Formal Verification Based on Relation Checking in SPIN: A Case Study

Erdogmus, Hakan; Johnston, R.; Cleary, C.

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC · CNRC

*Formal Verification Based on Relation Checking in SPIN: A Case Study **

Erdogmus, H., Johnston, R., Cleary, C.
January 1996

* published in the Proceedings of the First Workshop on Formal Methods in Software Practice. San Diego, California, USA. January 11, 1996. NRC 39183.

Canada

This report also appears in *Proceedings of FMPS'96, The First Workshop on Formal Methods in Software Practice*, San Diego, CA, January 10–11, 1996.

Additional copies are available free of charge from:

Publication Office
Institute for Information Technology
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R6

Des exemplaires supplémentaires peuvent être obtenus gratuitement à l'addresse suivante:

Bureau des publications
Institut de technologie de linformation
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R6

Formal Verification Based on Relation Checking in SPIN: A Case Study*

Hakan Erdogmus† Robert Johnston‡ Charles Cleary§

Abstract—A case study in formal verification of concurrent/distributed software is presented. The study concerns the modular specification and verification of a remote task protocol. The verification methodology used is based on semantic equivalence checking and is applicable to systems with hierarchical architectures. To support the methodology, we extended the verification tool SPIN with the ability to check a particular class of semantic relations, and the language PROMELA upon which SPIN is based with a simple mechanism to specify external operations. The foundations of semantic equivalence checking are also discussed briefly.

1 Introduction

This paper illustrates a formal verification methodology based on semantic equivalence checking in the context of a case study concerning the modular specification and verification of a remote task protocol. The method is applicable to concurrent/distributed software structured as a hierarchy of modules, layered protocols in particular. The modular architecture of these systems are represented in terms of block diagrams consisting of nested interconnected boxes. The systems of interest are *closed* in that we require their environments (contexts) to be modeled explicitly. The model of the environment is considered to be an integral part of the system.

At the lowest level of a given system architecture are primitive boxes, or *modules*, for which we define state machine behaviors in terms of Holzmann's language PROMELA. Replaceable modules are encapsulated in *functional groups*. Typically, a functional group contains two modules of different abstraction levels: a primitive, monolithic module which specifies the service to be provided, and a more elaborate, compound module which, for example, represents a distributed implementation of that service. In higher level modules, members of a functional group serve as interchangeable components. We define functionality in terms of a semantic equivalence defined on PROMELA behaviors. Thus a system specified by means of functional groups gives rise to proof obligations; it has to be verified that the resulting modules constitute an equivalence class with respect to the given semantic equivalence.

The proof obligations are specified in terms of nested AND/OR graphs which we call *task structures*. A task structure is a collection of *disjunctive* or *conjunctive* tasks organized to form a modular correctness requirements specification. A conjunctive task specifies obligations that must be satisfied individually, whereas, a disjunctive task specifies alternative obligations. The result of a task is a verdict—true, false, or inconclusive—depending on whether the underlying obligations are satisfied, dissatisfied, or infeasible to decide. Large tasks are decomposed into more manageable subtasks along two orthogonal dimensions. Along the *vertical* dimension, the hierarchical architecture of the system being specified is exploited; modules are verified using service specifications (or abstractions) of their submodules. Along the *horizontal* dimension, it is the underlying property being verified which is broken down into simpler properties that are easier to check. The decomposition of tasks into subtasks is a design activity, and as such it depends highly on the skill of the designer.

All primitive tasks are expressed in terms of semantic equivalence checking. The equivalence chosen is trace equivalence; two systems are considered equivalent if they possess the same set of visible execution traces. We attack trace equivalence checking in the context of a more general problem, namely, that of *inductive relation* checking. Inductive relations are a particular class behavioral relations which have local characterizations [10]. The decidability problem for non-trivial relations in this class is provably intractable. Thus a compositional verification methodology is of double importance.

Behaviors of primitive modules are described in the specification language PROMELA. System architectures which are initially expressed in terms of block diagrams are also translated to PROMELA models. This is done

*NRC no. 39183. Supported in part by Bell Northern Research Laboratories, Montréal and INRS-Télécommunications.

†National Research Council Canada, Building M-50, Montreal Road, Ottawa, Ontario, Canada K1A 0R6

‡INRS-Télécommunications, 16 Place du Commerce, Verdun, Québec, Canada H3E 1H6

§Département de génie electrique, Université Laval, Québec, Canada

in a straightforward manner using cpp macros. PROMELA is a CSP-like language on which Holzmann's tool SPIN is based. The latter is a general verification tool for specifying and proving correctness properties of concurrent/distributed systems [17, 18]. We extended SPIN with inductive relation checking to support our methodology.

