Verifying Semantic Relations in SPIN*

Hakan Erdogmus
Institute for Information Technology/Software Engineering Group
National Research Council
Building M-50, Montreal Road
Ottawa, Ontario, Canada K1A 0R6
erdogmus@iit.nrc.ca

1 Introduction

SPINe is an experimental verification system based on PROMELA/SPIN version 1.5.7. SPIN is a general verification tool for proving correctness properties of concurrent/distributed systems specified in the CSP-like modeling language PROMELA [8, 9]. This extended abstract describing the SPINe system and its foundations assumes familiarity with PROMELA/SPIN.

The SPINe system extends SPIN with 'limited' semantic relation checking capability implemented in terms of a new option, `-e`, whose usage is given below:

`spine -e`*Rel fileL fileR*

Here *Rel* indicates the particular semantic relation (usually a preorder or an equivalence) to be verified[1] and *fileL* and *fileR* are two files containing the PROMELA models to be compared (called *lhs* and *rhs* models, respectively). The term 'limited' is used in the sense that currently only a particular class of semantic relations are supported. This class is chosen for the following reasons: (1) there exists a general, easy-to-implement, on-the-fly checking algorithm for this class; (2) the class includes a number of well-known semantic relations, such as trace inclusion and testing equivalence, as well as others which underlie different extended trace models.

The existing options of SPIN version 1.5.7 work as before with SPINe. The command '`spine -e`' generates a customized relation checker—a collection of **C** files—which is subsequently compiled and run to return a verdict. The flow diagram of the SPINe system is shown in Fig. 1.

2 Extension to PROMELA's Syntax

PROMELA does not provide an explicit mechanism to distinguish between external (observable) and internal (invisible) behavior. Without such a mechanism, the ability to verify semantic relations is not

*Supported jointly by Bell Northern Research Laboratories, Montreal, and INRS-Télécommunications, Montreal.
[1] e.g., trace inclusion, trace equivalence, testing equivalence

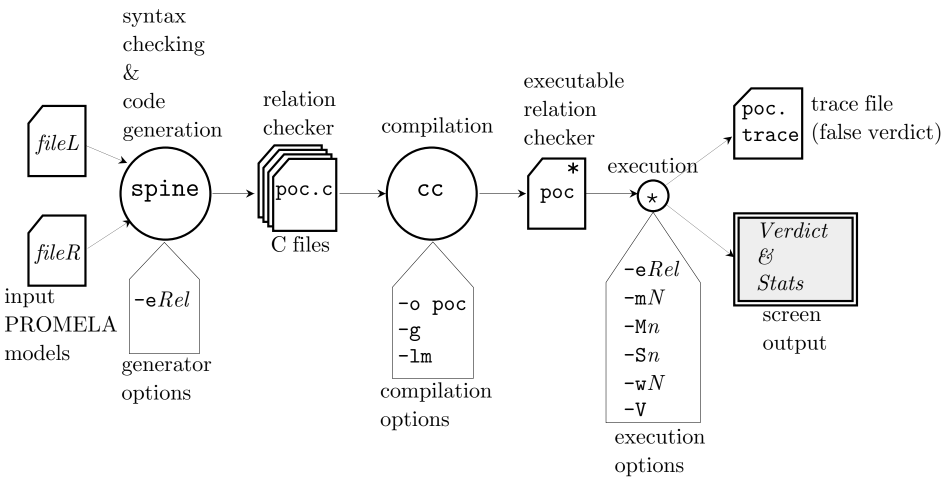


Figure 1: Flow diagram of the SPINe verification system.

very useful. The computations that can most obviously be subjected to external observation are channel operations (send/receive).[2] However, it is not necessary for all possible operations on all the declared channels to be externally observable: communications of certain types on selected channels may be considered invisible, or internal, while others may be externally observable. We modified the syntax of PROMELA's `chan` declaration to allow the association of an individual channel with an external name that is unique throughout the PROMELA model which contains it. The new syntax is as follows:

`chan` *intname* `(extern` *extname btype*`) = [`*nslots*`] of { ... }`

where *extname* is the external name of the channel whose internal name is *intname*. This syntax makes it possible to treat send (`!`) and/or receive (`?`) operations on external channels of a PROMELA model as externally observable communications during a SPINe verification. More precisely, these external communications correspond to the external actions of the underlying labeled transition system. In an external channel declaration, *btype* specifies which types of operations on that channel are considered as external communications: `!` is used to declare only the send operations to be external and `?` is used to declare only the receive operations to be external. If *btype* is omitted, both sends and receives are treated as external communications. For external synchronous channels, either `!` is specified or *btype* is omitted altogether.

It is important that each external name be unique in a given model. When a semantic relation is verified between two PROMELA models, a communication on an external channel in the first model is comparable to a communication on an external channel in the second model if and only if the external names of the two channels match.

