR in prob_start_graph: storage space needs to be initialised"
            "before use! and number of colours has to be set run function"
            "prob_init and prob_set_number_of_colours first \n");
  }
}


void prob_add_edge(int from, int to){

  /* memory for graph needs be allocated before an edge can be added
   */
  if(is_start_graph){
     add_edge(from, to, global_g, global_n); // global_n= number_of_nodes
  }
  else{
     printf("ERROR in prob_add_edge: memory for graph needs be allocated before"
            "an edge can be added! Run function prob_start_graph first. \n");
```

```
   }
}


void prob_set_colour_of_node(int node, int colour){

   if(is_start_graph && is_set_num_colour ){
      if(colour < global_number_of_colours){
         set_colour_of_node(global_n, colour, node, cell_list, global_cell_sizes);
      }
      else{
         printf("Error in prob_set_colour_of_node: num. for colour (%d) too big!"
                "Number must be < %d\n", colour, global_number_of_colours);
      }
   }
   else{
      printf("ERROR in prob_set_colour_of_node: graph does not exist or number of
     "colours is not set! Run function prob_start_graph and"
             "prob_set_number_of_colours first \n");
   }
}


/* ProB calls this function to test, if a state has been encountered before.
 * The NAUTY library is called to calculate the canonical label of the respective
 * graph.
 */
int prob_exists_graph(void){

   if(is_start_graph){

   int exists = -1;                        // set to error value

   /* The nauty parameter m is a value such that an array of m setwords is
    * sufficient to hold n bits.  The type setword is defined in nauty.h. The
    * number of bits in  a setword is WORDSIZE, which is 16, 32 or 64.
    * Here we calculate m = ceiling(n/WORDSIZE)
    */
   int m = (global_n + WORDSIZE - 1) / WORDSIZE;

   /* lab holds the initial order of the vertices and ptn holds where a cell ends
    */
   int lab[MAXN], ptn[MAXN], orbits[MAXN];

   static DEFAULTOPTIONS_GRAPH(options);
   statsblk stats;
   setword workspace[100*MAXM];            // need more workspace for better speed

   /* Default options are set by the DEFAULTOPTIONS_GRAPH macro above.
    * Here we change those options that we want to be different from the
    *  defaults.  writeautoms=TRUE causes automorphisms to be written.
    */
```

```
options.getcanon = TRUE;
options.digraph = TRUE;

if(global_number_of_colours  > 1){
   /* variables lab and ptn need to be set manually
    */
   options.defaultptn = FALSE;         // set initial partition manually

   /* check if every node has a colour;  give colour 0 if not
    */
   int i = 0;
   while(i<m){
      if(global_is_set[i] != -1){      // -1 = 111... in binary
         printf("not all vertices have a colour set! \n");
         printf("give colour 0 to vertices without colour! \n");
         int k= 0;
         for(k=0; k<global_n; k++){
            has_colour(k,cell_list, global_cell_sizes, global_n,
                        global_number_of_colours );
         }
         break;                        // while
      }
      i++;
   }

   int k= 0;

   /* set nauty variables lab and ptn
    */
   set_label(lab, ptn, cell_list, global_n, global_number_of_colours );
}
else if(global_number_of_colours  == 1){
   /* partition with unit partition, i.e. one colour don't need to set lab
    * and ptn
    */
    options.defaultptn = TRUE;
   /* need  to set the variable global_cell_sizes
    */
   global_cell_sizes[0] = global_n;     // all vertices have colour 0
   }
   else{
      printf("Error: the number of cells in initial partition is not set \n");
   }

   /* The following optional call verifies that we are linking
    * to compatible versions of the nauty routines.
    */
   nauty_check(WORDSIZE,m,global_n,NAUTYVERSIONID);

   /* call the NAUTY library
    */
   nauty(global_g,lab,ptn,NULL,orbits,&options,&stats, workspace,100*MAXM,m,
```

```
            global_n,global_canong);

      /* insert canonical form of graph in tree, if canonical form is not in the
       * tree yet; returns 0 if canonical form is already in tree, 1 if it is
       * inserted, -1 otherwise
       * variable global_cell_sizes has been set with set_colour_of_node or
       * manually if there is only one cell.
       */
      exists = insert_canon(global_canong, global_cell_sizes, &store_tree,
                            global_n);
      return exists;
   }
   else{
      printf("ERROR in prob_exists_graph: memory for graph needs be allocated"
             "before existence can be checked! Run function prob_start_graph()"
             "first \n");
   }
   return -1;                              // returns before here
}


/* free all allocated memory and set pointers to NULL
*/
void prob_free_storage(void){

   if(!is_storage_free){

      free_tree(&store_tree);

      free(global_g);
      global_g = NULL;

      free(global_canong);
      global_canong = NULL;

      free(global_cell_sizes);
      global_cell_sizes = NULL;

      free(cell_list);
      cell_list = NULL;

      free(global_is_set);
      global_is_set = NULL;

      is_storage_free = TRUE;

      /* will need to initialise variables again for next problem
       */
      is_init = FALSE;
      is_set_num_colour = FALSE;
      is_start_graph = FALSE;
   }
```

```
   else{
     printf("storage is free, initialize new with prob_init()\n");
   }
}
```

A.2 Interface Header

```
#include<stdlib.h>
#include<limits.h>

#define MAXN 200*WORDSIZE     // Define this before including nauty.h
#define ERROR_VALUE -20
#define TRACE_VERSION FALSE

#include "nauty22/nauty.h"    // which includes <stdio.h> and other system files

struct Node{
unsigned int number_of_nodes;
int tree_node_number;
int *cell_sizes;
unsigned int *label;
struct Node *left;
struct Node *right;
};

/* variables
*/
extern int check_number_of_nodes;
extern int check_number_of_colours;
extern int check_m;

extern int global_n;
extern int global_m;
extern graph *global_g;
extern graph *global_canong;
extern int global_number_of_colours;
extern int *global_cell_sizes;
extern int *global_is_set;
extern struct Node *store_tree;

/* functions called by ProB not added here
*/

/* functions needed to build and work with graphs
*/
int get_number_of_colours(int n_o_p);
graph* start_graph(int number_of_nodes);
void initGraph(graph *g, int n);
int add_edge(int origin, int dest, graph *g, int n);
void set_node(int node, int is_set[]);
```

```
int init_cell(int number_of_nodes, int number_of_colours, int *cell_list);
void init_cell_sizes(int *cell_sizes, int length);
int set_colour_of_node(int number_of_nodes, int colour, int node, int *cell_list,
                       int *cell_sizes);
int has_colour(int node, int *cell_list, int *cell_sizes, int number_of_nodes,
               int number_of_colours);
void set_label(int *lab, int *ptn, int *cell_list, int number_of_nodes,
               int number_of_colours);

/* functions needed to build the storage tree
*/
int compare(graph *canon1, graph *canon2, int n);
int compare_to_internal(unsigned int *canon, int *cell_sizes, int n,
                        struct Node *internalNode);
int insert_canon(unsigned int *canon, int *cell_sizes, struct Node **treeNode,
                 int n);
void copy_label(unsigned int *canon, unsigned int *label, int n);
void copy_cell_sizes();
void free_tree(struct Node **treeNode);
```

A.3 Internal Functions

The file *internal_functions.c* contains the functions that do all the work within the interface.

