MECHANIZING INVARIANT PROOFS
OF JOINT ACTION SYSTEMS

P. Kellomäki (Tampere, Finnland)

Abstract. This paper describes a system for mechanically deriving invariants for specifications given as joint action systems. We have implemented a tool which converts specifications in the DisCo specification language to the logic used by the PVS theorem prover. The tool derives instructions for the theorem prover for carrying out most of the proofs without user intervention. As an example, an invariant for a simple communication protocol is derived, and some empirical results about the performance of the system are given.

The main contributions of the paper are the formalization of DisCo specifications in PVS, and the idea of deriving proof strategies for the theorem prover from the original specification.

1. Introduction

This paper describes a system for mechanically deriving invariants for specifications given as joint action systems. We have implemented a tool which converts specifications in the DisCo specification language to the logic used by the PVS [13] theorem prover. The tool also derives instructions for PVS to carry out most of the proof without user intervention. We use a simplified version of the protocol proposed in [14] and analyzed in [8, 9] to show how invariant proofs are obtained with our system.

The main contributions of the paper are the formalization of DisCo specifications in PVS, and the idea of deriving proof strategies for the theorem prover from the original specification.
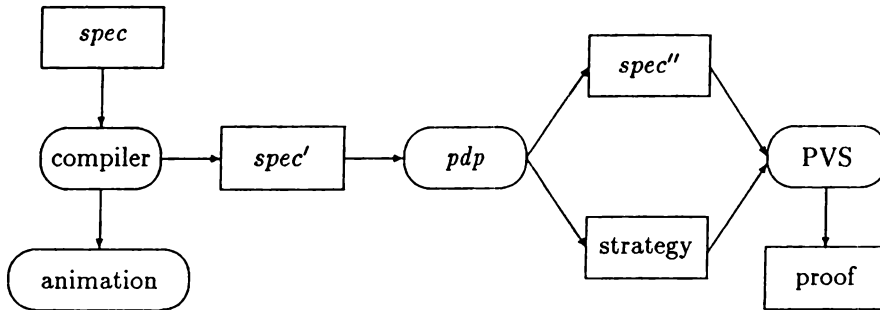

Fig.1. The DisCo specification environment

The *pdp* (short for Prepare DisCo Proof) tool is intended to be a part of the DisCo specification environment [7, 10, 15]. Figure 1 shows the components of the environment, and the path from a specification to an invariant proof. The user writes a specification *spec* in the DisCo specification language and gives the specification to the DisCo compiler. The compiler produces an intermediate representation *spec'* of the specification, which is given to *pdp*. The *pdp* tool then converts *spec'* to *spec''*, which is suitable as input to the PVS theorem prover.

The *pdp* tool also produces a custom *proof strategy* for proving each proposed invariant of *spec*, given in the specification. The strategy contains instructions for the theorem prover as to which inference steps to apply when proving each invariant. Invariant proofs always have the same structure, and the *pdp* tool uses this for automating the routine parts of proofs. When proving an invariant, PVS reads *spec''* and the generated strategy, and executes the steps in the strategy. Any unresolved subproofs are then completed interactively. The *pdp* tool does not yet handle the full DisCo language, but the implemented subset is large enough for writing realistic specifications.

The discussion proceeds as follows. Section 2 briefly introduces joint action systems and their formalization in the PVS logic. Section 3 describes the protocol and its formalization using the joint action approach. The proposed invariant is given and its proof is outlined in Section 4. Some figures about the size of the proof and the number of automated and manual steps are also given in this section. Some related work is discussed in Section 5, and conclusions are

made in Section 6. The Appendices contain the specification as given to the *pdp* tool, the generated PVS theory, and an excerpt of the strategy generated.

2. Joint action systems and DisCo

The *joint action* approach of Back and Kurki-Suonio [1, 2, 3] is a technique for specifying reactive systems. A joint action system consists of a set of *objects*, which can *participate* in *actions* to communicate and alter their local states. A joint action system starts from some initial state, and executes by choosing an enabled action, executing it, choosing again, and so forth, giving an interleaved model of execution. The nondeterministic scheduling of actions facilitates modeling of concurrent systems within a sequential formalism.