Note that in the current implementation, the above syntax is applicable only to individual channels, and cannot be used with arrays of channels.

[2]Although communication through shared variables is available in PROMELA, the basic mechanism for inter-process communication is message passing through synchronous or asynchronous channels. We considered only this latter mode of communication for specifying external behavior.

3 Example: Synchronous FIFO Buffers

As an example, consider a system consisting of a synchronous FIFO buffer with a maximum capacity of two slots (each capable of holding one byte of information) and an environment process depositing and retrieving one byte values in a somewhat random manner. A PROMELA model of this a system is shown in Fig. 2 where the environment process is defined in Fig. 4.

In a distributed implementation of the same system, the process `FIFO2` of Fig. 2 is replaced by the interconnection of two instances of a synchronous FIFO buffer having a capacity of a single slot (the process type `FIFO1`). This latter system is depicted in Fig. 3.

Note how the external behavior of the PROMELA models `specL` and `specR` are defined in terms of external channel declarations. The external name `IN` establishes a binding between the channel `inL` of `specL` and the channel `outL` of `specR`, allowing the send/receive operations on these two channels to be compared for a possible match during a SPINe verification. Similarly for the external name `OutL`. Note that in the distributed implementation `specR`, the channel `shift` which interconnects the two instances of the process type `FIFO1` is declared as an *internal* channel (because it does not have an external name). We can easily prove the two models to be trace-equivalent[3] by the following C-shell script whose result is shown in Fig. 5:

```
spine -e2 specL specR
cc -o poc poc.c -lm
poc
```

In LOTOS terms, the above verification is equivalent to deciding trace equivalence between the LOTOS behaviors

```
Environment(IN,OUT) |[IN, OUT]| FIFO2(IN, OUT)
```

and

```
hide shift in (Environment(IN, OUT) |[IN,OUT]|
  (FIFO1(IN, shift) |[shift]| FIFO1(shift,OUT)))
```

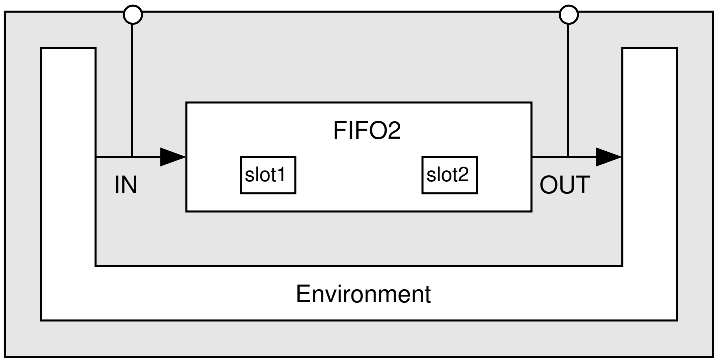
As an aside, we comment on the role of the process `Environment` in the verification. In SPIN, one can only reason about *closed* systems (complete, self-contained PROMELA models). If a semantic relation *Rel* is to be verified between two *open* systems S_1 and S_2, the systems must first be composed with a model of their respective intended environments, say E_1 and E_2, resulting in two closed systems. Then the verification takes the form $S_1||E_1$ *Rel* $S_2||E_2$ where $||$ denotes parallel composition. In PROMELA such composition is modeled by the `run` statement. Note that in the above example, a common environment is used.

4 Inductive Relations

SPINe supports a particular class of semantic relations, called *inductive relations*, first identified in [4]. Before discussing these relations, two relevant models—Extended Labeled Transition Systems and Weak Process Systems—need to be introduced.

[3]i.e., satisfy the same *safety* properties

(a)

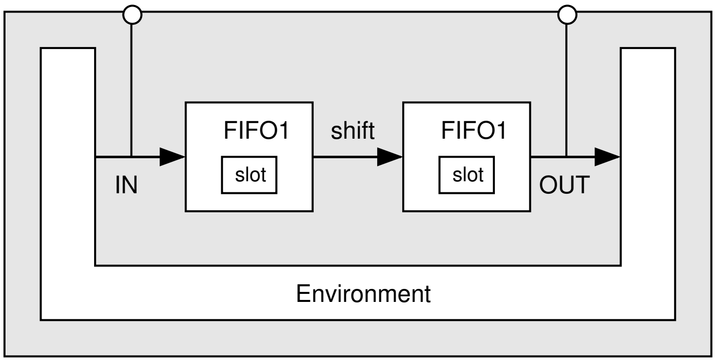


(b)

```
chan inL  (extern IN)  = [0] of {byte};
chan outL (extern OUT) = [0] of {byte};

#include "Environment"

proctype FIFO2(chan In, Out)
{
 byte slot1, slot2;
 do
   :: In?slot1 ->
      do
        :: In?slot2  -> Out!slot1 -> slot1 = slot2
        :: Out!slot1 -> break
      od
 od
}

init
{
 atomic{run FIFO2(inL, outL);
        run Environment(inL, outL)}
}
```

Figure 2: (a) Block diagram of the centralized FIFO buffer system consisting of an environment and a two-slot synchronous FIFO buffer. (b) The corresponding PROMELA model `specL`.