2 Inductive Relations and Trace Equivalence

Labeled Transition Systems (LTSs) are commonly used as the underlying formal model for verification problems. A LTS 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 (visible) 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 relations $=\cdot\Rightarrow$ and $=a\Rightarrow$ are defined in the usual manner:

$$=\cdot\Rightarrow^0 \stackrel{\text{def}}{=} \{\langle s, s\rangle \mid s \in \Sigma\}$$

$$=\cdot\Rightarrow \stackrel{\text{def}}{=} \bigcup\{=\cdot\Rightarrow^n \mid n \in Nat\}$$

$$=a\Rightarrow \stackrel{\text{def}}{=} =\cdot\Rightarrow -a\rightarrow =\cdot\Rightarrow$$

For $s \in \Sigma$, let $traces(s)$ denote the set of all finite external (visible) execution traces of s. *Trace equivalence*, $\equiv_{\text{trace}}$, simply equates two states of a LTS iff they have the same set of finite external traces. Let $s, r \in \Sigma$. Then $s \equiv_{\text{trace}} r$ iff $traces(s) = traces(r)$.

Rather than on a LTS, the notion of inductive relation is more easily defined on a general extended trace model called a *Weak Process System* (WPS). 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 *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]*.*

Theorem 1 states that every inductive relation is uniquely characterized by an underlying relation on local behaviors. To formulate $\equiv_{\text{trace}}$ as an inductive relation, we specify a transformation *Det* which maps a given LTS to a corresponding WPS by abstracting from internal transitions, and then identify the local relation underlying $\equiv_{\text{trace}}$. In this light, let $\mathbf{T} = \langle \Sigma, A, \{-a\rightarrow \mid a \in A\}, -\cdot\rightarrow\rangle$ be a LTS. The transformation *Det* is quite straightforward; it is similar to NFSA determinization: $Det(\mathbf{T}) \stackrel{\text{def}}{=} \langle \Pi, \Lambda, A, \mathcal{L}, \mathcal{A}, \{\cdot(a) \mid a \in A\}\rangle$, where $\Pi \stackrel{\text{def}}{=} 2^\Sigma$, $\Lambda \stackrel{\text{def}}{=} 2^A$, and for $P \in \Pi$, $\lambda \in \Lambda$, and $a \in A$, $\mathcal{L}(P) \stackrel{\text{def}}{=} \{a \in A \mid s{-a\rightarrow}$ for some $s \in P\}$, $\mathcal{A}(\lambda) \stackrel{\text{def}}{=} \lambda$, and $P(a) \stackrel{\text{def}}{=} \{p \in \Sigma \mid s{=a\Rightarrow}p$ for some $s \in P\}$. Note that *Det* does not fully take advantage of the potentially richer structure of a WPS since for $\equiv_{\text{trace}}$, the notion of local behavior conveniently coincides with that of local action set; i.e., $\mathcal{A}(\mathcal{L}(P)) = \mathcal{L}(P)$. The final step is the identification of the underlying local relation; let's call it TRACE_Λ. The relation TRACE_Λ simply coincides with equality between local behaviors: λ TRACE_Λ λ' iff $\lambda = \lambda'$.

Theorem 2 *Let* $s, r \in \Sigma$ *in* $\mathbf{T}$*. We have* $s \equiv_{\text{trace}} r$ *iff* $\{s\}$ TRACE_Λ $\{r\}$ *in* $Det(\mathbf{T})$*.*

The significance of the above theorems are explained in the next subsection.

3 Relation Checking in SPIN

The flow diagram of the SPINe system is given in Fig. 2.

The algorithm used to decide trace equivalence is based on Theorems 1 and 2. A PROMELA model consists of a network of communicating finite state processes, and as such it defines a global state machine—a finite LTS—with an initial state and with transitions labeled by communications and other executable PROMELA statements.