The initial state and actions describe which computations are allowed, i.e. they specify *safety* properties. However, they do not make any statements about *liveness* properties. In this paper, only safety properties are considered, and consequently our formalization only deals with safety properties.

When a joint action is executed, none of the participants is specified to initiate the execution. Initiation of an action, and the exact method of communication used are viewed as implementation details. There is also no concept of processes as such, even though the objects of a joint action system often correspond to processes in the eventual implementation. This lets the specifier to concentrate only on the activities that take place, without having to decide which of the participants are active and which are passive.

The DisCo specification language is a concrete specification language based on the joint action approach. Figure 2 gives a very simple example of a specification written in the language.

```
system S is
    class C is
        i    integer;
    end;
    action inc by o : C is
    when true do
        o.i  :=  o.i + 1;
    end;
end;
```

Fig.2. A DisCo specification

The language allows for incremental specification, where specifications can be refined using superposition, and new specifications can be composed out of independently specified parts. Refining a DisCo specification by superposition always preserves safety properties. The methodology for writing specifications in the language encourages to start with a very simple model of the system, and gradually to add new properties one by one. In this paper we are interested only in the result of the incremental specification process, so we refer the reader to [7] for a more detailed discussion of the language.

2.1. Formalizing joint action systems in PVS

We now sketch our formalization of joint action systems. It is very similar to [7], which gives the semantics of the DisCo language in terms of Temporal Logic of Actions [11].

2.1.1. State and objects

We use *state* to map variables (attributes of objects) to their values. State is represented as an abstract data type in PVS:

```
state  :  TYPE.
```

To represent objects, we use PVS records. For each class of objects, we define a corresponding record type, whose elements are functions from state to the type of the corresponding attribute. For example, to represent a class C of objects whose instances consist of a single integer attribute i, we use the PVS type

```
C : TYPE = [♯i:  [state->int] ♯].
```

The syntax $[♯ \ldots ♯]$ introduces a record type in PVS. To refer to the value of attribute i of object o in state s, we write $o.i(s)$.[2]

Most of the attributes can be directly mapped to the PVS logic. To represent references (similar to pointers in programming languages), we introduce an extra field *ref* in the record types representing classes, and give an axiom

$$(1) \qquad\qquad ref(a) = ref(b) \Rightarrow a = b$$

for each type. The *ref* field represents the identity of an object.

---

[2] The actual PVS syntax is $i(o)(s)$, but we stick to the more familiar dot notation for record field access. Ultimately, we would like to modify PVS to use the dot notation, since the specifications written in the DisCo language use it.

2.1.2. Actions

Actions are represented as functions of two states. The states represent the state before executing the action and after it has been executed, respectively. For instance, the action *inc* in Figure 2 is represented in PVS as

```
inc(unprimed, primed:  state) :  bool =
  exists (o :  C) :
    o.i(primed) = o.i(unprimed) + 1
    and forall (other :  C)
        other /= c => other.i(primed) = other.i(unprimed).
```

The represetation of *inc* spells out the values of all variables in the system, even those that are not changed by the action. In a system containing many classes, the part telling which objects are not changed can easily be larger than the part actually giving the changes to objects.

Existentially quantifying *o* means that we do not care which object is changed, as long as one object is changed and the others remain unchanged. This corresponds to the "participant" idea of joint action systems.

2.1.3. Proof rules

Our first proof rule is the standard invariant rule

$$(2) \qquad \frac{\begin{array}{c} INIT \Rightarrow P \\ P \wedge [\mathcal{N}] \Rightarrow P' \end{array}}{\Box P}.$$

INIT is a predicate describing the allowed initial states, P is a *state predicate* (a boolean function of state), and $\mathcal{N}$ is the disjunction of all actions of the system. The notation $[\mathcal{N}]$ denotes that either action $\mathcal{N}$ is executed, or nothing changes, and P' refers to the value of predicate P after the action has been executed. We omit the $\Box[\mathcal{N}]$ usually included in the rule to require that only actions in $\mathcal{N}$ are executed, as $\mathcal{N}$ includes by definition all the actions in the system.

