Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the
10th International Workshop on
Automated Verification of Critical Systems
(AVoCS 2010)

Verification of Symmetry Detection using PVS

Shamim Ripon and Alice Miller

16 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verification of Symmetry Detection using PVS

Shamim Ripon[1] **and Alice Miller**[2*]

[1]shamim@cs.york.ac.uk
Department of Computer Science, University of York, UK

[2] alice@dcs.gla.ac.uk
Department of Computing Science, University of Glasgow, UK

Abstract: One of the major limitations of model checking is that of state-space explosion. Symmetry reduction is a method that has been successfully used to alleviate this problem for models of systems that consist of sets of identical components. In earlier work, we have introduced a specification language, Promela-Lite, which captures the essential features of Promela but has a fully defined semantics. We used hand proofs to show that a static symmetry detection technique developed for this language is sound, and suitable to be used in a symmetry reduction tool for SPIN. One of the criticisms often levelled at verification implementations, is that they have not been proved mechanically to be correct, i.e., no mechanical formal verification technique has been used to check the soundness of the approach. In this paper, we address this issue by mechanically verifying the correctness of the symmetry detection technique. We do this by embedding the syntax and semantics of Promela-Lite into the theorem prover PVS and using these embeddings to both check the consistency of syntax/semantics definitions, and interactively prove relevant theoretical properties.

Keywords: Symmetry, Semantics, Theorem Proving, PVS

1 Introduction

Promela-Lite [DM08] is a specification language that captures the core features of Promela - the input language for the SPIN model checker [Hol03]. Unlike Promela, Promela-Lite has a rigorously defined semantics, making it a suitable vehicle for proving correctness of verification and state-space reduction techniques for Promela. The language was designed for proving correct an automatic symmetry detection technique for Promela [DM08]. The technique involves the derivation of state-space preserving automorphisms from the text of a Promela specification, to be exploited during search to reduce the space and time requirements of model checking via *symmetry reduction* [CEF$^+$96, ES96, ID96]. This symmetry detection technique, based on *static channel diagram analysis*, has been implemented as part of TopSPIN, a symmetry reduction package for the SPIN model checker [DM06].

Model checking of a specification described in Promela involves the construction of an associated Kripke structure [CGP99], the size of which grows exponentially as the number of

* This work is funded by EPSRC grant EP/E032354

components in the system increases. This phenomenon is known as *state space explosion*. In symmetry reduced model checking [CEF$^+$96, ES96, ID96], a *quotient* structure is checked instead of the entire Kripke structure. The quotient structure, which is generally smaller than the original Kripke structure, is constructed using the symmetry in the underlying model. An automatic symmetry detection technique for Promela is presented in [DM08]. Promela-Lite is defined to allow us to provide a formal proof of the techniques used in this automatic symmetry detection technique. A full grammar, type system, and a Kripke structure semantics of Promela-Lite have been defined in [DM08] to support the proofs of symmetry detection techniques. These proofs have been carried out by hand.

The soundness of model checking depends critically on the correctness of underlying algorithms and reduction techniques. For example, an erroneous symmetry detection method may compute state-space permutations which are not structure-preserving, potentially resulting in incorrect verification results. For this reason, it is highly desirable that correctness proofs for model checking techniques, such as the proof by hand presented in [DM08], are mechanically verified. Mechanical verification is widely used as a tool to verify the syntax and the semantic models of a language. Verification of language properties may identify flaws in the language, which can be remedied to give increased confidence in the language definition. Theorem provers are heavily used as a tool to mechanically verify language properties. To do this, the language and its semantic model must be embedded into the theorem prover.

Several theorem provers, such as PVS [ORS92], HOL [GM93], Coq [Be97] and Isabelle [Pau94], have rich specification languages, automated support for decision procedures, and proof strategies tailored to their logics. PVS (Prototype Verification System) is an automated framework for specification and verification. PVS supports higher order logic, allows abstract datatypes to model process terms, and has strong support for induction mechanisms.

In this paper, inspired by other successful attempts to embed specification languages into PVS [Hel06, POS04, RB09], we show how the Promela-Lite syntax, type system and semantics can be embedded into PVS, and use this embedding to interactively prove both consistency of the syntax/semantics definitions, and language properties. In particular, we concentrate on proving theorems related to automatic symmetry detection which have previously been proved only by hand. By demonstrating how a particular formal technique can be mechanically verified, we lead the way for mechanical proof to become standard in the development of such techniques.

The rest of the paper is organised as follows. Section 2 gives a brief overview of the Promela-Lite language and illustrates the language features using an example specification of a resource allocation system. Section 3 shows the embeddings of the Promela-Lite syntax, types, and the type system into PVS. In section 4 we introduce the static channel diagram (*SCD*) associated with a specification, and define the automorphism group of a Kripke structure and of an *SCD*. We state the main theorem to be mechanically proved in this paper, namely the Correspondence Theorem. The proof of this theorem is supported by a series of lemmas. We show how each of these lemmas are proved in PVS and how they are used in the proof of the correspondence theorem in Section 5. After discussing related work in Section 6, we give our conclusions in Section 7.

2 Promela-Lite

A specification in Promela usually consists of a series of global variables and channel declarations, and process type declarations, along with an initialisation process, `init`. Properties to be verified are specified either by using `assert` statements within specifications, or via LTL properties. Promela-Lite [DM08] includes some core features of Promela such as parameterized processes, channels (first class) and global variables, but omits some types such as enumerated types, records, arrays, and rendezvous channels. Unlike Promela, a full grammar and type system along with a Kripke structure semantics of Promela-Lite have been defined.