(a)



(b)

```
#include "Environment"

chan inR  (extern IN)  = [0] of {byte};
chan shift = [0] of {byte};
chan outR (extern OUT) = [0] of {byte};

proctype FIFO1(chan In, Out)
{
 byte slot;
 do
   :: In?slot -> Out!slot
 od
}

init
{
 atomic{run FIFO1(inR, shift);
        run FIFO1(shift, outR);
        run Environment(inR, outR)}
}
```

Figure 3: (a) Block diagram of the distributed FIFO buffer system consisting of two an environment an two synchronous single-slot FIFO buffers. (b) The corresponding PROMELA model `specR`.

```
proctype Environment(chan send, recv)
{
 byte any;
 do
   :: send!1
   :: send!2
   :: send!3
   :: send!4
   :: recv?any
 od
}
```

Figure 4: The include file `Environment` where the process type representing the common environment is defined.

```
(
   SPIN Preorder Checker Version 1.2 (based on SPIN Version 1.5.7)
   June 1994
   Hakan Erdogmus
   INRS-Telecommunications
   16 Pl. du Commerce, Verdun, Quebec, H3E 1C8 Canada
)
Verifying Trace Equivalence ...
Verdict: TRUE: specL [TRACE-EQUIVALENT] specR

Statistics:
         185 L state(s) stored permanently in the hash table
                      of a total of 851 L state(s) generated
              400 L state(s) matched
              185 L extended state(s) stored
              400 L extended state(s) matched
             1835 L link(s) executed with 1 atomic links
                0 L state(s) with internal cycles found
                0 L deadlock state(s) found
                0 L atomic sequence blockage(s) found
                0 L endstate(s) reached
         105 R state(s) stored permanently in the hash table
                      of a total of 1356 R state(s) generated
              480 R state(s) matched
              105 R extended state(s) stored
              480 R extended state(s) matched
             2420 R link(s) executed with 2 atomic links
                0 R state(s) with internal cycles found
                0 R deadlock state(s) found
                0 R atomic sequence blockage(s) found
                0 R endstate(s) reached
         185 composite state(s) stored
              400 truncated search(es) due to matched composite state
                0 truncated search(es) due to high-level (LR) stack overflow
                   5000 was the max allowed depth of LR stack
                     47 was the max reached depth of LR stack
                0 truncated search(es) due to low-level stack overflow
                   1000 was the max allowed depth of stack
                      3 was the max reached depth of stack
          36 was the max vector size reached
        1907 times an H_el record (state) has been recycled
                 a total of 300 such record(s) allocated
       16350 time(s) a TransDescr record has been recycled
                 a total of 138 such record(s) allocated
        2792 time(s) a StateSet record has been recycled
                 a total of 295 such record(s) allocated
         334 state comparison(s) made
       65536 was the size of state hash table
                  99.68% empty
                   1.39 sd 0.87 state(s) per non-empty entry
       65536 was the size of extended state hash table
                  99.68% empty
                   1.39 sd 0.87 extended state(s) per non-empty entry
                   1.00 sd 0.00 state(s) per extended state
       65536 was the size of composite state hash table
                  99.72% empty
                   1.00 sd 0.00 composite state(s) per non-empty entry
```

Figure 5: Result of SPINe verification for the synchronous FIFO buffer example.

4.1 Labeled Transition Systems

A PROMELA model defines a network of communicating processes, treated by the SPIN validator as a (possibly huge) NFSA[4] with transitions labeled by the communications and other executable PROMELA statements. With the added ability to distinguish between internal and external communications, the adoption of *Extended Labeled Transition Systems* as the underlying formal model is suitable for the purposes of semantic relation checking.

An ELTS is a quadruple $\langle \Sigma, A, \{-a\!\!\rightarrow \mid a \in A\}, -\cdot\!\!\rightarrow \rangle$, where Σ is a set of *states*, A is a set of *external actions*, the $-a\!\!\rightarrow \subseteq \Sigma \times \Sigma$ are called the *external transition relations*, and $-\cdot\!\!\rightarrow \subseteq \Sigma \times \Sigma$ is called the *internal transition relation*. The predicates $=\cdot\!\!\Rightarrow$ and $=a\!\!\Rightarrow$ are defined as follows: $=\cdot\!\!\Rightarrow^0 \stackrel{\mathrm{def}}{=} \{\langle \sigma, \sigma \rangle \mid \sigma \in \Sigma\}$, $=\cdot\!\!\Rightarrow \stackrel{\mathrm{def}}{=} \bigcup\{=\cdot\!\!\Rightarrow^n \mid n \in Nat\}$, and $=a\!\!\Rightarrow \stackrel{\mathrm{def}}{=} =\cdot\!\!\Rightarrow\!-a\!\!\rightarrow\!=\cdot\!\!\Rightarrow$. The last predicate may be extended to arbitrary sequences of actions in A^* as follows: $\sigma{=}as{\Rightarrow}\sigma' \stackrel{\mathrm{def}}{=} =a\!\!\Rightarrow\!=s\!\!\Rightarrow$, with the convention that $\sigma{=}\varepsilon{\Rightarrow}\sigma$ for all $\sigma \in \Sigma$, where ε denotes the empty sequence. For a state $\sigma \in \Sigma$, we use $traces(\sigma)$ denote the set of all finite execution traces of σ:

$$traces(\sigma) \stackrel{\mathrm{def}}{=} \{s \in A^* \mid (\exists \sigma' \in \Sigma)[\sigma{=}s{\Rightarrow}\sigma']\}.$$

The *must* set of σ is the set

$$must(\sigma) \stackrel{\mathrm{def}}{=} \{a \in A \mid (\forall \sigma' \in \Sigma)[\sigma{=}\cdot{\Rightarrow}\sigma' \text{ implies } (\exists \sigma'' \in \Sigma)[\sigma'{=}a{\Rightarrow}\sigma'']]\}$$

and the *must* set of σ after s is the set

$$must(\sigma, s) \stackrel{\mathrm{def}}{=} \{a \in A \mid (\forall \sigma' \in \Sigma)[\sigma{=}s{\Rightarrow}\sigma' \text{ implies } a \in must(\sigma')]\}.$$

If σ has an infinite internal computation

$$\sigma -\cdot\!\!\rightarrow \sigma_1 -\cdot\!\!\rightarrow \sigma_2 \cdots -\cdot\!\!\rightarrow \sigma_k -\cdot\!\!\rightarrow \cdots$$

then σ is said to be a *divergent* state, written $\sigma\!\uparrow$. Otherwise, σ is called a *convergent* state, written $\sigma\!\downarrow$. Finally, if for some $\rho \in \Sigma$ and for some prefix r of s, $\sigma{=}r{\Rightarrow}\rho$ such that $\rho\!\uparrow$, then we write $\sigma\!\uparrow\! s$; otherwise we write $\sigma\!\downarrow\! s$. Note that $\sigma\!\uparrow\!\varepsilon$ iff $\sigma\!\uparrow$.

4.2 Weak Process Systems

Rather than on the structure of an ELTS, the notion of inductive relation is more easily defined on a general extended trace model called a *Weak Process System*. A WPS is a structure $\langle \Pi, \Lambda, A, \mathcal{L}, \mathcal{A}, \{\cdot(a) \mid a \in A\}\rangle$, where Π is a set of *(weak) processes*, Λ is a set of *local behaviors*, A is a set of *(external) actions*, $\mathcal{L}\colon \Pi \longmapsto \Lambda$ is called the *labeling function*, $\mathcal{A}\colon \Lambda \longmapsto A$ is called the *local action set function*, and finally the $\cdot(a)\colon \Pi \longmapsto \Pi$ are called the *transition functions*. Because its transitions are defined as functions, a WPS has a deterministic branching structure.

Given a WPS, let REL_Λ be a binary relation on Λ. We call REL_Λ a *local relation*.

Theorem 1 *For every local relation* REL_Λ*, there exists a unique maximal binary relation* REL *on* Π *which satisfies for all* $P, Q \in \Pi$*,* P REL Q *iff* $\mathcal{L}(P)$ REL_Λ $\mathcal{L}(Q)$ *and* $P(a)$ REL $Q(a)$*, for every* $a \in \mathcal{A}(\mathcal{L}(P)) \cap \mathcal{A}(\mathcal{L}(Q))$*. The relation* REL *is called an* inductive relation[5].

[4] Nondeterministic Finite State Automaton

[5] The term inductive is used because this class of relations was originally formulated in an inductive manner.

4.3 Characterization of Semantic Relations using Weak Process Systems

Theorem 1 states that every inductive relation is uniquely characterized by an underlying relation on local behaviors. As examples, we consider three well-known semantic relations, trace equivalence ($\equiv_{\text{trace}}$), trace inclusion ($\leq_{\text{trace}}$), and testing equivalence ($\equiv_{\text{test}}$)—each of which can be formulated as an inductive relation on the structure of a WPS using a transformation *Det* which maps a given ELTS to a corresponding WPS. The purpose of the transformation *Det* is to abstract from internal transitions and extract the relevant 'local' information. Once *Det* is defined, we can identify the local relations underlying $\equiv_{\text{trace}}$, $\leq_{\text{trace}}$, and $\equiv_{\text{test}}$.