Some invariants cannot be proved by using (2) alone, as the proof may rely on some other invariants. Our second proof rule allows us to introduce other invariants as assumptions in a proof:

$$(3) \qquad \frac{\Box P}{P}.$$

The generated PVS files do not yet include this rule.

These proof rules are sound not only for the proposed invariants, but for arbitrary state predicates. This means that one does not need to write a proposed invariant in the DisCo specification (*spec* in Figure 1) in order to prove it. The PVS formalization of these rules is shown in Appendix A.

3. The protocol

We now briefly describe the protocol to be modeled. A network of stations is connected to a common bus. Each station is identified by a unique *id* ranging from 0 to $n-1$, where n is the number of stations in the network.

The stations circulate a token to share the bus. When a station has the token, it transmits its *id* and a message on the bus. The receiving stations read the sender *id* and message from the bus and compute

$$(received\ \ id + 1)\ \ \mod n,$$

and the station whose *id* matches the result receives the token. If a station does not have any data to send when it receives the token, it sends an empty message.

3.1. Objects

There are clearly two classes of objects: stations and buses. Each station contains an *id*, a register to hold a single message, and a reference to a bus object. In addition, it also holds n, which is the number of stations connected to the same bus, and a boolean flag f, whose role will be described in the next section. Class *station* is defined as

(4) $station \stackrel{\triangle}{=} (id\ :\ integer,\ me : integer,\ bu : objid,\ f : bool,\ n : integer)$.

A bus consists of a message and the *id* of the sender of the message. We also include a field *prev*, which stores the previous message on the bus, a boolean flag f, and a *ref* field. The *prev* field will be used when formulating the invariant. Class *bus* is defined by

(5) $bus \stackrel{\triangle}{=} (id : integer,\ me : integer,\ prev : integer,\ f : bool)$.

The Appendices show how classes are represented in the input to *pdp* and in the generated PVS file.

3.2. Actions

To model sending and receiving of messages, we introduce actions *send* and *receive*, which transfer messages between a station and a bus.

In a physical realization of the protocol, the physical properties of the communication medium ensure that all stations have received a message before the next one is sent. We do not model these properties, so a different approach is needed. The boolean flag f is used for synchronizing buses and stations. The *send* action is enabled when the f flags of all stations in a network agree with the value of the f flag of the bus. When *send* is executed, the participating station inverts the value of its f flag, and the value of the flag of the participating bus is also inverted. This disables the *send* action, and it only becomes enabled again when the rest of the stations have inverted their flags. The only way for the other stations to invert their f flag is to execute the *receive* action. Thus, when all the flags agree again, all stations have received the message on the bus.

The actions are defined as follows. We use $send(s, b)$ to denote that the *send* action has two participants, and the prime symbol to refer to the final values of variables.

$$send(s : station, b : bus) \triangleq$$
$$\exists (m : integer) :$$
$$s.bu = b$$
$$\wedge\ \forall(s\ station) : s.bu = b \Rightarrow s.f = b.f$$
$$\wedge\ s.id = (b.id + 1) \bmod s.n$$
(6)
$$\wedge\ b.prev' = b.me$$
$$\wedge\ b.f' = \neg b.f$$
$$\wedge\ s.f' = \neg s.f$$
$$\wedge\ s.me' = m$$
$$\wedge\ b.me' = m$$
$$\wedge\ b.id' = s.id$$

$$receive(s : station, b : bus) \triangleq$$
$$s.bu = b$$
(7)
$$\wedge\ s.f \neq b.f$$
$$\wedge\ s.me' = b.me$$
$$\wedge\ s.f' = b.f$$