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

```
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 1: Pseudocode for the relation checking algorithm used in SPINe.

Then deciding trace equivalence between two PROMELA models—let's call them *lhs* and *rhs* models—corresponds to verifying the relation $\equiv_{\text{trace}}$ for their initial states in the combined LTS. The LTS considered is the union of the two LTSs underlying the lhs and rhs models. This is computed on-the-fly by an exhaustive, synchronized exploration of the respective state spaces. While the state spaces are being traversed, the transformation *Det* is computed, again on-the-fly, by constructing subsets of states connected by continuous sequences of internal transitions. Each such subset gives rise to a weak process. A pair weak processes—L and R for the lhs and rhs models respectively—are produced at each iteration, allowing Theorem 1 to be applied: the local relation TRACE_Λ is checked between the local behaviors of L and R, an external action a common to both L and R is executed, after which the procedure recurses. The general algorithm is illustrated in Fig. 1. In the figure, *Add_internal_states* extends a given subset of states (a weak process) with those states reachable through a sequence of internal transitions from at least one state in the subset. Then *Execute_external_action*, applied to the result, constructs a new subset by executing a selected external action. Combined, these two procedures implement the weak process transition functions $L(a)$ and $R(a)$. *Check_local_rel* verifies a given local relation for the local behaviors associated with a pair of weak processes.

There was no mechanism in original PROMELA to distinguish between external (observable) and internal (invisible) transitions. This ability is essential for semantic relation checking to be of any practical use. Typically, details of sequential computations within individual processes would not be relevant as far as the external functionality of the overall system is concerned. Similarly, all inter-process communications may not be interesting to observe either. Therefore, it should be possible to selectively specify the relevant, externally observable communication actions in a given model. We modified the syntax of PROMELA's channel declaration allowing a channel to be annotated with the keyword *external* and a corresponding external name:

`chan` *int_name* `(extern` *ext_name b_type*`) = [`*n_slots*`] of {` *chan_structure* `}`

Here *ext_name* is the external name of the channel variable *int_name*. Each external name must be unique in a given model. This syntax makes it possible to treat send (`!`) and/or receive (`?`) operations on relevant channels of a PROMELA model as externally observable actions during a SPINe verification. More precisely, such operations correspond to external transitions of the underlying LTS, whilst all other executable PROMELA statements are treated as internal transitions. The keyword *b_type* specifies precisely which types of operations are external: `!` is used to declare only the send operations to be external and `?` is used to declare only the receive operations to be

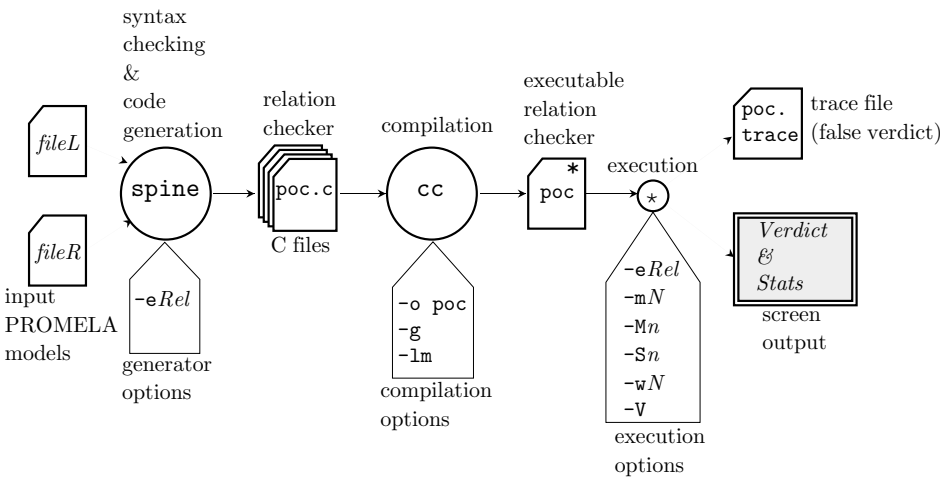

Figure 2: Flow diagram of the SPINe verification system. *Rel* indicates the inductive relation to be verified.

external. If *b_type* is omitted, both sends and receives are treated as external actions. For external synchronous channels ($n\_slots = 0$), either `!` is specified or *b_type* is omitted altogether.

4 Case Study: A Remote Task Protocol

To demonstrate the methodology described in the introduction, we present a small case study inspired by a proprietary protocol which was developed at BNR in the early eighties [27]. The case study concerns the redesign and verification of this protocol whose purpose is to provide remote procedure call service to end users over unreliable and semi-reliable[6] connections. In what follows the emphasis is on architectural and modeling concepts, and the details of the protocol are omitted.

4.1 The Remote Task System

We begin with a highly abstract view and then gradually refine it. Let us call this top level system view `RT_System`. The block diagram of `RT_System` is given in Fig. 3.

In the figure, the box with rounded corners specifies `RT_System` as an *observation module*. An observation module represents a closed system—one that cannot be used as a component in a higher level module. Such a module is only subject to external observation through its defined interface. Here, the interface of `RT_System` consists of two *observation ports*, `ACCEPT` and `INVOKE`. The shape of an interface port defines its type; in Fig. 3, `ACCEPT` and `INVOKE` are *synchronous* ports.

`RT_System` in turn consists of two *components*, an *instance* of a module called `RT_Users` and an instance of a module called `RT_Service_Group`. These are *interaction modules*; i.e., their instances can serve as components in larger systems, or higher level modules. The interface of module `RT_Users` has two *output* ports (dark circles) and that of the module `RT_Service_Group` has two *input* ports (clear circles). In addition, `RT_Users` has a *provide* port (dark square) and `RT_Service_Group` has a *use* port (clear square) for shared variable type communication. Interface ports of interaction modules are uniquely identified by their location on block diagrams. They can be interconnected in various ways; *connections* are represented by different types of arcs. A solid arrow from an output port of one component to an input port of another denotes a synchronous, or *rendezvous*, channel. A provide port of one component can be connected to a use port of another by a solid line: the component with the provide port owns a shared variable environment which the component with the use port accesses.

The module `RT_Service_Group` is drawn as a box with cut corners. This is a special kind of module we call a *union*. A union is a collection of modules, called *members*, with compatible interfaces. We use a union to

[6]The term semi-reliable is used in the following sense: The connection can fail but provisions are provided for error recovery in the case of retransmission errors and message losses. Upper layers can thus assume reliable service for normal operation, but must also be prepared for more persistent connection failures to be able to resynchronize. If a message cannot be delivered correctly in the first attempt, the upper layer must be able to detect this and resynchronize.

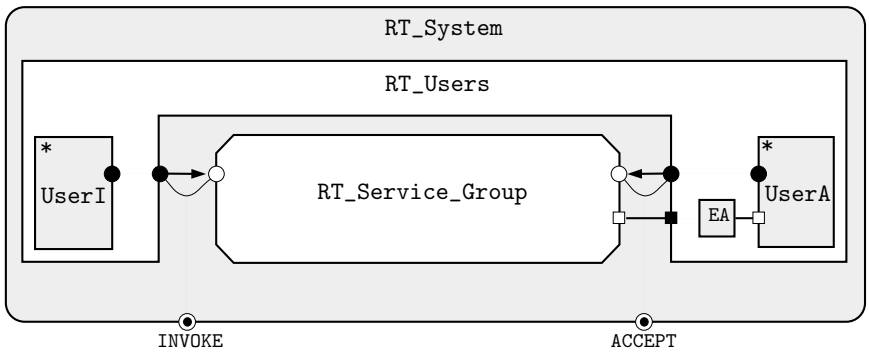

Figure 3: Block diagrams of RT_System and RT_Users. Primitive modules are indicated by a '∗'.