```
#include "interface.h"                    // includes nauty.h,

int check_number_of_nodes = 0;
int check_number_of_colours = 0;
int check_m = 0;

int *global_is_set;                       // to mark which node has got a colour


/* takes the number of vertices and allocates the memory for a graph with that
 * number of vertices initializes graph with the empty graph
 */
graph* start_graph(int number_of_nodes){

   /* m-number of unsigned ints to hold number_of_nodes bits
    */
   int m = (number_of_nodes + WORDSIZE - 1) / WORDSIZE;

   /* When interface gets a new graph as input, the value of
    * the global variables check_number_of_nodes and check_m need to be changed
    * as the number of vertices, and theref. m, of the new graph may be different
    */
   check_number_of_nodes = number_of_nodes;
   check_m = m;
```

```
   graph *temp = malloc( (sizeof(graph)) * number_of_nodes * m);

   /* return NULL-pointer and error message if no memory is allocated
    */
   if(!temp){
      printf("in start_graph: no memory to allocate");
      return NULL;
   }

   /* initialise graph with empty graph with n vertices
    */
   initGraph(temp, number_of_nodes);

   return temp;                // malloc( (sizeof(graph)) * number_of_nodes * m);
}


/* initialize the adgjacency matrix of the graph with zeros used in start_graph()
 */
void initGraph(graph *g, int n){

   int m,v;
   set *gv;

   m = (n + WORDSIZE - 1) / WORDSIZE;

   for (v = 0; v < n; ++v) {
      gv = GRAPHROW(g,v,m);
      EMPTYSET(gv,m);
   }
}


/* add an edge to the the graph *g
 */
int add_edge(int origin, int dest, graph *g, int n){

   int m = (n + WORDSIZE - 1) / WORDSIZE;

   /* origin and dest are two vertices of the graph, therefore they must be a
    * number between 0 and n-1
    */
   if(origin<0 || dest <0 || origin>=n || dest>=n){
      printf("error: origin or dest are not between 0 and %d \n", n-1);
      return -1;
   }
   else{
      set *gv;
      gv = GRAPHROW(g,origin,m);

      ADDELEMENT(gv, dest%n);
```

```
   }
   return 0;
}


/* compares two canonical forms in packed form, (adjacency matrix stored as an
 * array of integers; see nauty users guide)
 * need n to finish loop; n must be greater than 0;
 * assumes that both graphs have the same number of vertices, i.e. canonical
 * forms have the same length
 * returns 0 if graphs are isomorphic
 * 1 if first canonical label (label of a new graph) is bigger than the second
 * one (stored graph)
 * -1 if first canonical label (label of a new graph) is smaller than stored one
 */
int compare(graph *canon1, graph *canon2, int n){

   /* compare status; 0 for equal, 1 for canon1 > canon2, 2 for canon1 < canon2
    */
   int comp = ERROR_VALUE;

   if(canon1 && canon2){
      int m = (n + WORDSIZE - 1) / WORDSIZE;
      int k = 0;

      /* for every node n g has m sets that need to be considered as one row
       */
      while(k < n*m){

         int l=0;
         for(l=0; l<m; l++){                        // consider one row
            if(canon1[k+l] == canon2[k+l]){         // consider one setword
               comp = 0;
            }
            else{
               if(canon1[k+l] > canon2[k+l]){
                  comp = 1;
               }
               else{
                  comp = -1;
               }
               /* return after first difference in canonical labels is found
                */
               return comp;
            }
         }
         k=k+m;
      }
   }
   else{
      printf("error in compare: one parameter is Nullpointer \n");
      return ERROR_VALUE;                           // -20
```

```
   }
   /* 0 if no error occurred, like n*m <= 0, and canonical forms are equal
    */
   return comp;
}


/* functions to calculate the partition
 */

/* get the number of colours supplied by ProB;
 * returns -1 if parameter <=0 and error message
 */
int get_number_of_colours(int number_of_colours){
   if(number_of_colours <= 0){
   printf("Error in Function get_number_of_colours: "
          "parameter number_of_colours = %d must be > 0 \n", number_of_colours);
   return -1;
   }
   else{
      /* When interface gets a new graph as input, the value of
       * the global variable check_number_of_colours needs to be changed
       * as the number of colours of the new graph may be different
       */
      check_number_of_colours = number_of_colours;
      return number_of_colours;
   }
}


/* initialize the list with the cells with invalid -1-entries to distinguish from
 * valid entries (node numbers) later
 */
int init_cell(int number_of_nodes, int number_of_colours, int *cell_list){

   /* return -1 and error message if number_of_nodes or number_of_colours < 1
    */
   if(number_of_nodes < 1 || number_of_colours < 1){
      printf("Error in init_cell: number of vertices is %d number of colours "
             "is %d, both parameters must be must be > 0 \n",
              number_of_nodes, number_of_colours );
      return -1;
   }

   if(cell_list){
      int j;
      for(j=0; j<number_of_nodes*number_of_colours; j++){
         cell_list[j] = -1;
      }
   }
   else{
      printf("third parameter cell_list is Nullpointer \n");
```

```
      return -1;
   }

   return 0;                                          // function worked correctly
}


/* inserts a node into the cell list according to the number of the colour
 * initial cell_list:  cell_list[nodes_with_colour 0 -1 .. -1|
 * nodes_with_colour 2 -1 ...-1 | ... | nodes_with_colour num_of_col-1, -1 ...-1]
 */
int set_colour_of_node(int number_of_nodes, int colour, int node, int *cell_list,
                       int *cell_sizes){

   /* return -1 and error message if node is not in correct interval
    */
   if(node < 0 || node >= number_of_nodes){
      printf("error in set_colour_of_node: number of node is %d "
             "but must between 0 and %d", node, number_of_nodes-1);
      return -1;
   }
   /* return -1 and error message if colour < 0
    */
   if(colour < 0 ){
      printf("error in set_colour_of_node: colour is %d but must be >= 0 ",
              colour);
      return -1;
   }

   int j=0;
  /* offset = colour*number_of_nodes
   * search for first empty place in cell: offset + already stored vertices of
   * the same colour
   */
   while( cell_list[colour*number_of_nodes + j] >= 0){
      j++;
   }
   cell_list[colour*number_of_nodes + j] = node;  // store node

   /* a node got the colour  int colour, so the number of vertices with that
    * colour is increased by one, therefor also the size of the respective cell
    */
   cell_sizes[colour] = cell_sizes[colour]+1;

   /* mark that this node has been coloured
    */
   set_node(node, global_is_set);

   return 0;                                         // function worked correctly
}
```