Let $\sigma, \rho \in \Sigma$. *Trace equivalence* simply equates two states of an ELTS if they have the same set of finite traces:

Definition 1 *$\sigma \equiv_{\text{trace}} \rho$ if $traces(\sigma) = traces(\rho)$.*

Trace inclusion, defined below, is a preorder of this relation:

Definition 2 *$\sigma \leq_{\text{trace}} \rho$ if $traces(\sigma) \subseteq traces(\rho)$.*

These two relations capture safety properties only. The third, more interesting relation we consider is *testing equivalence*, which takes into account some liveness properties—divergence, deadlock, and internal nondeterminism in particular:

Definition 3 *$\sigma \equiv_{\text{test}} \rho$ if for all $s \in A^*$, we have:*

i) $\sigma\uparrow s$ iff $\rho\uparrow s$,

ii) $traces(\rho) = traces(\sigma)$, and

iii) $\sigma\downarrow s$ implies $must(\sigma, s) = must(\rho, s)$.

These relations and their variants have been discussed under different names and with different characterizations in several references; see for examples [3, 6, 2, 5]. The version of testing equivalence given above is derived from [2], but is more suitable for implementation purposes. Note that according to testing equivalence, divergence is persistent; i.e., all descendants of a divergent state are also treated as being divergent.

The transformation *Det* is similar to NFSA determinization, but also incorporates the *must* set and divergence information to the local behaviors. Let $\mathbf{T} = \langle \Sigma, A, \{-a\rightarrow \mid a \in A\}, -\cdot\rightarrow \rangle$ be an ELTS. Then $Det(\mathbf{T}) \stackrel{\text{def}}{=} \langle \Pi, \Lambda, A, \mathcal{L}, \mathcal{A}, \{\cdot(a) \mid a \in A\} \rangle$, where

- $\Pi \stackrel{\text{def}}{=} 2^{\Sigma}$;
- $\Lambda \stackrel{\text{def}}{=} 2^A \times 2^A \times \{\uparrow, \downarrow\}$;
- for $P \in \Pi$, $\mathcal{L}(P)$ is defined as the triple $\langle S, M, d \rangle$ such that
  - $S \stackrel{\text{def}}{=} \{a \in A \mid (\exists \sigma \in P)(\exists \sigma' \in \Sigma)[\sigma{=}a{\Rightarrow}\sigma']\}$,
  - $M \stackrel{\text{def}}{=} \{a \in A \mid (\forall \sigma \in P)[a \in must(\sigma)]\}$, and

- $d \stackrel{\text{def}}{=} \uparrow$ if $(\exists\sigma \in P)(\exists\rho \in \Sigma)(\exists s \in A^*)[\rho\uparrow \wedge \rho{=}s{\Rightarrow}\sigma]$; $\downarrow$ otherwise;

- for $S, M \in 2^A$ and $d \in \{\uparrow, \downarrow\}$, $\mathcal{A}(S, M, d) \stackrel{\text{def}}{=} S$;

- for $P \in \Pi$ and $a \in A$, $P(a) \stackrel{\text{def}}{=} \{\rho \in \Sigma \mid (\exists\sigma \in P)[\sigma{=}a{\Rightarrow}\rho]\}$.

Now we can identify the three local relations—TRCEQ_Λ, TRCINC_Λ, and TSTEQ_Λ—underlying $\equiv_{\text{trace}}$, $\leq_{\text{trace}}$, and $\equiv_{\text{test}}$, respectively. The relation TRCEQ_Λ simply coincides with equality between the S components, whereas TRCINC_Λ reduces to subset inclusion. TSTEQ_Λ is slightly more complex:

Definition 4 *Let $S, M \in 2^A$ and $d \in \{\uparrow, \downarrow\}$.*

i) $\langle S, M, d\rangle\ \mathrm{TRCEQ}_\Lambda\ \langle S', M', d'\rangle$ *if* $S = S'$.

ii) $\langle S, M, d\rangle\ \mathrm{TRCINC}_\Lambda\ \langle S', M', d'\rangle$ *if* $S \subseteq S'$.

iii) $\langle S, M, d\rangle\ \mathrm{TSTEQ}_\Lambda\ \langle S', M', d'\rangle$ *if* $S = S' \wedge d = d' \wedge (d = \downarrow$ *implies* $M = M')$.

Theorem 2 *Let $\sigma, \rho \in \Sigma$ in $\mathbf{T}$. We have*

i) $\sigma \equiv_{\text{trace}} \rho$ *iff* $\{\sigma\}$ TRCEQ $\{\rho\}$ *in* $Det(\mathbf{T})$.

ii) $\sigma \leq_{\text{trace}} \rho$ *iff* $\{\sigma\}$ TRCINC $\{\rho\}$ *in* $Det(\mathbf{T})$.

iii) $\sigma \equiv_{\text{test}} \rho$ *iff* $\{\sigma\}$ TSTEQ $\{\rho\}$ *in* $Det(\mathbf{T})$.