4. Proving an invariant

We now set out to prove an invariant for the system. The proposed invariant states that if the f flag of a station agrees with that of the bus, its me attribute also agrees with that of the bus. Otherwise the me attribute agrees with that of the $prev$ attribute of the bus. In terms of the protocol, this means that all stations either have received the current message, or they still have the previous message. Formally the invariant is expressed as

$$
\begin{aligned}
inv &\stackrel{\triangle}{=} \forall(s : station, b : bus) : \\
&(s.bu = b \wedge s.f = b.f \Rightarrow s.me = b.me) \\
&\wedge \ (s.bu = b \wedge s.f \neq b.f \Rightarrow s.me = b.prev)
\end{aligned}
$$

(8)

4.1. Outline of the proof of $\Box inv$

Proofs in the PVS theorem prover are done using a *sequent calculus*. A successful proof forms a tree, where each node is of the form $A_1, A_2, \ldots \vdash$ $\vdash C_1, C_2, \ldots$ (from *antecedents* $A_1, A_2, \ldots$ infer one of *consequents* $C_1, C_2, \ldots$). The root node is of the form

$$\vdash thm,$$

where thm is the theorem to be proved, and the leaf nodes are sequents that are recognized as true. Branching of the tree corresponds to "backwards" application of proof rules.

To prove that $\Box inv$ indeed is an invariant, we start with the sequent

(9) $\vdash \Box inv.$

We then introduce an instance of the invariant rule (2) as an assumption, giving

$$
\begin{aligned}
&(\forall(s, sp : state) : \\
&\quad (INIT(s) \Rightarrow inv(s)) \\
&\quad \wedge (inv(s) \wedge (receive(s, sp) \vee send(s, sp)) \Rightarrow inv(sp))) \\
&\Rightarrow \Box inv \\
&\vdash \\
&\Box inv
\end{aligned}
$$

(10)

Skolemizing and decomposing (10) gives us two sequents,

(11) $\Box inv \vdash \Box inv$,

which is trivially true, and

$$\vdash$$

$$(\forall(s, sp : state) :$$

(12) $$(INIT(s) \Rightarrow inv(s))$$

$$\wedge (inv(s) \wedge (receive(s, sp) \vee send(s, sp)) \Rightarrow inv(sp)),$$

$$\Box inv$$

Decomposing (12) further gives us the sequents

(13) $\vdash INIT(s!1) \Rightarrow inv(s!1)$

and

$$inv(s!1),$$

$$receive(s!1, sp!1) \vee send(s!1, sp!1)$$

(14) $$\vdash$$

$$inv(sp!1),$$

where $s!1$ and $sp!1$ are skolem constants introduced for the universally bound variables in (12). The rest of the proof is omitted for brevity, but the proof proceeds by expanding the definitions of *receive* and *send*, and simplifying the sequents until they can be resolved with the prover.

4.2. Efficiency of the generated strategies

The proof of $\Box inv$ generates a proof tree consisting of 201 nodes, 20 of which are leaf nodes. These nodes are generated automatically when PVS follows instructions in the custom proof strategy. Of the leaf nodes, one is

$$inv(s) \Rightarrow inv(s),$$

which is trivially true, and 14 are of the form

$$\ldots \vdash a = aa, a \neq aa, \ldots$$

which are also trivially true. Of the remaining five sequents, one is resolved automatically by PVS:

$$(s.bu = b.ref) \land (s.f \neq b.f)(((s.bu = b.ref) \land (s.f = b.f)) \Rightarrow (s.me = b.me))$$
$$\land\ (((s.bu = b.ref) \land (s.f \neq b.f)) \Rightarrow (s.me = b.prev))$$
$$\vdash$$
$$((((s.bu) = b.ref) \land ((b.f) = (b.f))) \Rightarrow ((b.me) = (b.me)))$$
$$\land\ ((((s.bu) = b.ref) \land ((b.f) \neq (b.f))) \Rightarrow ((b.me) = (b.prev))).$$

The remaining four leaf nodes, the only ones that need user attention, are resolved using the *iff* strategy of PVS, which converts boolean equality to equivalence, and then applying the *ground* strategy of PVS.