```
/* marks a node that has been coloured
 */
void set_node(int node, int is_set[]){

   int array_index = node/WORDSIZE;
   int bit_index = node % WORDSIZE;
   int bit_mask = 1<<bit_index;

   is_set[array_index] = is_set[array_index] | bit_mask ;
}


/* if node has a colour then the number is returned otherwise colour 0 is given
 * to that node
 */
int has_colour(int node, int *cell_list, int *cell_sizes, int number_of_nodes,
               int number_of_colours){

   /* return -1 and error message if node is not in correct interval
    */
   if(node < 0 || node >= number_of_nodes){
      printf("error in has_colour: number of node is %d "
             " but must between 0 and %d \n", node, number_of_nodes-1);
      return -1;
   }

   int i, j, offset;

   /* search in cell_list cell after cell
    */
   for(i=0; i<number_of_colours; i++){
      offset = i*number_of_nodes;         // next cell
      for(j=0; j<number_of_nodes; j++){
         if(cell_list[offset+j] == node){ // node is in cell i, so has colour i
            return i;
         }
      }
   }

   /* node is not in cell list
    */
   set_colour_of_node(number_of_nodes, 0, node, cell_list, cell_sizes);
   return 0;                              // node has now colour 0;
}

/* set *lab and *ptn manually in case options.defaultptn = FALSE
 */
void set_label(int *lab, int *ptn, int *cell_list, int number_of_nodes,
               int number_of_colours){

   int i,j, lab_index, offset;
   lab_index = 0;                         // index to fill *lab
```

```
   for(i=0; i< number_of_nodes-1; i++){   // default partition
      ptn[i] = NAUTY_INFINITY;
   }
   ptn[number_of_nodes-1] = 0;            // last elem. of *lab is always end of
                                          // a cell

   for(i=0; i<number_of_colours; i++){
      offset = i*number_of_nodes;         // next cell
      for(j=0; j<number_of_nodes; j++){
         if(cell_list[offset+j] >= 0){    // node of cell is put into *lab
            lab[lab_index] = cell_list[offset+j];
            lab_index++;
         }
      }
     /* indicates where end of one cell (vertices with same colour) is
      */
      ptn[lab_index-1] = 0;
   }

   /* lab_index == number_of_nodes otherwise not all or to many vertices have
    * been  copied into *lab
    * lab_index starts with 0, but is incremented once more after last node has
    * been inserted
    */
   if(!(lab_index == number_of_nodes)){
      printf("Error: In function set_label: number of vertices in *lab: %d, must"
             "be the same as the total number of vertices: %d \n", lab_index,
              number_of_nodes);
   }
}


/* initialize the array with the cell sizes with 0
 */
void init_cell_sizes(int *cell_sizes, int length){

   int i=0;
   for(i=0; i<length; i++){
      cell_sizes[i] = 0;
   }
}
```

A.4 Storing the Canonical Forms

For each symmetry class of graphs, there is the canonical form of one graph stored in a binary tree, together with the number of vertices and the sizes of each cell of the respective graph. The file *graph_tree.c* contains functions to insert newly encountered graphs in the binary tree.

```
#include "interface.h"
#include<stdio.h>
#include<stdlib.h>

int tree_node_counter;

/* compares new graph (with n vertices) with an already stored graph
 * returns 0 if new graph equal to the stored graph in the tree,
 * 1 if number of vertices, the first unequal cell of initial partition or
 * canonical label of new graph is bigger than the stored one
 * -1 if number of vertices,  the first unequal cell of initial partition or
 * canonical label of new graph is smaller than the stored one
 * ERROR_VALUE if an error occurred
*/
int compare_to_internal(unsigned int *canon, int *cell_sizes, int n,
                        struct Node *internalNode){

   if(internalNode){

      /* compare first the number of vertices of the new graph and stored graph
       * only if they are equal then the canonical forms need to be compared
       */
      if(n > internalNode->number_of_nodes){
         // new value is bigger
         return 1;
      }
      else if(n < internalNode->number_of_nodes){
         // new value is smaller
         return -1;
      }
      else{
         /* number of vertices are equal when program comes here
          * size of cell_sizes is MAXN for static memory allocation and will be
          * the number of colours in the interface version with dynamic memory
          * allocation
          * global_number_of_colours is the maximum number of colours that can
          * occur in any graph, so compare only so many colours
          */
         int i;
         for(i=0;i<global_number_of_colours;i++){

            if(cell_sizes[i] > internalNode->cell_sizes[i]) {
               // new value is bigger
               return 1;
            }
            else if(cell_sizes[i] < internalNode->cell_sizes[i]) {
               // new value is smaller
               return -1;
            }
            // else continue for loop
         }
```

```
        /* both graphs have same number of vertices: n and same cell sizes for
         * each colour; have to compare canonical labels (label is pointer)
         */
         return compare(canon, internalNode->label, n);
      }
   }
   else{
      printf("Error: internalNode is Nullpointer; can't compare");
      return ERROR_VALUE;
   }
   return ERROR_VALUE;                 // should return before program gets here
}


/* insert canonical form of graph in tree, if canonical form  is not in the tree
 * yet returns number of tree node if canonical form is already in tree,
 * negative of number of tree node if it is newly inserted, 0 otherwise
 */
int insert_canon(unsigned int *canon, int *cell_sizes, struct Node **treeNode,
                 int n_local){

   int m_local = (n_local + WORDSIZE - 1) / WORDSIZE;
   int compare_graph = ERROR_VALUE;    // initialized with error value

   /* canonical form yet to be found or inserted in tree
    */
   int insert = 0;

   /* insert new node here (*treeNode os still pointer)
    */
   if(*treeNode == NULL){

      /* {n , &cell_sizes, &label, NULL, NULL};
       */
      *treeNode = (struct Node*) malloc(sizeof(struct Node));

      /* the label is like the graph itself an adjacency matrix stored in a
       * 1-dim array
       */
      unsigned int *label =
                    (unsigned int*) malloc(sizeof(graph) * n_local * m_local);

      /* the local number of cells is maximum n_local, but the number for the
       * colour can be bigger, so use MAXN as for cell sizes
       */
      int *sizes = (int*) malloc( sizeof(int) * MAXN);

      if(*treeNode && label && sizes){ // memory allocated

         /* size of each cell is default 0 -> initialise allocated memory with
          * zeroes
          */
```

```
        init_cell_sizes(sizes, n_local);

        /* initialise label (which is a graph) with empty graph with n_local
         * vertices
         */
        initGraph(label, n_local);

        /* store the canonical form as label
         */
        copy_label(canon, label, n_local);

        /* store number of vertices with same colour for each colour
         */
        copy_cell_sizes(cell_sizes, sizes, global_number_of_colours);

        /* give any node (graph) in state space a number, if there are already
         * more than INT_MAX states (graphs) then vertices get number INT_MAX
         */
        if(tree_node_counter < INT_MAX){
           (*treeNode)->tree_node_number = tree_node_counter;
           tree_node_counter++;
        }
        else{
           printf("Warning: number of stored graphs exceeds INT_MAX; "
                  "all following vertices have number INT_MAX");
           (*treeNode)->tree_node_number = INT_MAX;
        }
        (*treeNode)->number_of_nodes = n_local;
        (*treeNode)->label = label;
        (*treeNode)->cell_sizes = sizes;
        (*treeNode)->left=NULL;
        (*treeNode)->right=NULL;