The syntax of Promela-Lite (see Fig. 1) is defined in [DM08] using the standard BNF form (e.g. [ASU86]). A channel declaration `chan` c `= [`a`] of` $\{\overline{T}\}$ defines a buffered channel with type $chan\{\overline{T}\}$ (where $\overline{T}$ is a comma separated list of types). Note that all channels are *static* – their names cannot be reassigned. A Promela-Lite proctype is a parameterized process definition. A statement has the form: `atomic` $\{\langle guard\rangle$ `->`$\langle update\text{-}list\rangle$‘`;`’$\}$, where $\langle guard\rangle$ is a boolean expression over variables and $\langle update\text{-}list\rangle$‘`;`’ is sequence of updates of variables and channels separated by semicolons. A special reference `null` is defined to denote an undefined channel reference and can be used as a default value. The `init` process consists of a sequence of `run` statements within an atomic block. Note that in Fig. 1 a list statement of the form $\langle foo\text{-}list\rangle$‘$*$’ consists of an $*$-separated sequence of type $\langle foo\rangle$, where $* \in \{$‘`;`’,‘`,`’,‘`::`’$\}$.

In Fig. 2 we present an example specification (from [DM08]) of a message passing system. It consists of three *server* processes, six *client* processes and three *load-balancer* processes. A particular *client* has been blocked by the system, indicated by the global *pid* variable *blocked_client*. A *load-balancer* process continuously receives requests sent by *client* processes. A request consists of two parts: the identity of a *client* (derived from its `_pid` variable), and the input channel of the *client*. If the message originates from the blocked *client* then the *load-balancer* returns the value 0, indicating that the request has been denied. Otherwise the *load-balancer* forwards the name of the input channel of the given *client* to the *server* with the shortest queue of incoming messages (choosing non-deterministically between *server*s which share the shortest queue length). On receiving a *client* channel name, a *server* uses it to send the value 1 to the *client*.

3 Embedding Promela-Lite

We mechanise the language in two phases. First we define the types, syntax and type system of the language. The PVS type checker checks the definition types of the language terms and ensures that the type system is well-defined. We then define the semantics and other related definitions that are needed for our major theorem. Finally, we define and prove this theorem in PVS using these definitions and following the hand proofs published in earlier work [DM08].

3.1 Types

The language has primitive types *int* (integers) and *pid* (process ids). The basic channel type has the form $chan\{\overline{T}\}$, where $\overline{T}$ denotes a comma separated list of types. Intuitively, this definition allows the channel type to be recursively defined as a list of types, including the channel type.

⟨*spec*⟩ ::= ⟨*channel*⟩* ⟨*global*⟩* ⟨*proctype*⟩$^+$ ⟨*init*⟩

⟨*channel*⟩ ::= ⟨*chantype*⟩ ⟨*name*⟩ = [⟨*number*⟩] of { ⟨*type-list*, ‘,’⟩ } ;

⟨*global*⟩ ::= ⟨*type*⟩ ⟨*name*⟩ = ⟨*number*⟩ ;

⟨*proctype*⟩ ::= ⟨*name*⟩ (⟨*param-list*, ‘;’⟩$^?$) { do ⟨*statement-list*, ‘::’⟩ od }

⟨*param*⟩ ::= ⟨*type*⟩ ⟨*name*⟩

⟨*statement*⟩ ::= atomic { ⟨*guard*⟩ -> ⟨*update-list*, ‘;’⟩ }

⟨*init*⟩ ::= init { atomic ⟨*run-list*, ‘;’⟩ }

⟨*run*⟩ ::= run ⟨*name*⟩ (⟨*arg-list*, ‘,’⟩$^?$) ;

⟨*guard*⟩ ::= ⟨*expr*⟩ ⋈ ⟨*expr*⟩
  (where ⋈ ∈ {==, !=, <, <=, >, >=})
  | nfull (⟨*name*⟩)
  | nempty (⟨*name*⟩)
  | ! ⟨*guard*⟩
  | ⟨*guard*⟩ && ⟨*guard*⟩
  | ⟨*guard*⟩|| ⟨*guard*⟩
  | (⟨*guard*⟩)

⟨*expr*⟩ ::= ⟨*name*⟩
  | ⟨*number*⟩
  | _pid
  | null
  | len (⟨*name*⟩)
  | (⟨*expr*⟩)
  | ⟨*expr*⟩ ∘ ⟨*expr*⟩ (where ∘ ∈ {+, -, *})

⟨*update*⟩ ::= skip
  | ⟨*name*⟩ = ⟨*expr*⟩
  | ⟨*name*⟩ ? ⟨*name-list*, ‘,’⟩
  | ⟨*name*⟩ ! ⟨*expr-list*, ‘,’⟩

⟨*arg*⟩ ::= ⟨*name*⟩
  | ⟨*number*⟩
  | null

⟨*name*⟩ ::= an alpha-numeric string

⟨*number*⟩ ::= a positive integer

Figure 1: Syntax of Promela-Lite.

To encode a language in PVS we must define the available types of the language. The primitive types of `int` and `pid` can be easily defined in PVS. However, due to the recursive nature of the channel, all types are defined as a `DATATYPE` in PVS. The type syntax of Promela-Lite and the PVS definition is shown in Fig. 3. A type, `Name`, is defined to represent variable names. We define a function to map a variable name to its type. It allows us to identify the type of a variable and use an appropriate semantic definition.

3.2 Syntax

Proofs about a language with a BNF style syntax definition often require induction over the terms of the language. The syntax definition can be directly encoded using abstract datatypes. PVS generates an induction scheme for the abstract datatypes. A property *p* on terms can be proved by showing that it holds for all atoms and that it holds for all operators if it holds for the subterms.