```
#include "Common.env"
#include "RT_Users.int"
#include "%RT_Service_Group.int"
chan UItoRT (extern INVOKE) = [0] of {byte};
chan UAtoRT (extern ACCEPT) = [0] of {byte};
init
{
  atomic{InitEA;
         RT_Users(UItoRT, UAtoIRT);
         RT_Service_Group(UItoRT, UAtoRT)}
}
```

Figure 4: The file %RT_System.obs.

represent either a set of replaceable modules of the same intended external functionality (a functional union) or customizable parts with varying external functionalities of a given system (a specialization union). A *group*, defined recursively, is either a union or a module which has a component that is a group.

The PROMELA model of RT_System is contained in the file %RT_System.obs given in Fig. 4. The mapping from the block diagram notation to PROMELA is straightforward. Note the one-to-one correspondence between observation ports and external channels.

The PROMELA models specify more information than those of block diagrams, such as content types of channels and initialization behaviors. For primitive modules, complete behaviors are specified in terms of process type definitions. The following file naming conventions are used: files for interaction modules have the suffix ".int", and those for observation modules have the suffix ".obs". For groups, we use the prefix "%", and for environment files we use the suffix ".env". An environment file declares shared global variables and defines macros used by more than one module. Shared variables are manipulated (tested, set, or initialized) through the macros defined in the associated environment file. Access to a shared variable environment is limited by provide-use connections in block diagrams. In this way, arbitrary patterns of shared variable communication are forbidden. In Fig. 4, the include file Common.env contains message type definitions and other macros common to all modules of the current example. InitEA is a macro defined in the file EL.env which in turn is included in the file RT_Users.int (not shown).

4.2 End Users and the Remote Task Service Group

The purpose of the module RT_Service_Group is to provide remote task service to two asymmetric end users in terms of four service primitives. No direct communication exists between these users. One end user, UserI, is the *source* (the Invoking user), and the other, UserA, is the designated *target* (the Accepting user). These make up the module RT_Users, as shown in Fig. 3. RT_Users also contains a shared variable environment, EA, which UserA uses. Outside access to this environment is made available by RT_Users through a provide port (dark square). In EA.env (not shown), two flag variables indicating the current status (idle/busy, blocked/unblocked) of the target are declared and the associated macros are defined.

The source can initiate a remote task session by issuing a RRinvoke service primitive to RT_Service_Group. Once the task has been initiated (the specified task has started its execution on the target's machine), the target can issue a number of Push and Pull service primitives to access the source's data space, and when the task

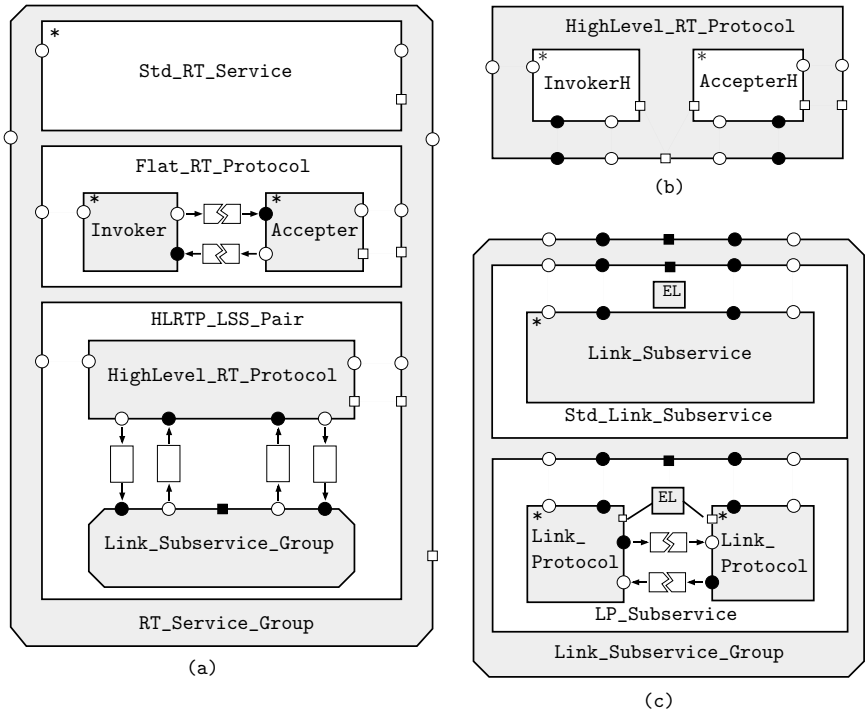

Figure 5: Block diagrams for `RT_Service_Group`.

completes, it issues a `RTcomplete` service primitive to signal the termination of the current session.

The block diagram of `RT_Service_Group` is given in Fig. 5a. This is a functional union (group) consisting of three member modules: The first module, `Std_RT_Service`, represents the service to be provided to the two end users in terms of the four service primitives. It has a concise, abstract specification given in Fig. 7. The macros `RTServiceAvailable`, `RTServiceNotAvailable`, `UserAError`, and `RTServiceFailure` are defined in `EA.env`. The second module, `Flat_RT_Protocol`, provides transparent remote task service over an unreliable connection in terms of a single layer protocol. This module combines two normally orthogonal functionalities—namely those of providing a remote task service and a semi-reliable data link connection—within one monolithic protocol. It consists of two module instances, `Invoker` and `Accepter`, communicating over two unreliable asynchronous connections (one for each direction). The third module, `HLRTP_LSS_Pair`, is a group which provides transparent remote task service over a semi-reliable connection by means of a more focused, higher level protocol which relies on the services of a data link sublayer. The data link sublayer is responsible for semi-reliable data transfer. Hence, the two orthogonal functionalities mentioned above are separated into two layers at the expense of some redundancy. The decomposition of the module `HighLevel_RT_Protocol` (Fig. 5b) is similar to that of `Flat_RT_Protocol`; it consists of two module instances, `InvokerH` and `AccepterH`, to interface with the end users. However, instead of communicating directly, these modules take advantage of the services of the data link sublayer module. `HighLevel_RT_Protocol` provides minimal error control; instead, this responsibility is delegated to `Link_Subservice_Group`, the module implementing the data link sublayer.

The PROMELA model of the module `RT_Service_Group` is specified as a logical switch statement entirely in terms of `cpp` macros. The file `%RT_Service_Group.int` is given in Fig. 6.

4.3 The Link Subservice Group

The block diagram of this group is given in Fig. 5c. `Link_Subservice_Group` is a functional union consisting of two modules: (1) a standard link subservice module, `Std_Link_Subservice`, specifying the external functionality of the data link sublayer in terms of the proper temporal ordering between 'send message' and 'receive message' primitives, and (2) a link protocol module, `LP_Subservice`, which implements the standard link subservice via two peer protocol entities, with full error and flow control, over an unreliable connection.

The module `Link_Protocol` is based on the alternating bit protocol. We translated the implementation given