        /* canonical form is correctly inserted
         * save the number of that node (negative for not existing before)
         */
        insert = (-1) * ((*treeNode)->tree_node_number);
     }
     else{
        printf("in insert_canon: no memory \n");
     }
  }
  else{
     compare_graph = compare_to_internal(canon, cell_sizes, n_local, *treeNode);
     switch(compare_graph){
     case 0: /* canonical form is already in tree (exists = TRUE)
              * return the number of that node (positive)
              */
             insert = (*treeNode)->tree_node_number;
             break;
     case 1: /* go down left branch
              * use pointer because treeNode->left is changed
```

```
                */
               insert = insert_canon(canon, cell_sizes, &((*treeNode)->left),
                                     n_local);
               break;
      case -1: /* go down right branch
                */
               insert = insert_canon(canon, cell_sizes, &((*treeNode)->right),
                                     n_local);
               break;
      default: printf("error during comparison in function insert_canon");
               break;
      }
   }

   /* return tree node number, if canonical form was already in the tree and
    * negative tree node number, if it was inserted;  0 indicates error
    */
   return insert;
}


/* needed in insert_canon to store the cell_sizes of the current graph
 * the parameter n must be the number of vertices of that graph otherwise the
 * cell sizes may not be stored correctly
 */
void copy_cell_sizes(int *cell_sizes, int *sizes, int n){
   int k = 0;

   while(k < n){                        // for max. every node one cell
      sizes[k] = cell_sizes[k];
      k++;
   }
}


/* needed in insert_canon to store the canonical form as label
 */
void copy_label(unsigned int *canon, unsigned int *label, int n){

   int m = (n + WORDSIZE - 1) / WORDSIZE;
   int k = 0;

   /* for every node n g has m sets that need to be considered as one row
    */
   while(k < n*m){
      int l=0;
      for(l=0; l<m; l++){               // consider one row
         label[k+l] = canon[k+l];       // consider one setword
      }
      k=k+m;
   }
}
```

```
/* free memory that is stored in the tree
 */
void free_tree(struct Node **treeNode){

   if(*treeNode != NULL){

      struct Node *pleft;
      struct Node *pright;

      pleft = (*treeNode)->left;      // pointers to subtrees
      pright = (*treeNode)->right;

      free_tree(&pleft);              // call function for subtrees
      free_tree(&pright);

      /* can free headnode when subtrees are freed;
       * (pleft == NULL && pright == NULL)
       * makes sure that subtrees are empty (not strictly necessary)
       */
      if(!pleft && !pright){
          free((*treeNode)->cell_sizes);
          (*treeNode)->cell_sizes = NULL;

          free((*treeNode)->label);
          (*treeNode)->label = NULL;

          free(*treeNode);
          *treeNode = NULL;
          counter++;
      }
   }
}
```

A.5 The *main* Function

The file *testing.c* contains the *main()* function, and a list of simulated function calls. This file is used to test the interface, without having to integrate it into PROB.

```
// includes nauty.h
#include "interface.h"

int main(int argc, char *argv[])
{

/* example run; function calls have been automatically generated during model
 * checking of scheduler0.mch. The interface was compiled with
```

```
 * #define TRACE_VERSION TRUE in the interface header file. The function calls
 * were written into a file called scheduler0_test1.c. The listing for testing
 * here is only a small extract of that file.
 */
int exists = 0;
prob_init();
prob_set_number_of_colours(230);

prob_start_graph(2);
prob_set_colour_of_node(0, 0);
prob_set_colour_of_node(1, 17);
prob_add_edge(1, 0);
prob_exists_graph();

prob_start_graph(40);
prob_set_colour_of_node(0, 5);
prob_add_edge(0, 8);
prob_set_colour_of_node(8, 6);
prob_add_edge(28, 18);
prob_add_edge(28, 38);
prob_add_edge(28, 22);
// ...
prob_exists_graph();

prob_start_graph(50);
prob_set_colour_of_node(0, 0);
prob_add_edge(0, 10);
prob_set_colour_of_node(10, 1);
prob_add_edge(10, 0);
prob_add_edge(10, 20);
prob_set_colour_of_node(20, 2);
prob_add_edge(20, 10);
prob_add_edge(20, 30);
prob_set_colour_of_node(30, 3);
// ...
prob_exists_graph();

// ...

prob_free_storage();