5 Relation Checking in SPINe

The algorithm used by SPINe to check inductive relations on PROMELA models is based on Theorems 1 and 2. It is illustrated in Fig. 6. This is a recursive fixpoint algorithm which computes a representation of the composite state space of the two input PROMELA models. It performs a depth-first search of the composite state space. The weak process representations (the transformation *Det*) of the ELTSs underlying the two input PROMELA models are computed on-the-fly.

For a given set of states, *Add_internal_states* computes all states which are reachable through a sequence of internal communications (transitions) from the given states employing a depth-first strategy. Note that each PROMELA statement which is not a send/receive operation on an external channel is treated as an internal transition. The output is a data structure called an extended state, a linked list of states with the proper 'local' information attached to it. An extended state is the implementation equivalent of a weak process. Divergent and deadlocked states are detected on-the-fly during this internal state search phase and labeled accordingly. Although not yet implemented, the *must* sets should also be computed on-the-fly within *Add_internal_states*. To reduce the overall storage requirements, those states in which no external communications are enabled are considered transient and deleted before *Add_internal_states* returns. Such states are not needed once the divergence, deadlock, and *must* information (i.e., the 'local' information) of the extended state has been computed from its constituents and recorded.

A pair of extended states—the first formed of the states of the lhs model and the second of those of the rhs model—constitutes a composite state. The initial composite state is computed from the respective initial states of the lhs and rhs models. Then the recursive routine $Verify\_relation$ is called. If the composite state has been generated before, $Verify\_relation$ returns. Otherwise the composite state is stored and then analyzed.

The function *Local_behavior* computes the data structure representing the local behavior of an extended state. This routine implements the WPS labeling function $\mathcal{L}$. $Check\_local\_rel$ tests whether the specified local relation[6] holds true for the composite state being analyzed. If this test fails, the relation being checked is not satisfied, and a false verdict is returned. If the test succeeds, a new composite state is generated for each external communication (action) enabled in the current composite state. Note that $Check\_local\_relation$ is the only routine that is dependent on the inductive relation being verified.

The routine *Execute_external_action* performs the execution of a selected external action enabled in an extended state. The WPS transition functions $\cdot(a)$ are implemented by applying *Add_internal_states* on the result of *Execute_external_action*. Applied component-wise to a composite state, this procedure produces a potentially new composite state to be analyzed, and thus $Verify\_relation$ is called recursively. This is done once for each enabled external action in a loop. The algorithm terminates with a true verdict when all the possible composite states have been generated with successful local tests.

5.1 Implementation Notes

The relation checking system takes advantage of most of SPIN's existing data structures, with some modifications and additions. Its implementation evolved from Holzmann's implementation of his exhaustive state exploration algorithm supplied with the SPIN validator. The structure of the SPINe state vector is identical to the one used in the SPIN validator with the exception of a prevailing byte to differentiate between lhs and rhs states and a trailing byte to store some extra information (the deadlock status and the divergence status of the state, whether the state is the source of an external transition, whether the state has at least one enabled external transition, etc.) used by the relation checking algorithm.

There are three levels of states implemented by three data structures. Individual states belonging to both lhs and rhs models are stored in a common hash table using SPIN's data structure `H_el`. Transient states are recycled when no longer needed. Extended states are stored in a second hash table using the data structure `StateList`. A composite state is implemented by the data structure `ProdState` and is stored in a third hash table. See Figure 7 for an illustration of the data structures involved.

5.2 Complexity and Performance

It is established in [15] that checking NFSA equivalence is a PSPACE-hard problem. Since for finite state systems (which are the systems of interest), the decision problem for NFSA equivalence can be reduced to a decision problem for trace equivalence (and vice versa), trace equivalence checking is also

[6] one of the predicates TRCEQ_Λ, TRCINC_Λ, TSTEQ_Λ, or another local relation

```
global variables sl, sr, initL, initR, LR_Table, BL, BR, result;
begin
        sL ← initial state of the lhs model; sR ← initial state of the rhs model;
        InitL ← Add_internal_states({sL}); InitR ← Add_internal_states({sR});
        LR_Table ← ∅;
        return Verify_relation(InitL, InitR)
end

Verify_relation(L, R)
local variables A, a;
begin
        if ⟨L, R⟩ ∈ LR_Table then
          return true
        endif;
        LR_Table ← LR_Table ∪ {⟨L, R⟩};
        BL ← Local_behavior(L); BR ← Local_behavior(R);
        if Check_local_rel(BL, BR) then
          A ← External_actions(BL) ∩ External_actions(BR);
          for all a ∈ A do
              L ← Add_internal_sates(Execute_external_action(L, a));
              R ← Add_internal_sates(Execute_external_action(R, a));
              if Verify_relation(L, R) then
                result ← true
              else
                result ← false;
                break
              endif
          endfor;
          return result
        else
          return false
        endif
end
```

Figure 6: Pseudocode for the relation checking algorithm used in SPINe.