```
#if nRT_Service_Group==nStd_RT_Service
#include "Std_RT_Service.int"
#define RT_Service_Group(spI, spA) Std_RT_Service(spI, spA)
#endif
#if nRT_Service_Group==nFlat_RT_Protocol
#include "Flat_RT_Protocol.int"
#define RT_Service_Group(spI, spA) Flat_RT_Protocol(spI, spA)
#endif
#if nRT_Service_Group==nHLRTP_LSS_Pair
#include "%HLRTP_LSS_Pair.int"
#define RT_Service_Group(spI, spA) HLRTP_LSS_Pair(spI, spA)
#endif
```

Figure 6: The file `%RT_Service_Group.int`.

```
proctype _Std_RT_Service(chan fromUI, fromUA)
{
 restart:
   fromUI?RTinvoke -> if
      :: RTServiceAvailable -> StartRT
      :: RTServiceNotAvailable -> goto restart
                     fi;
                     do
                       :: fromUA?RTcomplete -> goto restart
                       :: fromUA?Pull
                       :: fromUA?Push
                       :: UserAError -> goto restart
                       :: RTServiceFailure -> goto restart
                     od
}
#define Std_RT_Service(spI, spA)   run _Std_RT_Service(spI, spA)
```

Figure 7: The file `Std_RT_Service.int`.

in [32] to a PROMELA process type definition, and then defined `LP_Subservice` as the composition of two instances of this process type communicating via two unreliable asynchronous connections. We also explicitly modeled message losses and link failures, and added a resynchronization mechanism to recover from persistent link failures. The shared environment `EL` is used to model the effect that a user-level timeout is sufficiently longer than (i.e., cannot happen before) a link-level timeout.

4.4 Correctness Requirements for the Remote Task System

Our correctness criteria is based on trace equivalence; we require that all modules in the same functional observation group belong to the same equivalence class with respect to this relation. The observation group `RT_System` involves two functional unions, `RT_Service_Group` with three modules and `Link_Subservice_Group` with two modules, which together give rise to a functional group of a total of four observation modules. Let us enumerate these:

1. `RT_System` *where* `RT_Service_Group` = `Std_RT_System`.
2. `RT_System` *where* `RT_Service_Group` = `Flat_RT_Protocol`.
3. `RT_System` *where* `RT_Service_Group` = (`HLRTP_LSS_Pair` *where* `Link_Subservice_Group` = `Std_Link_Subservice`).
4. `RT_System` *where* `RT_Service_Group` = (`HLRTP_LSS_Pair` *where* `Link_Subservice_Group` = `LP_Subservice`).

The PROMELA model of each of the above modules is specified by a corresponding `cpp` macro file. For example, module 3 above is specified by:

```
#include "Mods.env"
#define nLink_Subservice_Group nStd_Link_Subservice
#define nRT_Service_Group nHLRTP_LSS_Pair
#include "%RT_System.obs"
```

where the include file `Mods.env` contains module selectors of the form "`#define n`*ModuleName Integer*."

According to our correctness criteria, the above functional group must constitute a trace equivalence class. The task structure given in Fig. 8 breaks down the correctness requirements of the remote task system into

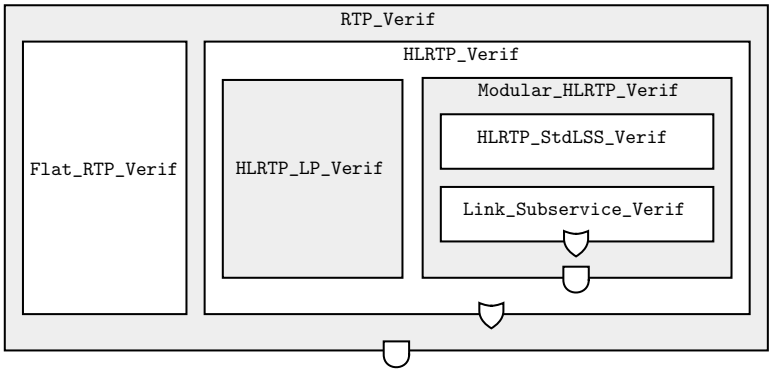

Figure 8: Task structure for the verification of `RT_System`.

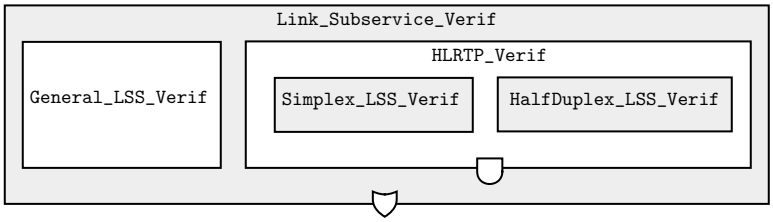

Figure 9: Task structure for the verification of the data link sublayer.

individual proof obligations, including those given rise by the functional observation group. Each box in Fig. 8 represents a task; unmarked boxes represent primitive tasks. Conjunctive and disjunctive (alternative) tasks are indicated with AND and OR gate symbols, respectively. The alignment of subtasks indicates whether the associated decomposition of the parent task is vertical or horizontal. The highest level task `RTP_Verif` is horizontally decomposed into two subtasks: `Flat_RTP_Verif`, which consists in checking the equivalence of module 1 to module 2 above; and `HLRTP_Verif`, a compound subtask constructed to decide the equivalence of module 1 to `RT_System` with `RT_Service_Group` = `HLRTP_LSS_Pair`. Note that these subtasks respectively correspond to the verification of `Flat_RTP_Protocol` and `HLRTP_LSS_Pair` *vis-à-vis* the service specification `Std_RT_Service`, where the common environment (context) is `RT_Users`. `HLRTP_Verif` is in turn horizontally decomposed into two disjunctive subtasks: `HLRTP_LP_Verif` and `Modular_HLRTP_Verif`. The former consists in deciding the equivalence of module 1 to module 4, which turns out to be inconclusive because of state explosion. The latter subtask is proposed as an alternative. This is an approximation which involves the vertical decomposition of `HLRTP_LP_Verif` into two conjunctive subtasks: `HLRTP_StdLSS_Verif`, which stipulates the equivalence of module 1 and module 3, and `Link_Subservice_Verif`, an independent verification of the data link sublayer which will be discussed later. This decomposition follows the two-layer architecture of `HLRTP_LSS_System`; the system in question is verified with the abstract service specification of the data link sublayer `Std_Link_Subservice`, and the actual implementation of the data link sublayer, `LP_Subservice`, is verified independently for conformance to `Std_Link_Subservice`.

4.5 Correctness Requirements of Link Subservice System

The subtask `Link_Subservice_Verif` is structured as shown in Fig. 9. The aim here is to verify the data link sublayer as an independent general purpose module since the functionality provided by this layer can be used by other higher level protocols. To do this, a specialization union, `LSS_User_Group`, is defined (Fig. 10). This group contains three types of general purpose users of the functional union `Link_Subservice_Group`. The module `SR_Pair` defines two asymmetric users