return 0;
}
```

Appendix B

Machines used for Empirical Results

B.1 Scheduler0

MACHINE $scheduler0$

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

VARIABLES $proc,\ pst$

DEFINITIONS
$scope\_PROC == 1..5;$
$/*\ scope\_PROC2 == p1, p2, p3;\ \ */$
$ASSERT\_LTL0 == \text{“}G([new] \Rightarrow X\ \ e(del))\text{“};$
$ASSERT\_LTL1 == \text{“}G([del] \Rightarrow X\ \ e(new))\text{“};$
$ASSERT\_LTL2 == \text{“}G([enter] \Rightarrow X\ \ e(leave))\text{“};$
$ASSERT\_LTL3 == \text{“}G([leave] \Rightarrow X\ \ e(enter))\text{“}$

INVARIANT
$proc : I\!P(PROC) \wedge$
$pst : proc \rightarrow STATE \wedge$
$/*\ card(q \mid q \in proc \wedge pst(q) = active) \leq 1\ */$
$card(pst^{-1}[\{active\}]) \leq 1$

INITIALISATION
$proc := \emptyset\ ||$
$pst := \emptyset$

OPERATIONS
$new(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC - proc$
THEN

$pst(p) := idle \;||$
$proc := proc \cup p$
END;

$del(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \wedge$ /* *should really be* $p \in proc$; *to avoid undefined* $pst(p)$ */
$pst(p) = idle$
THEN
$proc := proc - \{p\} \;\;||$
$pst := \{p\} \lhd pst$
END;

$ready(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \;\wedge$ /*should really be $p \in proc$; to avoid undefined $pst(p)$ */
$pst(p) = idle$
THEN
$pst(p) := ready$
END;

$enter(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \wedge$
$pst(p) = ready \wedge$
$/^* q \mid q \in proc \wedge pst(q) = active = \emptyset$ */
$pst^{-1}[\{active\}] = \emptyset$
THEN
$pst(p) := active$
END;

$leave(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \wedge$
$pst(p) = active$
THEN
$pst(p) := idle$
END
END

B.2 Scheduler1

REFINEMENT *scheduler*1

REFINES *scheduler*0

DEFINITIONS

$scope\_PROC == 1..5;$
$ASSERT\_LTL ==$ "$G([new] \rightarrow X\ e(del))$"

VARIABLES $proc, readyq, activep, activef, idleset$

INVARIANT
$proc \in I\!P(PROC) \wedge$
$readyq \in seq(PROC) \wedge$
$activep \in PROC \wedge$
$activef \in BOOL \wedge$
$idleset \in I\!P(PROC)$

INITIALISATION
$proc := \emptyset \ ||$
$readyq := \emptyset \ ||$
$activep :\in PROC \ ||$
$activef := FALSE \ ||$
$idleset := \emptyset$

OPERATIONS
$new(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC - proc$
THEN
$idleset := idleset \cup \{p\} \ ||$
$proc := proc \cup \{p\}$
END;

$del(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \wedge$
$p \in idleset$
THEN
$proc := proc - \{p\} \ ||$
$idleset := idleset - \{p\}$
END;

$ready(p) \mathrel{\widehat{=}}$
PRE
$p \in idleset$
THEN
$readyq := readyq \leftarrow p \ ||$
$idleset := idleset - \{p\}$
END;

$enter(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \wedge$
$readyq \neq [] \wedge$

$p = first(readyq) \wedge$
$activef = FALSE$
THEN
$activep := p \ ||$
$readyq := tail(readyq) \ ||$
$activef := TRUE$
END;

$leave(p) \mathrel{\widehat{=}}$
PRE
$p \in PROC \wedge$
$activef = TRUE \wedge$
$p = activep$
THEN
$idleset := idleset \cup \{p\} \ ||$
$activef := FALSE$
END
END

B.3 Russian_Postal_Puzzle

MACHINE *RussianPostalPuzzle*

SETS
$KeyIDs$;
$PERSONS = \{natasha, boris\}$

DEFINITIONS
$scope\_KeyIDs == 1..3$;
$GOAL == (padlocks = \emptyset \wedge box\_contains\_gem = TRUE \wedge hasbox = natasha)$

VARIABLES
$keysforsale, \ hasbox, \ padlocks, \ has\_keys, \ box\_contains\_gem$

INVARIANT
$keysforsale \in \mathbb{P}(KeyIDs) \wedge$
$hasbox \in PERSONS \wedge$
$padlocks \subseteq KeyIDs \wedge$
$has\_keys \in PERSONS \rightarrow POW(KeyIDs) \wedge$
$box\_contains\_gem \in BOOL$

INITIALISATION
$keysforsale := KeyIDs \ ||$
$hasbox := boris \ ||$
$padlocks := \emptyset \ ||$
$has\_keys := \{natasha \mapsto \emptyset, \ boris \mapsto \emptyset\} \ ||$
$box\_contains\_gem := TRUE$

OPERATIONS

buy_padlock_and_key(*keyid*, *person*) $\widehat{=}$
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*) $\widehat{=}$
PRE
$keyid \in KeyIDs \wedge$
$person \in PERSONS \wedge$
$person = hasbox \wedge$
$keyid \in has\_keys(person) \wedge$
$keyid \neq padlocks$
THEN
$padlocks := padlocks \cup \{keyid\}$
END;

remove_padlock(*keyid*, *person*) $\widehat{=}$
PRE
$keyid \in KeyIDs \wedge$
$person \in PERSONS \wedge$
$person = hasbox \wedge$
$keyid \in padlocks \wedge$
$keyid \in has\_keys(person)$
THEN
$padlocks := padlocks - keyid$
END;

send_box(*from*, *to*) $\widehat{=}$
PRE
$from \in PERSONS \wedge$
$from = hasbox \wedge$
$to \in PERSONS \wedge$
$to \neq hasbox$
THEN
IF $padlocks = \emptyset$ **THEN**
$box\_contains\_gem := FALSE$
END ||
$hasbox := to$
END
END

B.4 USB_4 Endpoints

/ In this model, we describe USB transfers in an abstract way:*

USB protocol is described in term of transfers as a n-tuple: (host, device endpoint, transfer type). Transfers begin and terminate. Contraints are:
- control transfers only occur on endpoint #0
- IBI transfers occur on endpoinr #1 to #15
- only one transfer at a time for any endpoint
**/*

MACHINE USB

SETS
$USB\_TRANSFER\_TYPE = \{$
$BULK\_TRANSFER,$
$CONTROL\_TRANSFER,$
$INTERRUPT\_TRANSFER,$
$ISOCHRONOUS\_TRANSFER$
$\};$
$TRANSFERS$ */* Set of all possible transfers **/*

CONSTANTS
$ENDPOINTS$

PROPERTIES
$ENDPOINTS = 0..4$ */* $\wedge\ ENDPOINTS : \mathbb{P}(NAT)$ **/*

VARIABLES
$transfers,$ */* list of all transfers initiated since last power on reset **/*
$transfer\_type,$ */* type of transfer between host and device **/*
$transfer\_endpoint,$ */* device endpoint used for the transfer between host and device **/*
$transfer\_completed$ */* indicates if a transfer has terminated or is pending **/*

DEFINITIONS
$scope\_TRANSFERS == 1..2$

INVARIANT
$transfers \subseteq TRANSFERS \wedge$
$transfer\_type \in transfers \nrightarrow USB\_TRANSFER\_TYPE \wedge$
$transfer\_endpoint \in transfers \nrightarrow ENDPOINTS \wedge$
$transfer\_completed \in transfers \nrightarrow BOOL \wedge$

/ Structural: a transfer has a type, an end point associated and a completion status **/*
$\forall tr.(tr \in transfers \Rightarrow$
$tr \in dom(transfer\_type) \wedge$
$tr \in dom(transfer\_endpoint) \wedge$
$tr \in dom(transfer\_completed)) \wedge$

/ Endpoint 0 is only used for control transfers **/*
$transfer\_type^{-1}[\{CONTROL\_TRANSFER\}] = transfer\_endpoint^{-1}[\{0\}]$

INITIALISATION
transfers := ∅ ||
transfer_type := ∅ ||
transfer_endpoint := ∅ ||
transfer_completed := ∅

OPERATIONS
initiate_control_transfer ≙
ANY *tr* **WHERE**
tr ∈ *TRANSFERS* − *transfers* ∧