PSPACE-hard.[7] Trace equivalence is an inductive relation, therefore deciding inductive relations on the structure of a LTS using a generalized algorithm is at least as hard as deciding trace equivalence, the actual complexity depending on that of the local relations involved. Note however that inductive relation checking is polynomial on the structure of a WPS, but the required transformations from LTS to WPS are exponential since they involve some kind of 'determinization' to abstract from internal transitions.

Practical experience shows that the real complexity arises from the subset construction process performed by the *Add_internal_states* routine. Contrary to what the theoretical results suggest, the total number of extended and composite states generated is far from being exponential in the total

[7] Proposed reduction: choose a symbol, say $\surd$, which is not in the alphabet of the two NFSA being tested for equivalence. For each final state σ of both automata, create an extra 'exit' state σ' and define a transition labeled by $\surd$ from σ to σ'. Use σ as the 'entry' state and σ' as the 'exit' state of the final state. Treat the resulting automata as two ELTSs and verify trace equivalence between them. The language of a NFSA can be deduced from the traces of the corresponding ELTS by deleting those traces not ending with a $\surd$ and then projecting the remaining traces on the original alphabet.

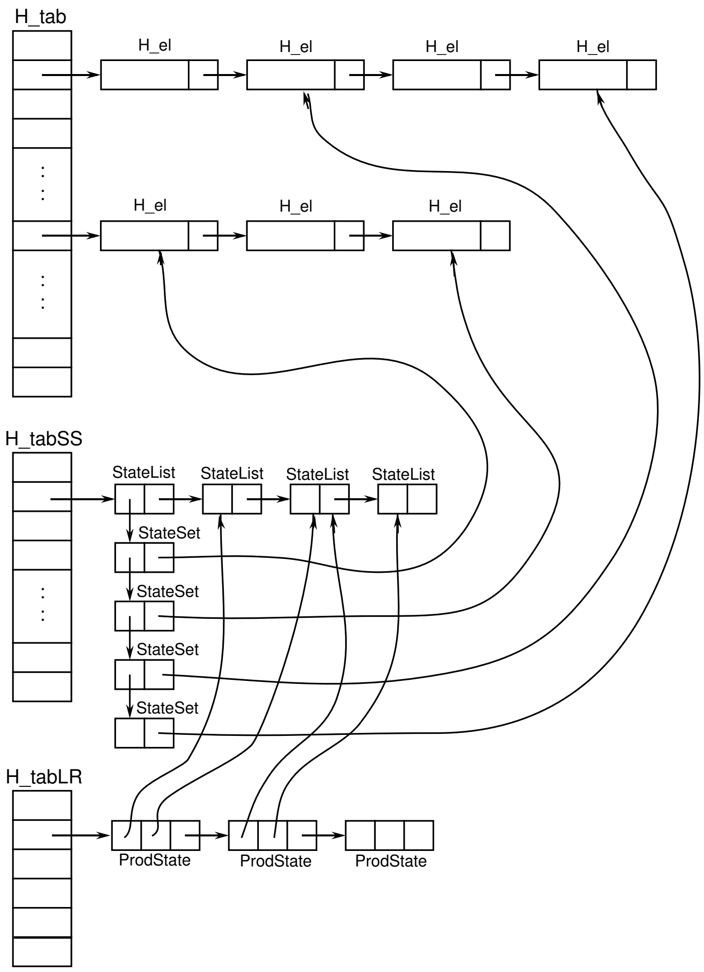


Figure 7: Data structures used in stage storage.

#	*Verdict*	*No. of states*		*Mem*	*Time*
1	True	169/1313	4015/5649	1332	3:39
2	?	69/?	$[3 \times 10^7]/[9 \times 10^9]$	?	?
3	True	69/1960	3045/46715	1416	2:28
4	?	1037/?	$[1 \times 10^7]/?$	?	?
5	True	191/2031	10098/210845	1468	13:01
6	True	20/458	5192/51633	1456	4:14

Table 1: Example SPINe verifications. *Mem* is the total memory required in kB. Under *No. of states*, the first subcolumn relates to the lhs model and the second subcolumn relates to the rhs model. In each subcolumn, the first figure is the total number of states generated by a SPIN exhaustive validation run, whereas the second figure indicates the total number of states (not necessarily distinct) revisited during the SPINe trace equivalence verification. A question mark indicates an inconclusive verdict due to time limitations and the figures between square brackets are estimated upper-bound values. In all of the verifications, the total numbers of extended and composite states generated were low (not shown).

number of individual states generated; indeed in all of the examples tried out, these numbers were several orders of magnitude lower then the number of individual states. Instead, extended states tend to be large, and possibly overlapping. With the state recycling scheme used, the space complexity is alleviated to a large extent at the expense of a sometimes substantial increase in time complexity. Table 1 shows the results of several SPINe verifications carried out on a Sparc2 with 32 MB of memory. In two cases, the results were inconclusive because of unacceptable time requirements. In one case, even an exhaustive SPIN validation was not possible due to state explosion. In all of the examples tried with false verdicts, the results were obtained relatively quickly (not shown).