in a sender/receiver-type simplex communication via an underlying data link service. It is used to verify the simplex operation of the data link sublayer. `IR_Pair` defines two asymmetric users in an initiator/responder-type half-duplex communication. This module is used to verify the half-duplex operation of the data link sublayer. And finally, the module `Generic_LSS_User` defines two symmetric users in a somewhat arbitrary full-duplex communication. It is used to verify the full-duplex operation of the data link sublayer.

Three observation groups are derived from the interconnection of `Link_Subservice_Group` with `LSS_User_Group`

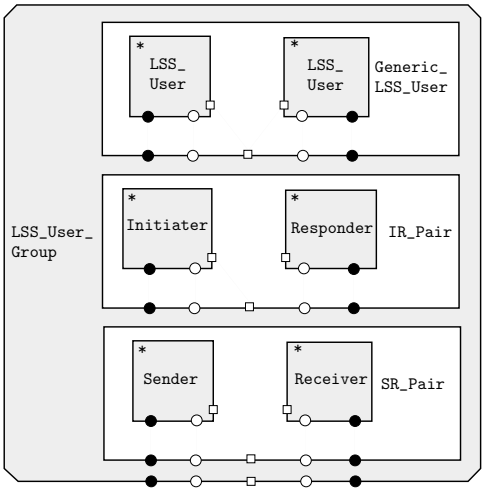

Figure 10: Block diagrams for `LSS_User_Group`.

via four asynchronous connections (Fig. 11). These groups differ only in terms of their interfaces (observation ports). The interface observed is that of `Link_Subservice_Group`. The first of these modules, `Simplex_LSS_System`, is for observing the simplex operation of the resulting system. Its PROMELA model is shown in Fig. 12. The second, `HalfDuplex_LSS_System`, is defined for observing its half-duplex operation. And the last one, `Generic_LSS_System`, is for observing the full-duplex operation. If we specialize each of these observation groups by selecting the appropriate member from `LSS_User_Group`, we obtain three functional groups:

- Functional Group 1

  1a) `Simplex_LSS_System` *where* (`LSS_User_Group` = `SR_Pair`
  and `Link_Subservice_Group` = `Std_Link_Subservice`).

  1b) `Simplex_LSS_System` *where* (`LSS_User_Group` = `SR_Pair`
  and `Link_Subservice_Group` = `LP_Subservice`).

- Functional Group 2

  2a) `HalfDuplex_LSS_System` *where* (`LSS_User_Group` = `IR_Pair`
  and `Link_Subservice_Group` = `Std_Link_Subservice`).

  2b) `HalfDuplex_LSS_System` *where* (`LSS_User_Group` = `IR_Pair`
  and `Link_Subservice_Group` = `LP_Subservice`).

- Functional Group 3

  3a) `Generic_LSS_System` *where* (`LSS_User_Group` = `Generic_LSS_User`
  and `Link_Subservice_Group` = `Std_Link_Subservice`).

  3b) `Generic_LSS_System` *where* (`LSS_User_Group` = `Generic_LSS_User`
  and `Link_Subservice_Group` = `LP_Subservice`).

Each of the above functional groups gives rise to an obligation involving one equivalence checking represented in Fig. 9 by a corresponding primitive subtask. It is understood from the disjunctive horizontal decomposition that the subtasks `General_LSS_Verif` and `Modular_LSS_Verif` are alternatives. These alternatives, however, do not represent equivalent obligations. Rather, `Modular_LSS_Verif` is proposed as an approximation to `General_LSS_Verif`.

4.6 Historical Notes and Results

The starting point of this case study was the original FSM descriptions of the modules `Invoker` and `Acceptor` and some accompanying text describing the purpose of the protocol. From the informal text, we obtained the service specification `Std_RT_Service` and defined the behaviors of the end users. From the FSM descriptions, we

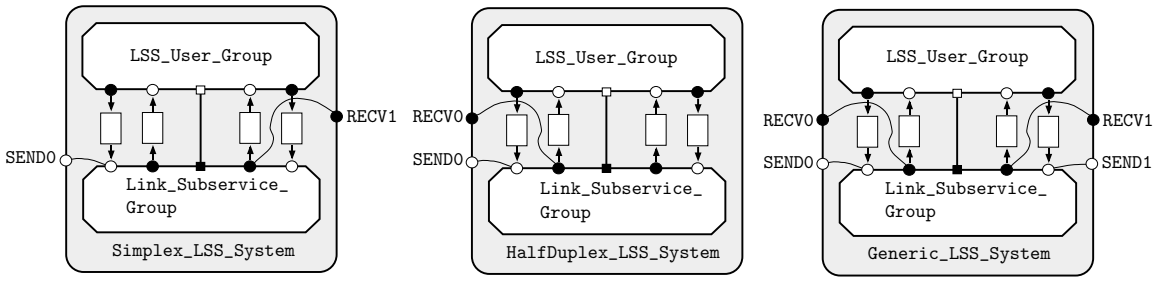

Figure 11: Block diagrams of the systems used in the verification of data link sublayer.