/* *Can initiate a control transfer only if there is no pending control transfer* */
$transfer\_type^{-1}[\{CONTROL\_TRANSFER\}] \cap$
$transfer\_completed^{-1}[\{FALSE\}] = \emptyset$
THEN
transfers := *transfers* ∪ {*tr*} ||
transfer_type := *transfer_type* ⩤ {*tr* ↦ *CONTROL_TRANSFER*} ||
transfer_endpoint := *transfer_endpoint* ⩤ {*tr* ↦ 0} ||
transfer_completed := *transfer_completed* ⩤ {*tr* ↦ *FALSE*}
END;

initiate_bulk_transfer ≙
ANY*tr*, *ep***WHERE**
tr ∈ *TRANSFERS* − *transfers* ∧
ep ∈ *ENDPOINTS* − {0} ∧
/* *Can initiate a bulk transfer for an endpoint of a device only if there is no pending transfer for this endpoint* */
$transfer\_endpoint^{-1}[\{ep\}] \cap transfer\_completed^{-1}[\{FALSE\}] = \emptyset$
THEN
transfers := *transfers* ∪ {*tr*} ||
transfer_type := *transfer_type* ⩤ {*tr* ↦ *BULK_TRANSFER*} ||
transfer_endpoint := *transfer_endpoint* ⩤ {*tr* ↦ *ep*} ||
transfer_completed := *transfer_completed* ⩤ {*tr* ↦ *FALSE*}
END;

initiate_interrupt_transfer ≙
ANY *tr*, *ep* **WHERE**
tr ∈ *TRANSFERS* − *transfers* ∧
ep ∈ *ENDPOINTS* − {0} ∧
/* *Can initiate an interrupt transfer for an endpoint of a device only if there is no pending transfer for this endpoint* */
$transfer\_endpoint^{-1}[\{ep\}] \cap transfer\_completed^{-1}[\{FALSE\}] = \emptyset$
THEN
transfers := *transfers* ∪ {*tr*} ||
transfer_type := *transfer_type* ⩤ {*tr* ↦ *INTERRUPT_TRANSFER*} ||
transfer_endpoint := *transfer_endpoint* ⩤ {*tr* ↦ *ep*} ||

$transfer\_completed := transfer\_completed \lhd\!\!+ \{tr \mapsto FALSE\}$
END;

initiate_isochronous_transfer $\widehat{=}$
ANY tr, ep **WHERE**
$tr \in TRANSFERS - transfers \wedge$
$ep \in ENDPOINTS - \{0\} \wedge$
/* Can initiate a isochronous transfer for an endpoint of a device only if there is no pending transfer for this endpoint */
$transfer\_endpoint^{-1}[\{ep\}] \cap transfer\_completed^{-1}[\{FALSE\}] = \emptyset$
THEN
$transfers := transfers \cup \{tr\}$ ||
$transfer\_type := transfer\_type \lhd\!\!+ \{tr \mapsto ISOCHRONOUS\_TRANSFER\}$ ||
$transfer\_endpoint := transfer\_endpoint \lhd\!\!+ \{tr \mapsto ep\}$ ||
$transfer\_completed := transfer\_completed \lhd\!\!+ \{tr \mapsto FALSE\}$
END;

terminate_control_transfer $\widehat{=}$
ANY tr **WHERE**
$tr \in transfers \wedge$
$transfer\_type(tr) = CONTROL\_TRANSFER \wedge$
$transfer\_endpoint(tr) = 0 \wedge$
$transfer\_completed(tr) = FALSE$
THEN
$transfer\_completed(tr) := TRUE$
END;

terminate_bulk_transfer $\widehat{=}$
ANY tr **WHERE**
$tr \in transfers \wedge$
$transfer\_type(tr) = BULK\_TRANSFER \wedge$
$transfer\_endpoint(tr) \in 1..15 \wedge$
$transfer\_completed(tr) = FALSE$
THEN
$transfer\_completed(tr) := TRUE$
END;

terminate_interrupt_transfer $\widehat{=}$
ANY tr **WHERE**
$tr \in transfers \wedge$
$transfer\_type(tr) = INTERRUPT\_TRANSFER \wedge$
$transfer\_endpoint(tr) \in 1..15 \wedge$
$transfer\_completed(tr) = FALSE$
THEN
$transfer\_completed(tr) := TRUE$
END;

terminate_isochronous_transfer $\widehat{=}$
  ANY tr **WHERE**
    $tr \in transfers \wedge$
    $transfer\_type(tr) = ISOCHRONOUS\_TRANSFER \wedge$
    $transfer\_endpoint(tr) \in 1..15 \wedge$
    $transfer\_completed(tr) = FALSE$
  THEN
    $transfer\_completed(tr) := TRUE$
  END;

*configure_endpoint*0 $\widehat{=}$ *skip*; /**introduced in USB_1.ref* */
configure_IBI_endpoint $\widehat{=}$ *skip*; /**introduced in USB_1.ref* */
deconfigure_IBI_endpoint $\widehat{=}$ *skip*; /**introduced in USB_1.ref* */

initiate_setup_transaction $\widehat{=}$ *skip*; /**eventintroduced in USB_2* */
end_setup_transaction $\widehat{=}$ *skip*; /**eventintroduced in USB_2* */
initiate_data_transaction_control_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */
end_data_transaction_control_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */
initiate_status_transaction $\widehat{=}$ *skip*; /**event introduced in USB_2* */
end_status_transaction $\widehat{=}$ *skip*; /**event introduced in USB_2* */

initiate_data_transaction_interrupt_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */
end_data_transaction_interrupt_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */

initiate_data_transaction_bulk_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */
end_data_transaction_bulk_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */

initiate_data_transaction_isochronous_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */
end_data_transaction_isochronous_transfer $\widehat{=}$ *skip*; /**event introduced in USB_2* */

issue_packet_in_interrupt_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_data_interrupt_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_ack_interrupt_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_nack_interrupt_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_stall_interrupt_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */

issue_packet_in_isochronous_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_data_in_isochronous_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_out_isochronous_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_data_out_isochronous_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */

issue_packet_in_bulk_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_out_bulk_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
issue_packet_data_bulk_transaction $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */

issue_packet_sof $\widehat{=}$ *skip*; /**eventintroducedinUSB_3* */
receive_transaction_packet $\widehat{=}$ *skip*; /**introducedinUSB_4* */
receive_sof_packet $\widehat{=}$ *skip* /**introducedinUSB_4* */

END

B.5 Token Ring

MACHINE *TokenRing*

SETS
Servers

DEFINITIONS
$scope\_Servers == 1..5;$
$ASSERT\_LTL ==" G([Release] \Rightarrow X\{in\_critical = \emptyset\})"$

CONSTANTS
next

PROPERTIES
$next : Servers \rightarrowtail Servers$

VARIABLES
token,
requests,
in_critical

INVARIANT
$token \in Servers \wedge$
$requests \subseteq Servers \wedge$
$in\_critical \subseteq Servers$

INITIALISATION
$token :\in Servers \ ||$
$requests := \emptyset \ ||$
$in\_critical := \emptyset$

OPERATIONS
$MoveToken \mathrel{\widehat=}$
PRE $in\_critical = \emptyset$
THEN $token := next(token)$
END;

$ClientRequest(s) \mathrel{\widehat=}$
PRE $s \in Servers \wedge s \notin requests$
THEN $requests := requests \cup \{s\}$