First, we define a datatypes for an expression (`expr`) and for a guard (`guard`), each contain-

```
chan se1 = [3] of {chan{int}};
chan se2 = [3] of {chan{int}};
chan se3 = [3] of {chan{int}};

chan lb1 = [1] of {pid,chan{int}};
chan lb2 = [1] of {pid,chan{int}};
chan lb3 = [1] of {pid,chan{int}};

chan cl1 = [1] of {int}; chan cl2 = [1] of {int};
chan cl3 = [1] of {int}; chan cl4 = [1] of {int};
chan cl5 = [1] of {int}; chan cl6 = [1] of {int};

pid blocked_client = 9;

proctype loadbalancer(chan{pid,chan{int}} in;
           chan{int} client_link; pid client_id; int pc) {
  do
    :: atomic { pc==1 && nempty(in) -> in?client_id,client_link; pc = 2 }
    :: atomic { pc==2 && client_id!=blocked_client -> pc = 3 }
    :: atomic { pc==2 && client_id==blocked_client && nfull(client_link)
                -> client_link!0; pc = 4 }
    :: atomic { pc==3 && len(se1)<=len(se2) && len(se1)<=len(se3) && nfull(se1)
                -> se1!client_link; pc = 4 }
    :: atomic { pc==3 && len(se2)<=len(se1) && len(se2)<=len(se3) && nfull(se2)
                -> se2!client_link; pc = 4 }
    :: atomic { pc==3 && len(se3)<=len(se1) && len(se3)<=len(se2) && nfull(se3)
                -> se3!client_link; pc = 4 }
    :: atomic { pc==4 -> client_id = 0; client_link = null; pc = 1 }
  od
}

proctype server(chan{chan{int}} in; chan{int} client_link; int pc) {
  do
    :: atomic { pc==1 && nempty(in) -> in?client_link; pc = 2 }
    :: atomic { pc==2 && nfull(client_link) -> client_link!1; pc = 3 }
    :: atomic { pc==3 -> client_link = null; pc = 1 }
  od
}

proctype client(chan{int} in; chan{pid,chan{int}} lb; int response; int pc) {
  do
    :: atomic { pc==1 && nfull(lb) -> lb!_pid,in; pc = 2 }
    :: atomic { pc==2 && nempty(in) -> in?response; pc = 3 }
    :: atomic { pc==3 -> response = -1; pc = 1 }
  od
}

init {
  atomic {

    run server(se1,null,1); run server(se2,null,2); run server(se3,null,3);

    run loadbalancer(lb1,null,0,1); run loadbalancer(lb2,null,0,1);
    run loadbalancer(lb3,null,0,1);

    run client(cl1,lb1,-1,1); run client(cl2,lb1,-1,1); run client(cl3,lb2,-1,1);
    run client(cl4,lb2,-1,1); run client(cl5,lb3,-1,1); run client(cl6,lb3,-1,1);

  }
}
```

Figure 2: Promela-Lite specification of a load-balancing system.

$\langle type \rangle$ ::= `int`
  | `pid`
  | $\langle chantype \rangle$

$\langle chantype \rangle$ ::=
$\langle recursive \rangle^{?}$ `chan` $\{\langle type\text{-}list, `,' \rangle\}$

```
pid : TYPE={n:int|0<=n AND n<=MAX}
Types : DATATYPE
 BEGIN
   int(i:int): int?
   pd(p:pid) : pid?
   channel(chlen: int,
     type_list:list[Types]): chan?
 END Types
Name : TYPE
typeof : [Names -> Types]
```

Figure 3: Promela-Lite type syntax and PVS definition

ing constructors, accessors and recognizers. In the `guard` datatype, constructors are defined for boolean operators, and relational operators. The definition for $\langle expr \rangle \bowtie \langle expr \rangle$ is divided into two parts: one for ($\bowtie \in$ {`==`, `!=`}), and another for ($\bowtie \in$ {`<`, `<=`, `>`, `>=`}). Later in the type system definition, we show how these two definitions are used separately. In the definition of guard, two constructors are defined to check the status of a channel (`nfull, nempty`). An update (`update`) consists of a `skip`, an assignment, or a read/write from/to a channel (denoted ? and ! respectively). The PVS definitions of `expr`, `guard` and `update` are shown in Fig. 4. Note that we do not use the symbols "$=$" or "$<$" directly as they cause a typing conflict in PVS.

```
expr: DATATYPE                  guard : DATATYPE
  BEGIN                           BEGIN
   +(e1,e2: expr) : plus?          rel(e1,e2:expr): rel?
   -(e1,e2: expr) : minus?         eq(e1,e2:expr) : eq?
   *(e1,e2: expr) : star?          Nt(g:guard)    : not?
   name(n: Names) : name?          /\ (g1,g2:guard): and?
   len(nm: Names) : len?           \/ (g1,g2:guard): or?
              nul : nul?           nfull(n:Names) : nfull?
      num(n: int) : num?           nempty(n:Names): nempty?
      pid(p: pid) : pd?          END guard
 END expr
        update : DATATYPE
        BEGIN
         skip                                 : skip?
         assign(x:Names, e:expr)              : assign?
         cin(c:Names,namelist:list[Names])    : cin?
         cout(c:Names, exprlist: list[expr]): cout?
        END update
```

Figure 4: Syntax of expression, guard and update in PVS

3.3 Type System

Promela-Lite typing rules are defined following the notation used in [Car97] and ensure that the language terms are well-formed. A Promela-Lite specification $\mathscr{P}$ is well-typed if its statements and declarations are well-formed according to these rules. Typing rules for `expr`, `guard` and `update` in PVS are shown in Fig 5.