5.3 Summary and Future Work

SPINe is an experimental verification system for concurrent/distributed systems based on PROMELA/SPIN. It adds to PROMELA/SPIN the capability to verify a particular class of semantic relations.

The SPINe system was developed in 1994, and since early 1995, a new version of PROMELA/SPIN has been in place. The system needs to be upgraded for compatibility with SPIN version 2.x.

The current implementation suffers from some limitations which restrict its practical application in sizable projects. These limitations will have to be addressed in the future. The first limitation concerns the exclusive support of a particular class of relations whose decision algorithms are provably intractable. Although this class (inductive relations) represents an important category[8], other more practical semantic relations with polynomial checking algorithms should be considered as well. Obvious candidates are weak bisimulation equivalence [12], its variants, and preorders of these, reformulated following Park's elegant notion of bisimulation [13]. A polynomial algorithm for checking weak bisimulation equivalence (observation equivalence) has been proposed in [10], but unfortunately, it is not

[8]For example, Hoare's widely-known failures equivalence has been shown by De Nicola [2] to be just another characterization of testing equivalence. Hennessy's *must* testing preorder [6] is one of the preorders of testing equivalence. One can also find several other semantic relations that underlie other extended trace theories: for examples see [16] and [11, Ch. 3].

an on-the-fly algorithm in that it assumes that the entire state spaces have been computed and stored in advance. This algorithm should be examined to see whether it can be adapted appropriately and incorporated to the SPINe system.

It should however be pointed out that such relatively stronger relations as weak bisimulation equivalence—although they are easier to verify—do not abstract from internal behavior in a completely satisfactory manner. As such they may sometimes fall short of making desirable identifications. This limitation should be taken into account. For example, although weak bisimulation equivalence can serve as an excellent 'approximation' to testing equivalence most of the time, it is not universally suitable for all applications.

Other methods of complexity reduction that are worthwhile to examine include:

- Approximative relation checking based on partial state exploration, possibly using SPIN's bitstate hashing technique [7], as suggested in [1].
- The exploitation of partial order state exploration [14] in semantic relation checking.
- Taking advantage of PROMELA2's `hidden` construct to reduce the size and total number of individual states.

Other potential improvements to the SPINe system include:

- The ability to define arrays of channels as external channels.
- The ability to treat shared variable type communications as external operations by allowing ordinary variables to be declared external.

Availability — The SPINe system is availabe through FTP from the site **zoosun.iit.nrc.ca** (subdirectory '/pub/incoming/hakan'). For questions and suggestions, please contact the author.

References

[1] R. Civalero, B. Jonsson, and J. Nilsson. Validating simulations between large nondeterministic specifications. In *Proceedings of Sixth International Conference on Formal Description Techniques*, pages 3–17, 1993.

[2] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[3] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[4] H. Erdogmus. *A Flexible Framework for the Design of Concurrent Nondeterministic Processes*. PhD thesis, INRS-Télécommunications, Verdun, Québec, 1993.

[5] R. Fournier and G. von Bochmann. The equivalence in the DCP model. *Theoretical Computer Science*, 87:97–114, 1991.

[6] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, MA, 1988.

[7] G. J. Holzmann. Algorithms for automated protocol validation. *AT&T Technical Journal*, 69(1), Jan./Feb. 1990.

[8] G. J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.

[9] G. J. Holzmann. Basic spin manual. Technical report, AT&T Bell Laboratories, Murray Hill, N.J., Mar. 1994.

[10] P. C. Kannellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, (86):43–68, 1990.

[11] G. Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS*. Thèse d'agréation de l'enseignement supérieur, Faculté des sciences appliquées, Université de Liège, Belgium, June 1991.

[12] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[13] D. M. R. Park. Concurrency and automata for infinite sequences. In *Proceedings of 5th GI Conference*, number 104 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

[14] D. A. Peled. Combining partial order reductions with on-the-fly model checking. In *Proceedings of 6th Workshop on Computer-Aided Verification*, June 1994.

[15] L. J. Stockmeyer and A. R. Meyer. World problems requiring exponential time. In *Proceedings of 5th ACM Symposium on Theory of Computing*, pages 1–9, Austin, Texas, 1973.

[16] R. J. van Glabbeek. The linear time – branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90 — Theories of Concurrency: Unification and Extension*, number 458 in Lecture Notes in Computer Science, pages 278–297. Springer-Verlag, 1990.