END;

$GrantRequest(s) \mathrel{\widehat=}$
PRE $s = token \wedge in\_critical = \emptyset$
THEN $in\_critical := \{s\}$
END;

$Release(s) \mathrel{\widehat{=}}$
PRE $s \in in\_critical$
THEN $in\_critical := in\_critical - \{s\}$
END

END

B.6 Dining

MACHINE *Dining*

SETS
Phil;
Forks

DEFINITIONS
$scope\_Phil == 1..5$;
$scope\_Forks == 1..5$

CONSTANTS
lFork,
rFork

PROPERTIES
$lFork : Phil \rightarrowtail\!\!\!\!\!\!\rightarrow Forks \wedge$
$rFork : Phil \rightarrowtail\!\!\!\!\!\!\rightarrow Forks \wedge$
$\forall pp.(pp \in Phil \Rightarrow lFork(pp) \neq rFork(pp))$

VARIABLES
taken

INVARIANT
$taken \in Forks \nrightarrow Phil \wedge$
$\forall xx.(xx \in dom(taken) \Rightarrow (lFork(taken(xx)) = xx \vee rFork(taken(xx)) = xx))$

INITIALISATION
$taken := \emptyset$

OPERATIONS
$TakeLeftFork(p, f) \mathrel{\widehat{=}}$
PRE $p \in Phil \wedge f \in Forks \wedge f \notin dom(taken) \wedge lFork(p) = f$
THEN $taken(f) := p$
END;

$TakeRightFork(p, f) \mathrel{\widehat{=}}$
PRE $p \in Phil \wedge f \in Forks \wedge f \notin dom(taken) \wedge rFork(p) = f$
THEN $taken(f) := p$
END;

$DropFork(p, f) \mathrel{\widehat{=}}$
PRE $p \in Phil \wedge f \in Forks \wedge f \in dom(taken) \wedge taken(f) = p$

THEN $taken := \{f\} \lhd taken$
END

END

B.7 Towns

MACHINE $Towns(TOWN)$

SETS
$ANSWER = \{connected, notconnected\}$

VARIABLES
$roads$

INVARIANT
$roads \in TOWN \leftrightarrow TOWN$

INITIALISATION
$roads := \emptyset$

OPERATIONS
$link(tt1, tt2) \mathrel{\widehat{=}}$
PRE $tt1 \in TOWN \wedge tt2 \in TOWN$
THEN $roads := roads \cup \{tt1 \mapsto tt2\}$
END;

$ans \longleftarrow connectedquery(tt1, tt2) \mathrel{\widehat{=}}$
PRE $ut1 \in TOWN \wedge ut2 \in TOWN$
THEN
IF $tt1 \mapsto tt2 \in closure1(roads \cup roads^{-1}) \vee (tt1 = tt2)$
THEN $ans := connected$
ELSE $ans := notconnected$
END
END

END

B.8 TicTacToe_Sym

MACHINE $TicTacToe\_Sym$

SETS $NC = \{O, X\}$;
$Rows$; /* $top, middle, bottom$ */
$Cols$ /* $left, middle, right$ */

DEFINITIONS
$win\_vert\_horiz(ox) == \exists(rc, r).(rc \in RowCol \wedge r \in Pos \wedge$
$\forall c.(c \in Pos \Rightarrow (rc, r, c) \in dom(board) \wedge board(rc, r, c) = ox))$;
$win\_diag1(ox) == \exists(rc).(rc \in RowCol \wedge$

$\forall\, c.(c \in Pos \Rightarrow (rc, c, c) \in dom(board) \ \wedge \ board(rc, c, c) = ox));$
$win\_diag2(ox) == \exists (rc, p1, p2).(rc \in RowCol \wedge \ p1 \in Pos \wedge \ p2 \in Pos \wedge$
$\{p1, p2\} = Pos - middle \wedge$
$(rc, p1, p2) \in dom(board) \wedge \ board(rc, p1, p2) = ox \wedge$
$(rc, p2, p1) \in dom(board) \wedge \ board(rc, p2, p1) = ox \wedge$
$(rc, middle, middle) \in dom(board) \wedge \ board(rc, middle, middle) = ox);$
$win(ox) == (win\_vert\_horiz(ox) or win\_diag1(ox) or win\_diag2(ox));$
$GOAL == win\_vert\_horiz(X)$

CONSTANTS

$middle,$
$other$

PROPERTIES

$middle \in Pos \ \wedge$
$card(Pos) = 3 \ \wedge$
$card(RowCol) = 2 \ \wedge$
$other = \{O \mapsto X, X \mapsto O\}$

VARIABLES

$turn,$
$board$

INVARIANT

$turn \in \{O, X\} \ \wedge$
$board \in RowCol * Pos * Pos \nrightarrow NC$

INITIALISATION

$board := \emptyset \ \ ||$
$turn := X$

OPERATIONS

$Win(ox) \ \widehat{=}$
PRE
$turn = ox \ \wedge \ win(ox)$
THEN
skip
END;

$Put(nc, rc, rc2, r, c) \ \widehat{=}$
PRE
$turn = nc \ \wedge \neg(win(other(nc))) \ \wedge (rc, r, c) \notin dom(board) \ \wedge \ rc2 \neq rc$
THEN
$board := board \Leftarrow \{(rc, r, c) \mapsto nc, (rc2, c, r) \mapsto nc\} \ \ ||$
$turn := other(turn)$
END
END

B.9 TicTacToe_SimplerSym

MACHINE *TicTacToe_SimplerSym*

/* *Rows can be swapped by symmetry in this model; similarly Columns; but not rows against columns* */

SETS

$NC = \{O, X\}$;
Rows; /* *top*, *middle*, *bottom* */
Cols /* *left*, *middle*, *right* */

DEFINITIONS

$win\_vert(ox) == \exists(r).(r \in Rows \wedge$
$\forall c.(c \in Cols \Rightarrow (r, c) \in dom(board) \wedge board(r, c) = ox))$;
$win\_horiz(ox) == \exists(c).(c \in Cols \wedge$
$\forall r.(r \in Rows \Rightarrow (r, c) \in dom(board) \wedge board(r, c) = ox))$;
$win\_diag(ox) == \exists(r1, r2, c1, c2).(\{r1, r2\} = Rows - \{middleR\} \wedge$
$c1, c2 = Cols - \{middleC\} \wedge$
$(r1, c1) \in dom(board) \wedge board(r1, c1) = ox \wedge$
$(r2, c2) \in dom(board) \wedge board(r2, c2) = ox \wedge$
$(middleR, middleC) \in dom(board) \wedge board(middleR, middleC) = ox)$;
$win(ox) == (win\_vert(ox) or win\_horiz(ox) or win\_diag(ox))$;
GOAL $== win\_vert\_horiz(X)$;

CONSTANTS

middleR,
middleC,
other

PROPERTIES

$middleR \in Rows \wedge$
$card(Rows) = 3 \wedge$
$middleC \in Cols \wedge \quad card(Cols) = 3 \wedge$
$other = \{O \mapsto X, X \mapsto O\}$

VARIABLES

turn,
board

INVARIANT

$turn \in \{O, X\} \wedge$
$board \in Rows * Cols \nrightarrow NC$

INITIALISATION

$board := \emptyset \ ||$
$turn := X$

OPERATIONS

$Win\_Vert(ox) \mathrel{\widehat{=}}$
PRE

$turn = other(ox) \ \wedge win\_vert(ox)$
THEN
skip
END;

$Win\_Horiz(ox) \mathrel{\widehat{=}}$
PRE
$turn = other(ox) \ \wedge win\_horiz(ox)$
THEN
skip
END;

$Win\_Diag(ox) \mathrel{\widehat{=}}$
PRE
$turn = other(ox) \wedge \ win\_diag(ox)$
THEN
skip
END;

$Put(nc, r, c) \mathrel{\widehat{=}}$
PRE
$turn = nc \wedge \neg(win(other(nc))) \wedge (r, c) \notin dom(board)$
THEN
$board(r, c) := turn \ ||$
$turn := other(turn)$
END
END

Appendix C

Additional Empirical Results

Table C.1 shows for various machines, the number of graphs encountered for the given cardinality of the deferred set. Then, for each machine, the number of graphs in the rightmost column is broken down, according to the number of vertices. The average number of edges is listed in the second rightmost column. For example, we had 94 graphs calculated for the *scheduler1*, with cardinality three of the deferred set in the machine. Of those graphs, three graphs have 45 vertices - and, on average, 81 edges.

Table C.1: Empirical Results III

Card.	Number of Vertices	Number of Edges (average)	Number of Graphs
3	scheduler 0		59
	4	6,0	2
	6	8,0	10
	8	12,0	8
	10	14,3	19
	12	18,0	15
	14	20,0	5
3	scheduler1		94
	20	37,0	11
	25	46,0	29
	30	55,0	33
	35	63,7	12
	40	71,7	6
	45	81,0	3
2	RussianPostalPuzzle		105
	40	72,0	4
	45	80,6	101
1	USB_4Endpoints		838
	6	5,0	1
	35	65,0	31
	45	81,0	806
3	Token Ring		183
	3	3,0	6
	16	30,9	66
	20	36,7	111