The proof takes about ten minutes to complete on a lightly loaded SparcStation. Converting the specification and generating a strategy for it takes about twenty seconds.

5. Related work

We are not aware of any published work using the PVS prover for reasoning about action systems. Långbacka and von Wright [16, 12] have formalized TLA, the Temporal Logic of Actions [11] using the HOL [6] theorem prover, which is based on similar typed higher order logic. Our formalization is substantially different from theirs, mostly due to the fact that we need to handle objects, which are not part of TLA, and to different treatment of state. They also adopt a safe approach and build the formalization as a conservative extension of the base logic, which ensures that their formalization is consistent.

Engberg [4] has implemented a mechanical verification system for TLA using the LP [5] verification system. In this system, the converter tool processes the specification to a much lower level, so the state is not visible to the user at all. For example, the relation $i' = i + 1$ is expressed in our formalization as $i(primed) = i(unprimed) + 1$, but in the LP representation generated by TLP, i and i' are simply separate constants.

6. Conclusions and future work

From the small case studies with *pdp*, the idea of deriving a custom strategy to drive the PVS prover seems to work quite well. Work is underway for tackling more complex examples. The most time consuming part of deriving a proof is actually performing the proof with PVS. The generated strategies contain large amounts of formulas to be parsed by PVS, and we suspect that a fair amount of the running time is spent in parsing these. A more efficiently parsed notation for formulas might help in this respect.

We have set out to prove invariant properties, as invariant proofs follow an easily mechanizable pattern. Our intention is to explore the possibility of providing similar support for liveness properties by identifying classes of properties for which such support is feasible.

References

[1] **Back R.J.R. and Kurki-Suonio R.,** Distributed cooperation with action systems, *ACM Transactions on Programming Languages and Systems*, **10** (4) (1988), 513-554.

[2] **Back R.J.R. and Kurki-Suonio R.,** Serializability in distributed systems with handshaking, *Automata, Languages and Programming*, eds. T.Lepisto and A.Salomaa, *Lecture Notes in Computer Science,* Springer, 1988, 52-66.

[3] **Back R.J.R. and Kurki-Suonio R.,** Decentralization of process nets with a centralized control, *Distributed Computing*, (3) (1989), 73-87.

[4] **Engberg U., Gronning P. and Lamport L.,** Mechanical verification of concurrent systems with TLA, *Computer Aided Verification – Fourth International Workshop CAV'92. Montreal, Canada. June 29 - July 1*, eds. G.v.Bochmann and D.K.Probat, *Lecture Notes in Computer Science* **663**, Springer, 1992.

[5] **Garland S.J. and Guttag J.V.,** An overview of LP, the Larch prover, *Proceedings of the Third International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science*, Springer, 1989.

[6] **Gordon M.J.C.,** HOL: A proof generating system for higher-order logic, *VLSI Specification, Verification and Synthesis*, eds. G.Birtwistle and P.A.Subrahmanyam, Kluwer, 1988, 73-128.

[7] **Järvinen H.-M.**, *The Design of a Specification Language for Reactive Systems*, PhD thesis, Tampere University of Technology, 1992.

[8] **Kellomäki P.**, *Analysis of a stabilizing protocol*, Licentiate of Technology thesis, Tampere University of Technology, 1994. http://www.cs.tut.fi/~pk /papers.html.

[9] **Kellomäki P.**, Self stabilization of a protocol, *Proceedings of the Nordic Seminar on Dependable Computing Systems, Lyngby, Denmark, 24-26. August 1994.* http://www.cs.tut.fi/~pk/papers.html.

[10] **Kurki-Suonio R., Järvinen H.-M., Sakkinen M. and Systä K.,** Object-oriented specification of reactive systems, *Proceedings of the 12th International Conference on Software Engineering*, IEEE Computer Society Press, 1990, 63-71.