Each expression in `expr` is type checked, ensuring it is well-typed according to the typing rules. For example, the arithmetic expression `+(e1,e2)` returns an `int` value if the constituent expressions, `e1` and `e2`, are of type `int`.

```
chktype_expr(e:expr,t:Types):RECURSIVE bool=
 CASES e OF
  +(e1,e2): EXISTS (t1:Types): int?(t1) AND
            chktype_expr(e1, t1) AND
            chktype_expr(e2, t1) AND int?(t),
  ...
       nul: chan?(t),
    pid(p): pid?(t),
  num(num): int?(t),
   name(n): (int?(types(n)) AND int?(t)) OR
            (pid?(types(n)) AND pid?(t)) OR
            (chan?(types(n)) AND chan?(t)),
   len(nm): chan?(types(nm)) AND int?(t),
  ENDCASES
 MEASURE e BY <<

chktype_guard(g:guard) : RECURSIVE bool =
 CASES g OF
   rel(e1,e2): EXISTS (t:Types): int?(t)AND
               chktype_expr(e1,t) AND
               chktype_expr(e2,t),
   eq(e1,e2) : EXISTS (t:Types):
               chktype_expr(e1,t) AND
               chktype_expr(e2,t),
   Nt(g1)    : chktype_guard(g1),
   /\ (g1,g2) : chktype_guard(g1) AND
               chktype_guard(g2),
   \/(g1,g2) : chktype_guard(g1) AND
               chktype_guard(g2),
   nfull(n)  : chan?(types(n)),
   nempty(n) : chan?(types(n))
  ENDCASES
 MEASURE g BY <<

 chktype_update(u:update) : bool =
 CASES u OF
  skip            : TRUE,
  assign(x,e)     : EXISTS (t:Types):
                    chktype_expr(e,t) AND t = types(x) AND
                    NOT(chan?(types(x))),
  cout(ch,explst): chan?(types(ch)) AND
                    compare_list(type_list(types(ch)),explst),
  cin(ch,nmlist) : chan?(types(ch)) AND chlen(types(ch)) > 0 AND
                      notchan(name2type(nmlist)) AND
                    compare_list(type_list(types(ch)),nmlist) AND
                      cons?(nmlist) AND diff(nmlist)
 ENDCASES
```

Figure 5: Typing rules for expression and guard and update statements in PVS

The typing rules for `guard` include both `expr` and `guard` and we use the type system of `expr` in the definition. Two typing rules are defined for relational operators: `rel(e1,e2)` where the only allowed type is `int`, and `eq(e1,e2)` where any type (T) is allowed.

The boolean function `compare_list` is defined within the PVS channel write definition (`cout`) to ensure that the types of the expressions to be written ($\overline{e}$) to the channel are the same as that of the channel. The functions `notchan` and `diff` are defined for a channel read to ensure that the variables ($\overline{x}$) to be updated are different and not of type *chan*.

The other language terms can be defined using these definitions. For example, the type system for a statement ($\langle$*guard*$\rangle$ `->` $\langle$*update-list*, '`;`'$\rangle$) can be defined using the type systems of both `guard` and `update`.

4 The Automorphism Theorem

In this section we define a Kripke structure, a static channel diagram and automorphism groups of both. We then give the main theorem to be mechanically proved, namely the Correspondence theorem, which relates these two automorphism groups (for a given specification).

Definition 1 A Kripke structure is a tuple $\mathscr{M} = (S, S_0, R)$ where:

- S is a finite set of states
- $S_0 \subseteq S$ is a set of initial states
- $R \subseteq S \times S$ is a transition relation.

A path in $\mathscr{M}$ from a state $s \in S$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ where $s_0 = s$, such that for all $i > 0$, $(s_{i-1}, s_i) \in R$. A state $s \in S$ is *reachable* if there is a path $s_0, s_1, \ldots, s, \ldots$ in $\mathscr{M}$ where $s_0 \in S_0$. A transition $(s,t) \in R$ is *reachable* if s is a reachable state.

Definition 2 Let $\mathscr{M} = (S, S_0, R)$ be a Kripke structure. An *automorphism* of $\mathscr{M}$ is a permutation $\alpha : S \to S$ which preserves the transition relation (R) and set of initial states. That is α satisfies:

- For all $s,t \in S$, $(s,t) \in R \Rightarrow (\alpha(s), \alpha(t)) \in R$
- $\alpha(s_0) \in S_0$ for all $s_0 \in S_0$.

In fact, we will assume that there is only one initial state, i.e. $S_0 = \{s_0\}$.

The *static channel diagram* ($SCD(\mathscr{P})$) of a Promela-Lite specification ($\mathscr{P}$) is a graphical structure extracted by syntactic inspection of the specification and it can be seen as a static approximation of the communication structure for the specification.

The static channel diagram is defined in PVS as a coloured graph consisting of a set of vertices and a set of edges. The vertices consist of a set of *pid*s and a set of channels. The edges are constructed by taking one vertex from each of these sets. We define a colouring function to add colours to both *pid*s and channels so that *pid*s of the same proctype have the same colour and channels of the same type have the same colour.

```
pidset: TYPE = set[pid]
chset: TYPE = set[chan]
vertices: TYPE+ = [pidset, chset]
edge: TYPE = [# pd: pid, ch: chan #]
```

```
edges : TYPE = setof[edge]
graph: TYPE+ = [vertices, edges]
SCD: TYPE = {g: graph | FORALL (e:edge): g‘2(e) IMPLIES
                        (g‘1)‘1(pd(e)) AND (g‘1)‘2(ch(e))}
```

Definition 3 Let $\Gamma = (V,E,\Upsilon)$ be a coloured digraph where Υ is a colouring of (V,E) and α a permutation of V. Then α is an *automorphism* of Γ if the following conditions are satisfied:

- For all $(u,v) \in E$, $(\alpha(u),\alpha(v)) \in E$
- For all $v \in V$, $\Upsilon(v) = \Upsilon(\alpha(v))$.