Table C.2 shows for the machine *scheduler*0, the number of graphs encountered for the given cardinality of the deferred set. The number of graphs is broken down as in Table C.1.

Table C.2: Empirical Results IV

Card.	Number of Vertices	Number of Edges (average)	Number of Graphs
2	scheduler0		23
	4	6,0	2
	6	8,0	9
	8	12,0	3
	10	14,0	9
3			59
	4	6,0	2
	6	8,0	10
	8	12,0	8
	10	14,3	19
	12	18,0	15
	14	20,0	5
4			121
	4	6,0	2
	6	8,0	11
	8	12,0	9
	10	14,7	27
	12	18,2	30
	14	21,5	30
	16	24,0	12
5			216
	4	6,0	2
	6	8,0	12
	8	12,0	10
	10	14,7	30
	12	18,5	40
	14	21,8	52
	16	25,2	49
	18	28,0	21
6			351
	4	6,0	2
	6	8,0	13
	8	12,0	11
	10	14,7	33
	12	18,5	44
	14	22,0	65
	16	25,5	79
	18	29,1	72
	20	32,0	32
7			533
	4	6,0	2
	6	8,0	14
	8	12,0	12
	10	14,7	36
	12	18,5	48
	14	22,0	71
	16	25,8	95
	18	29,4	111
	20	33,0	99
	22	36,0	45

Bibliography

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 186–201, New York, NY, USA, 1999. ACM.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975. Second printing, Addison-Wesley Series in Computer Science and Information Processing.

[4] M. A. Armstrong. *Groups and Symmetry*. Springer-Verlag, 1988.

[5] AT&T Labs-Research. Graphviz - open source graph drawing software. Obtainable at `http://www.research.att.com/sw/tools/graphviz/`.

[6] U. B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.

[7] S. Barner and O. Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. *Form. Methods Syst. Des.*, 27(1-2):29–66, 2005.

[8] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: an industry-oriented formal verification tool. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 655–660, New York, NY, USA, 1996. ACM.

[9] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 1–19, London, UK, 2000. Springer-Verlag.

[10] D. Bosnacki, D. Dams, and L. Holenderski. A heuristic for symmetry reductions with scalarsets. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 518–533, London, UK, 2001. Springer-Verlag.

[11] M. J. Butler. *A CSP Approach To Action Systems.* PhD thesis, University of Oxford, 1992.

[12] K. M. Chandy and J. Misra. *Parallel Program Design – a foundation.* Addisson-Wesley, 1988.

[13] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[14] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[15] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR 1996*, pages 148–159, 1996.

[16] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 530–534, New York, NY, USA, 2004. ACM.

[17] G. Dennis. Tsafe: Building a trusted computing base for air traffic control. Master's thesis, Department of Electrical Engineering and Computer Science, 32 Vassar Street, 32-G706, Cambridge, MA 02139, 2003.

[18] R. Diestel. *Graphentheorie.* Springer-Verlag, 2000.

[19] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[20] S. Flannery and D. Flannery. *In Code: A Mathematical Journey.* Algonquin Books of Chapel Hill, 2002.

[21] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* PhD thesis, University of Liege, 1994.

[22] G. J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[23] G. J. Holzmann. An analysis of bitstate hashing. In *Formal Methods in System Design*, pages 301–314. Chapman & Hall, 1995.

[24] G. J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[25] G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *In Proceedings of Third International SPIN Workshop*, 1997.

[26] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *In Proceedings of Second International SPIN Workshop*, 1996.

[27] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 172–184, New York, NY, USA, 1974. ACM.

[28] C. N. Ip and D. L. Dill. Better verification through symmetry. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 97–111, Amsterdam, The Netherlands, 1993. North-Holland Publishing Co.

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

[30] D. Jackson. The Alloy Analyser. Available at `http://alloy.mit.edu/alloy4/`, 2006.

[31] D. S. Johnson. The np-completeness column. *ACM Trans. Algorithms*, 1(1):160–176, 2005.

[32] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Workshop on Algorithm Engineering and Experiments (ALENEX07)*. Society for Industrial and Applied Mathematics, 2007.

[33] P. J. Kelly. A congruence theorem for trees. *Pacific J. Math.*, 7:961–968, 1957.

[34] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, Search*. CRC Press, 1999.

[35] L. Laboratory. *B Site page*. `http://www-lsr.imag.fr/B/Bsite-pages.html`.

[36] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[37] M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.

[38] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B-method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[39] M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.

[40] M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. *Proceedings International Symmetry Conference*, pages 71–85, Januar 2007.

[41] M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In *Proceedings Isola 2007*, Revue des Nouvelles Technologies de l'Information RNTI-SM-1, pages 73–84, December 2007.

[42] B. D. McKay. Nauty users guide. Available at `http://cs.anu.edu.au/people/bdm/nauty/`.

[43] B. D. McKay. Practical graph isomorphism. *Congress Numerantium*, pages 45–87, 1981. Available at `http://cs.anu.edu.au/ bdm/nauty/PGI/`.

[44] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.

[45] A. Miller, A. F. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006.

[46] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[47] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.

[48] A. Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[49] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.

[50] SICStus. *SICStus Prolog User's Manual*, 2005. Available at `http://www.sics.se/isl/sicstus/docs/3.12.3/pdf/sicstus.pdf`.

[51] C. Spermann and M. Leuschel. Prob gets Nauty: Effective symmetry reduction for B and Z models. In *Proceedings Symposium TASE 2008*, pages 15–22, Nanjing, China, June 2008. IEEE.

[52] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1992.

[53] F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.

[54] Tcl/Tk. Tcl/Tk Developer Exchange Website, 2009. http://www.tcl.tk/doc/.

[55] E. Turner. *Improving the Process of Model Checking through State Space Reductions*. PhD thesis, University of Southampton, 2007.

[56] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.