[11] **Lamport L.**, The temporal logic of actions, *ACM Trans. Prog. Lang. Syst.*, **16** (3) (1994), 872-923.

[12] **Långbacka T.**, A HOL formalization of the temporal logic of actions, *Lecture Notes in Computer Science* **859**, Springer, 1994.

[13] **Owre S., Rushby J.M. and Shankar N.,** PVS: A prototype verification system, *11th International Conference on Automated Deduction*, ed. D.Kapur, *Lecture Notes in Artificial Intelligence* **607**, Springer, 1992, 748-752.

[14] **Sintonen L.**, Event driven bus architecture for bounded area networks, *Proceedings of the 16th Annual Conference of IEEE Industrial Electronics Society, Pacific Grove, California, November 27-30, 1990*, 539-541.

[15] **Systä K.**, A graphical tool for specification of reactive systems, *Proceedings of the Euromicro Workshop on Real-Time Systems, Paris, France, 1991*, 12-19.

[16] **von Wright J. and Långbacka T.,** Using a theorem prover for reasoning about concurrent algorithms, *Computer Aided Verification – Fourth International Workshop CAV'92, Montreal, Canada, June 29 - July 1, 1992*, eds. G.v.Bochmann and D.K.Probst, *Lecture Notes in Computer Science* **663**, Springer, 1992.

A The specification as given to *pdp*

The specification below corresponds to *spec'* in Figure 1, i.e. it is intended to be generated automatically. The description of action *receive* is omitted for brevity.

```
(system proto
  (class station
    (parameters)
    (variable id integer 0)    (variable me integer 0)
    (variable f boolean false)  (variable n integer 0)
    (variable bu objid null))
  (class bus
    (parameters)
    (variable id integer 0)    (variable me integer 0)
    (variable prev integer 0)  (variable f boolean false))
  (global-assertion
   inv
   (and/ (formals (ss station)
                  (bb bus))
    true
    (and (=> (and (= (variable-ref ss bu) (object-ref bb))
                  (= (variable-ref ss f) (variable-ref bb f)))
          (= (variable-ref ss me) (variable-ref bb me)))
       (=>  (and (= (variable-ref ss bu) (object-ref bb))
                  (/= (variable-ref ss f) (variable-ref bb f)))
           (= (variable-ref ss me) (variable-ref bb prev))))))
  action receive omitted
  (action send
    (parameters (m integer))
    (participants (s station) (b bus))
    (guard (and (= (variable-ref s bu) (object-ref b))
               (and (and/ (formals (s station))
                          true
                          (=> (= (variable-ref s bu)
                                 (object-ref b))
                             (= (variable-ref s f)
                                (variable-ref b f))))
                  (= (variable-ref s id)
                     (mod (+ (variable-ref b id) 1)
                          (variable-ref s n))))))
    (body
```

```
(assignment (b prev) (variable-ref b me))
(assignment (b f) (not (variable-ref b f)))
(assignment (s f) (not (variable-ref s f)))
(assignment (s me) (parameter-ref m))
(assignment (b me) (parameter-ref m))
(assignment (b id) (variable-ref s id))))))
```

B The generated PVS theory

This is the theory given to the PVS prover. It has been formatted for readability, and some parts have been omitted for the sake of brevity.

```
proto:  THEORY BEGIN
discolib :   LIBRARY = "/home/kaarne-b/pk/vaikkari/pdp/pvs"
importing discolib@mod
importing discolib@disco
station:  TYPE = [# id:  [state ->int],me:  [state ->int],
  f:  [state ->bool], n:  [state ->int],bu:  [state ->objid],
      ref:  objid#]
station_unique_ax :
  AXIOM forall (obj1,obj2:station):ref(obj1)=ref(obj2)
    implies obj1=obj2
bus:  TYPE = [# id:  [state ->int],me:  [state ->int],prev:
    [state ->int], f:  [state ->bool],ref :   objid #]
bus_unique_ax :
  AXIOM forall (obj1,obj2:bus):ref(obj1)=ref(obj2) implies
    obj1=obj2
```