An automorphism of the static channel diagram is an automorphism of a coloured graph. To define the automorphism we define a bijection for the vertices. The vertices consist of both *pid*s and channels and bijections are defined for them both.

```
p_perm: TYPE = (bijective?[pid, pid])
c_perm: TYPE = (bijective?[chan, chan])
Automorph(G): TYPE = {g: SCD | FORALL (p,c,eg):
            G‘2(eg) AND pd(eg)=p AND ch(eg)=c IMPLIES
            EXISTS (pp,cp):
            g‘2((#pd := pp(p), ch := cp(c)#)) AND
            color(p) = color(pp(p)) AND
            color(c) = color(cp(c)) }
Alpha(p) : TYPE ={ p1: pid |
                 EXISTS (g:SCD, ag:Automorph(g),pp):
                 (g‘1)‘1(p) AND (ag‘1)‘1(p) AND p1 = pp(p)}
%%Similar definition is also defined for channel
```

Given an expression e, guard g, update u and statement s of $\mathscr{P}$, the permutations $\alpha(e), \alpha(g), \alpha(u)$ and $\alpha(s)$ are obtained by replacing each occurrence of static channel name and *pid* literal with its respective permutations.

The automorphisms of a Kripke structure $\mathscr{M}$ form a group under the composition of mappings denoted $Aut(\mathscr{M})$. In a model of a concurrent system with many replicated processes, Kripke structure automorphisms typically involve the permutation of process identifiers throughout all states of the model. There is a group G which permutes the set of process identifiers, and an action of G on S. G partitions the state set S into equivalence classes called *orbits*. A quotient Kripke structure $\mathscr{M}_G$ can be constructed by using a representative from each orbit. The state space of the quotient model is usually smaller than the original state space making it convenient to verify larger structures.

This paper does not concern symmetry reduction, rather the detection of symmetry (to be later used in reduction). Symmetry reduction involves replacing sets of symmetrically equivalent states by a single representative state. Details of the technique of symmetry reduction can be found in [CEF$^+$96, ES96, ID96]. Our symmetry detection technique is based on a correspondence between the automorphisms of the static channel diagram and automorphisms of the underlying Kripke structure. By showing this correspondence we can establish that the symmetries detected by analysing static channel diagram (using computational graph theoretic tools like, for example, NAUTY [McK90]) infer the symmetries in the Kripke structure and these symmetries can be used for reduced model checking. In this paper, we prove the following theorem:

Theorem 1 *Let $\mathscr{P}$ be a Promela-Lite specification, and $\alpha \in Aut(SCD(\mathscr{P}))$ and let ρ be the permutation representation of $Aut(SCD(\mathscr{P}))$. If α is valid for $\mathscr{P}$ then $\rho(\alpha) \in Aut(\mathscr{M})$.*

Note that an automorphism is said to be *valid* for $\mathscr{P}$ if it maps $\mathscr{P}$ to an equivalent specification, i.e. one that is identical up to ordering of (and within) statements. The proof of the theorem uses four supporting lemmas. The mechanical proof relies upon giving definitions of these lemmas in PVS, which are shown in the following section.

5 Proof Mechanisation

In this section we give a flavour of our proof mechanization. Note that for space reasons several details are omitted e.g. the (rather complex) PVS definition of a state s, and the action of a static channel diagram automorphism α on s, $\alpha(s)$, together with some subcases of lemmas. Full details can be obtained from the authors.

5.1 Expressions

Let e be an expression in a proctype p. The result of evaluating e for process i (of type p) at a state s is denoted $eval_{p,i}(s,e)$.

Lemma 1 *Let $\alpha \in Aut(SCD(\mathscr{P}))$ and let e be an expression in $\mathscr{P}$.*
If $e : int$ then

$$eval_{p,i}(s,e) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e))$$

and if $e : pid$ or $e : chan\{\overline{T}\}$ then

$$eval_{p,i}(\alpha(s), \alpha(e)) = \alpha(eval(s,e))$$

To prove the lemma it is required to evaluate expressions of type int, pid or $chan$. The rules for evaluating expressions are shown in Fig. 6.

- $eval_{p,i}(s,x) = a$ if $(x = a) \in s$ (i.e. x is a global variable)
- $eval_{p,i}(s,c) = c$ if c is a static channel name or `null`
- $eval_{p,i}(s,a) = a$ if $a \in \mathbb{Z}$
- $eval_{p,i}(s, \_\texttt{pid}) = i$
- $eval_{p,i}(s, \texttt{len}(c)) = k$ if c is a static channel and $(c \in s\ (0 \leq k \leq cap(c))$
- $eval_{p,i}(s, \texttt{len}(\texttt{null})) = 0$ $(p[i].x = c) \in s$
- $eval_{p,i}(s, e_1 \circ e_2) = eval_{p,i}(s,e_1) \circ eval_{p,i}(s,e_2)$ (where $\circ \in \{+,-,*\}$).

  $\ldots$

Figure 6: Promela-Lite expression evaluation

For expressions of type *int* the proof of Lemma 1 is a direct implication of permutation over expressions. For arithmetic expressions the results hold by induction, e.g., when $e = e_1 + e_2$, we show that,

$$eval_{p,i}(s,(e_1+e_2)) = eval_{p,\alpha(i)}(\alpha(s),(\alpha(e_1)+\alpha(e_2)))$$

We define a lemma for the addition operation for expressions of type *int*.

```
arith_lemma : LEMMA
 evaluate(s,i)(+(e1,e2)) =
       evaluate(alpha(s),alpha(i))(+(alpha(e1),alpha(e2)))
```

It is fairly straight forward to prove this lemma in PVS. However, the proofs for *pid* and *chan* are more complex. The lemmas for other expressions are similar.

5.2 Guard Statements

In the following, the relation $s \models_{p,i} g$ states that a guard g is satisfied for a process $p(i)$ at the state s.

Lemma 2 *If $\alpha \in Aut(SCD(\mathscr{P}))$ and g is a guard in $\mathscr{P}$ then*

$$s \models_{p,i} g \;\Leftrightarrow\; \alpha(s) \models_{p,\alpha(i)} \alpha(g)$$

Promela-Lite guards consist of a boolean combination of propositional formulas. The definition of the relation $s \models_{p,i} g$ is shown in Fig. 7.

- $s \models_{p,i} e_1 \bowtie e_2 \Leftrightarrow eval_{p,i}(s,e_1) \bowtie eval_{p,i}(s,e_2)$ (where $\bowtie \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$)
- $s \models_{p,i} \texttt{nfull}(c) \Leftrightarrow (c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ and $cap(c) > m$, where c is a static channel
- $s \models_{p,i} \texttt{nempty}(c) \Leftrightarrow (c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ and $m > 0$, where c is a static channel
- $s \models_{p,i} \texttt{nfull}(x)/\texttt{nempty}(x) \Leftrightarrow (p[i].x = c) \in s$ and $s \models_{p,i} \texttt{nfull}(c)/\texttt{nempty}(c)$,
- $s \models_{p,i} \texttt{!}g$ iff $s \not\models_{p,i} g$
- $s \models_{p,i} g_1 \texttt{\&\&} g_2$ iff $s \models_{p,i} g_1$ and $s \models_{p,i} g_2$
- $s \models_{p,i} g_1 \texttt{||} g_2$ iff $s \models_{p,i} g_1$ or $s \models_{p,i} g_2$
- $s \models_{p,i} \texttt{(}g\texttt{)}$ iff $s \models_{p,i} g$.

Figure 7: Satisfaction of guards

The proof of this lemma uses the proof of Lemma 1. We prove Lemma 2 with PVS for each type of guard statement. As an example, consider the case $g = e_1 \bowtie e_2$, where e_1 and e_2 are expressions of type *pid*. For such a guard, we formulate the following equation from Lemma 2.

$$s \models_{p,i} e_1 \bowtie e_2 \quad\Leftrightarrow\quad \alpha(s) \models_{p,\alpha(i)} \alpha(e_1) \bowtie \alpha(e_2)$$

For a guard consisting of boolean operators, e.g. $g = g_1 \texttt{\&\&} g_2$, we formulate the following equation:

$$s \models_{p,i} g_1 \texttt{\&\&} g_2 \quad\Leftrightarrow\quad \alpha(s) \models_{p,\alpha(i)} \alpha(g_1) \texttt{\&\&} \alpha(g_2)$$

Both guard statements are defined as follows:

```
eqlema : LEMMA
  Guard?(eq(e1,e2))(i,s) =
      Guard?(eq(alpha(e1),alpha(e2)))(alpha(i),alpha(s))
andlema : LEMMA
  Guard?(/\(g1,g2))(i,s) =
      Guard?(/\(alpha(g1),alpha(g2)))(alpha(i),alpha(s))
```

5.3 Update Statements

Here $exec_{p,i}(s,u)$ denotes the result of applying an update u to state s. A statement is said to be well-defined if the application condition of the statement is sufficient to ensure that the update (or sequence of updates) results in a well-defined state.

Lemma 3 *Let u be an update of $\mathscr{P}$, $\alpha \in Aut(SCD(\mathscr{P}))$ and s a state such that $exec_{(}s,u)$ is well-defined. Then $exec_{p,\alpha(i)}(\alpha(s),\alpha(u)) = \alpha(exec_{p,i}(s,u))$.*

To prove Lemma 3, first we define the update execution rules. The update execution rules are described in Table 1. We prove Lemma 3 by proving it for each type of update statement. The proof for `skip` is immediate.

Table 1: Update execution rules

u	**Conditions on** s	**Resulting state** $exec_{p,i}(s,u)$
‘`skip`’	none	s
‘x `=` e’	$(var(x) = a) \in s$	$\big(s \setminus \{(var(x) = a)\}\big) \cup$ $\{(var(x) = eval_{p,i}(s,e))\}$
‘$c\,!\,e_1, e_2, \ldots, e_k$’	$(c = [a_1, a_2, \ldots, a_m]) \in s$ $s \models_{p,i}$ `nfull`(c)	$\big(s \setminus \{(c = [a_1, a_2, \ldots, a_m])\}\big) \cup$ $\{(c = [a_1, a_2, \ldots, a_m, (eval_{p,i}(s, e_1),$ $eval_{p,i}(s, e_2), \ldots, eval_{p,i}(s, e_k))])\}$
‘$c?x_1, x_2, \ldots, x_k$’	$(c = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}),$ $a_2, \ldots, a_m]) \in s$ $s \models_{p,i}$ `nempty`(c) $(var(x_j) = b_j) \in s$ $(1 \le j \le k)$	$\big(s \setminus \{(c = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), a_2, \ldots, a_m]),$ $(var(x_1) = b_1), (var(x_2) = b_2), \ldots,$ $(var(x_k) = b_k)\}\big)$ $\cup\{(c = [a_2, \ldots, a_m]), (var(x_1) = a_{1,1}),$ $(var(x_2) = a_{1,2}), \ldots, (var(x_k) = a_{1,k})\}$
‘$x\,!\,e_1, e_2, \ldots, e_k$’	$(p[i].x = c) \in s$	$exec_{p,i}(s, $‘$c\,!\,e_1, e_2, \ldots, e_k$’$)$ (if well-defined)
‘$x?x_1, x_2, \ldots, x_k$’	$(p[i].x = c) \in s$	$exec_{p,i}(s, $‘$c?x_1, x_2, \ldots, x_k$’$)$ (if well-defined)