action receive omitted

```
proving_send :   bool
send_guard(ss:station,s:station,b:bus,m:int,other:state) :
   bool = (bu(s)(other) = ref(b))
   and ((bu(ss)(other) = ref(b))
      implies(f(ss)(other) = f(b)(other)))
   and(id(s)(other) =(mod(id(b)(other) + 1,n(s)(other))))
```

```
send(unprimed, primed :  state) :  bool =
   (exists (s:station,b:bus,m:int) :
   (forall(ss:station):send_guard(ss,s,b,m,unprimed))
   and prev(b)(primed)= (me(b)(unprimed))
   and f(b)(primed)= (not (f(b)(unprimed)))
   and f(s)(primed)= (not (f(s)(unprimed)))
   and me(s)(primed)= (m)
   and me(b)(primed)= (m)
   and id(b)(primed)= (id(s)(unprimed))
   and ((forall(other:station):
      ((other)/=(s))implies(me(other)(primed)=
      (me(other)(unprimed)))))
   and ((forall(other:station):
      ((other)/=(s))implies(f(other)(primed)=
      (f(other)(unprimed)))))
   and ((forall(other:bus):
      ((other)/=(b))implies(id(other)(primed)=
      (id(other)(unprimed)))))
   and ((forall(other:bus):
      ((other)/=(b))implies(me(other)(primed)=
      (me(other)(unprimed)))))
   and ((forall(other:bus):
      ((other)/=(b))
         implies (prev(other)(primed)= (prev(other)(unprimed)))))
   and((forall(other:bus):
      ((other)/=(b)) implies (f(other)(primed)=
      (f(other)(unprimed))))))
   and (proving_send)
   and ((forall(other:station):  id(other)(primed)=
        (id(other)(unprimed))))
   and ((forall(other:station):  n(other)(primed)=
      (n(other)(unprimed))))
   and((forall(other:station):  bu(other)(primed)=
      (bu(other)(unprimed))))
END proto

proto_assertions :THEORY BEGIN
```

```
IMPORTING proto
inv_body(bb:bus,ss:station,other:state) :  bool =
  (((bu(ss)(other) = ref(bb)) and(f(ss)(other) = f(bb)(other)))
       implies(me(ss)(other) = me(bb)(other)))
  and (((bu(ss)(other) = ref(bb))
  and (f(ss)(other) /= f(bb)(other)))
       implies(me(ss)(other) = prev(bb)(other)))
inv(other:state) :  bool =
  (forall(ss:station):(forall(bb:bus):inv_body(bb,ss,other)))

INIT(other:state) :  bool = inv(other)

ACTIONS(s,sp:state) :  bool = receive(s,sp) or send(s,sp)

inv_is_invariant:  THEOREM invariant(inv,INIT,ACTIONS)

END proto_assertions
```

C An excerpt from the generated strategy

```
(defstep proto-inv ()
  (then* (lemma "invariant_rule" ("P" "inv"))
  (split) (skosimp*) (delete-formula "[]inv")
  (branch (split)
    ((then* (expand "INIT") (ground))
    (then* (flatten)
      (branch (split) ((else (proto-receive-inv$)
        (proto-send-inv$)))))))))
"Top level strategy." "Top level strategy.")

(defstep proto-send-inv ()
  (try
   (expand "send")
   (then*
    (skolemize-action "s") (skolemize-action "b")
        (skolemize-action "m")
    (flatten) (expand "inv")
```

```
(skolemize
  "ss" "(forall(ss:station):(forall(bb:bus):
    inv_body(bb,ss,sp!1)))")
(skolemize "bb" "(forall(bb:bus):inv_body(bb,ss,sp!1))")
(spread
 (case-replace "bb = b")
 ((then*
   (spread
    (case-replace "ss = s")
```

P. Kellomäki
Department of Computer Science
Tampere University of Technology
P.O.B. 553
FIN-331011 Tampere, Finland
pk@cs.tut.fi