Rules interpreted in the context of process i, an instantiation of proctype p

Assignment: For an assignment statement of the form $(x = e)$, where x is a variable and e an expression, we prove the following:

$$exec_{p,\alpha(i)}(\alpha(s),\alpha(x = e)) \quad = \quad \alpha(exec_{p,i}(s, x = e))$$

Following the update rule for an assignment statement in Table 1, we define a function for `update` and the following lemma in PVS.

```
update_assgn : LEMMA
   exec(alpha(s),alpha(i),alpha(assign(x,e))) =
     alpha(exec(s,i,assign(x,e)))
```

Channel Write: Let $e_1, e_2, \ldots, e_k$ be the expressions whose values are to be written to a channel

x in a state s.

$$exec_{p,\alpha(i)}(\alpha(s), `\alpha(x)!\alpha(e_1),\alpha(e_2),\ldots,\alpha(e_k)') \quad = \quad \alpha(exec_{p,i}(s, `x!e_1,e_2,\ldots,e_k'))$$

In PVS, we define the rules for writing to a channel and define the following lemma:

```
update_chwr : LEMMA
   exec(alpha(s),alpha(i),alpha(c_nm),alpha(ex_lst)) =
     alpha(exec(s,i, c_nm, ex_lst))
```

Channel Read: Let $x_1,x_2,\ldots,x_k$ be the variables to be assigned values reading from a channel x in a state s.

$$exec_{p,\alpha(i)}(\alpha(s), `\alpha(x)?x_1,x_2,\ldots,x_k') \quad = \quad \alpha(exec_{p,i}(s, `x?x_1,x_2,\ldots,x_k'))$$

After defining the rules for reading from channel we define the following lemma:

```
 update_read: LEMMA
   exec(alpha(s),alpha(i), alpha(c_nm), v_lst)) =
    alpha(exec(s,i, c_nm, v_lst))
```

The following lemma shows the result of applying a sequence of update statements $u_1,u_2,\ldots,u_k$ in a state s. This can be proved using Lemma 3.

Lemma 4 *Let $u_1,u_2,\ldots,u_k$ be updates of $\mathscr{P}$, $\alpha \in Aut(SCD(\mathscr{P}))$ and s be a state such that $exec_{p,i}(s,u_1;\ u_2;\ \ldots;\ u_k)$ is well-defined. Then*

$$exec_{p,\alpha(i)}(\alpha(s),\alpha(u_1);\ \alpha(u_2);\ \ldots;\ \alpha(u_k)) = \alpha(exec_{p,i}(s,u_1;\ u_2;\ \ldots;\ u_k))$$

Proof of Theorem 1. According to Definition 2, we must show that any automorphism α preserves transitions and fixes the initial state.

If $(s,t) \in R$ then there is a process with pid i such that $proctype(i) = p$ (for some proctype p), and a statement z in p such that the guard of z holds for process i at s, and execution of the updates of z by process i at s leads to state t. Since $\alpha(\mathscr{P}) \equiv \mathscr{P}$ the statement $\alpha(z)$ (possibly re-arranged) also appears in proctype p. By Lemma 2, the guard of $\alpha(z)$ holds for process $\alpha(i)$ at $\alpha(s)$, and by Lemma 4, execution of the updates of $\alpha(z)$ by process $\alpha(i)$ at $\alpha(s)$ leads to state $\alpha(t)$. Therefore $(\alpha(s),\alpha(t)) \in R$. Proof that α fixes the initial state is omitted here. □

We define a transition involving a *pid* variable and the transition preservation property in PVS:

```
step(s,t): bool = EXISTS (i : pid, z : Statement):
                   statement_in_proc(i)(z) AND
                   Guard?(guard(z))(i,s) AND
                   t = exec(s, i, updates(z))
Automorph: THEOREM
       step(s,t) => step(alpha(s),alpha(t))
```

This property is proved by using Lemma 2 and Lemma 4.

Similar cases involving local variables and channels are omitted.

5.4 PVS vs other provers

The most difficult aspect of the presented work was that of embedding the language constructs and semantics into the prover. There were various possibilities here, we chose a way that was convenient for our particular proofs.

In any proof, one of the main purposes is to decompose the goal into one or more simpler subgoals and find suitable proof steps for each of them. We had the advantage of having access to existing hand proofs, which indicated ways to decompose the proof of Theorem 1. It would have been feasible to use an alternative prover in a similar way. Each prover has its own way of defining language terms automation facility. PVS provides automated support for combining proof steps, which would have had to have been performed individually using other provers.

6 Related Work

Promela-Lite and hand proofs of the Automorphism Theorem are presented in [DM08]. No previous work has been carried out to embed Promela-Lite in a theorem prover or mechanise these proofs.

In [TBL10] the soundness of symmetry reduction for model checking is proved using the in the B-Method [Abr96] and its associated tools. They do not consider symmetry detection as we do, but their work represents one of the few examples where one formal technique is used to verify another.

In [Hel06], the formal semantics of a specification language Ocsid is embedded into PVS. A parallel approach is taken where the language is embedded by using both shallow and deep embedding. Language syntax and corresponding semantics are embedded using deep embedding. A simple specification is embedded using shallow embedding. A correspondence proof is shown between the two embeddings. The syntax is embedded using an abstract datatype mechanism and semantics are defined recursively to return a value type. We use a similar approach.

The specification language A_g is a First-Order Dynamic Logic of Fork Algebra. In [POS04], the semantics of the language is embedded into the PVS theorem prover allowing for the construction of specifications and readable proofs of various properties of the specifications. The steps taken to embed the syntax and the semantics, and the features of the theorem prover have informed our work.

7 Conclusions and Future Directions

We have shown how a mechanical formal verification approach can be used in practice to verify a formal method - a symmetry detection technique for model checking. Our purpose was to gain confidence in our symmetry detection technique and to find a feasible mechanisation technique with which to prove properties of a language, thereby minimising the need for hand proofs.

We have presented a case study, namely the proof of a correspondence theorem supported by two significant lemmas for the modelling language Promela-Lite. With the strong datatype support of PVS, such as abstract datatypes and predicate subtypes, we have succinctly defined the syntax and type system of the language. The formulation of each lemma requires additional definitions and, crucially, a clear understanding of the related semantics. In the hand proof it is easy to be imprecise about various definitions, and typing of the rules. The mechanisation forces us to be strict about definitions and datatypes. Often the process of performing a proof is more instructive than getting a final yes/no answer. We have used the theorem prover as a proof checker. To do this it is necessary to fully understand the reasoning steps of the theorem prover.

We have followed a systematic approach where we have defined the syntax, the type system and the semantics of the language. Our embedding approach can be easily adapted to other languages. The main challenge of defining any language in a theorem prover is to properly embed the semantics. Our experience from this work suggests that the verification of such language properties for a similar language can be achieved with a reasonable amount of effort.

Mechanical proof should be applied to check the soundness of a verification implementation *during development*. Our current goal is to use the structure of our mechanical proof to prove the soundness of a symmetry detection technique for a new specification language for symmetric probabilistic systems [PM10].

Bibliography

[Abr96] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ARW10] *Proceedings of the 17th workshop on Automated Reasoning (ARW 2010)*. London, UK, March 2010.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Be97] B. Barras, et al. The Coq Proof Assistant Reference Manual : Version 6.1. Technical report 0203, INRIA, August 1997.

[Car97] L. Cardelli. *The Computer Science and Engineering Handbook*. Chapter Type Systems, pp. 2208–2236. CRC Press, Boca Raton, 1997.

[CEF$^+$96] E. Clarke, R. Enders, T. Filkorn, , S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9(1–2):77–104, 1996.

[CGP99] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. MIT Press, 1999.

[DM06] A. F. Donaldson, A. Miller. A Computational Group Theoretic Symmetry Reduction Package for the Spin Model Checker. In *AMAST'06*. LNCS 4019, pp. 374–380. Springer, 2006.

[DM08] A. F. Donaldson, A. Miller. Automatic Symmetry Detection for Promela. *Journal of Automated Reasoning* 41:251–293, 2008.

[Don07] A. Donaldson. *Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking*. PhD thesis, Department of Computing Science, University of Glasgow, UK, 2007.

[ES96] E. Emerson, A. Sistla. Symmetry and model checking. *Formal Methods in System Design* 9(1–2):105–131, 1996.

[GM93] M. Gordon, T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Hel06] J. Helin. Combining Deep and Shallow Embeddings. *ENTCS* 164(2):61–79, 2006.

[Hol03] G. J. Holzman. *The SPIN model checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[ID96] C. Ip, D. Dill. Better verification through symmetry. *Formal Methods in System Design* 9:41–75, 1996.

[McK90] B. McKay. *nauty* user's guide (version 1.5). Technical report TR-CS-90-02, Australian National University, Computer Science Department, 1990.

[ORS92] S. Owre, J. Rushby, N. Shankar. PVS: A Prototype Verification System. In Kapur (ed.), *CADE'92*. LNAI 607, pp. 748–752. Springer-Verlag, June 1992.

[OS93] S. Owre, N. Shanker. Abstract datatypes in PVS. Technical report SRI-CSL-93-9R, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer-Verlag, 1994.

[PM10] C. Power, A. Miller. An approach to probabilistic symmetry reduction. Pp. 32–33 in [ARW10].

[POS04] C. L. Pombo, S. Owre, N. Shankar. A Semantic Embedding of the Ag Dynamic Logic in PVS. Technical report SRI-CSL-02-04, SRI International, Menlo Park, CA, Oct. 2004.

[RAM$^+$93] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, J. van Tassel. Experience with embedding hardware description languages in HOL. In *TPCD'93*. Pp. 129–156. North-Holland, 1993.

[RB08] S. H. Ripon, M. J. Butler. PVS Embedding of cCSP Semantic Models and their Relationship. In Calder and Miller (eds.), *AVoCS'08*. Pp. 128–142. 2008.

[RB09] S. H. Ripon, M. J. Butler. PVS Embedding of cCSP Semantic Models and their Relationship. *ENTCS* 250:103 118, 2009.

[SO99] N. Shankar, S. Owre. Principles and Pragmatics of Subtyping in PVS. In Bert et al. (eds.), *WADT '99*. LNCS 1827, pp. 37–52. Springer-Verlag, September 15-18 1999.

[TBL10] E. Turner, M. Butler, M. Leuschel. A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking. In *Proceedings ABZ'2010*. LNCS, pp. 231–244. Springer-Verlag, 2010.