```
#include "Common.env"
#include "%Link_Subservice_Group.int"
#include "%LSS_User_Group.int"
chan U0toLSS (extern SEND0?) = [1] of {byte}; /* User0 to local LP */
chan LSStoU0                 = [1] of {byte}; /* local LP to User0 */
chan U1toLSS                 = [1] of {byte}; /* User1 to local LP */
chan LSStoU1 (extern RECV1!) = [1] of {byte}; /* local LP to User1 */
init
{
 atomic{InitEL;
        Link_Subservice_Group(U0toLSS, LSStoU0, U1toLSS, LSStoU1);
        LSS_User_Group(U0toLSS, LSStoU0, U1toLSS, LSStoU1)}
}
```

Figure 12: The file `%Simplex_LSS_System.obs`.

obtained an initial version of the module `Flat_RT_Protocol`. Although the original FSM descriptions considered only the behavior of a single remote task session, we considered a cyclic behavior in which a session may immediately be followed by a new one. First, by an independent exhaustive validation of the underlying state space using the SPIN tool alone, we found out that this first version of `Flat_RT_Protocol` contained deadlocks resulting from unspecified receptions, and some unreachable states. We modified the protocol and obtained a second version, which turned out to be free of both deadlocks and unreachable states. However, this time a subsequent SPINe verification revealed that the resulting protocol did not behave as expected with respect to its service specification `Std_RT_Service`; i.e., modules 1 and 2 of Section 4.4 were not trace equivalent. We identified the source of the problem as follows: the protocol was not self-stabilizing; it was not always possible for the two protocol modules to resynchronize when a session was abnormally terminated due to a timeout, leading to session overlap. We modified the protocol again, adding the appropriate provisions and constraining the behavior while remaining faithful to the intended external functionality. This resulted in a third and correct version of the module `Flat_RT_Protocol`. We also realized that the protocol was attempting to mix the orthogonal functionalities of providing simple data link and remote task session management services within one monolithic layer. This lead us to believe that separating the data link functionality and transferring it a lower layer module would highlight better the purpose of the protocol, thereby simplifying it further. This decomposition was captured by the module `HLRTP_LSS_Pair`. We then described the data link functionality by the functional group `Link_Subservice_Group` which was further decomposed and verified as described in Sections 4.3 and 4.5.

The results of SPINe verifications for the final version of the system are shown in Table 1. Each entry in the table corresponds to a task or subtask. All verifications were originally performed on a DEC workstation and then replicated on a Sparc2 with 32 MB of memory. Verdicts for non-primitive subtasks were calculated using 3-value logic. The total memory required by a conjunctive task is given by the maximum of the total memories required by the constituent subtasks. The total time is the sum of the total times of the constituents. For a disjunctive task, both the total memory and the total time required are the minimum of those required by the constituent subtasks. Some runs were inconclusive due to state explosion, in which case rough worst-case estimates are provided.

For this relatively small case study, our prototype implementation was sufficient since for most of the verifications, the state spaces involved were relatively small (less than 300,000 states). Although for two of the verifications the results were inconclusive due to time and space demands, the decomposition of the corresponding tasks into simpler, alternative subtasks proved successful; we were ultimately able to show our final design to be correct with a high degree of confidence.

Task or Subtask	*Verdict*	*No. of states*		*Mem*	*Time*
`Flat_RTP_Verif`	True	169/1313	4015/5649	1332	3:39
`HLRTP_LP_Verif`	?	69/?	$[3 \times 10^7]/[9 \times 10^9]$	?	?
`HLRTP_StdLSS_Verif`	True	69/1960	3045/46715	1416	2:28
`General_LSS_Verif`	?	1037/?	$[1 \times 10^7]/?$	?	?
`Simplex_LSS_Verif`	True	191/2031	10098/210845	1468	13:01
`HalfDuplex_LSS_Verif`	True	20/458	5192/51633	1456	4:14
`Link_Subservice_Verif`	True	—		1468	17:15
`Modular_HLRTP_Verif`	True	—		1468	19:43
`HLRTP_Verif`	True	—		1468	19:43
`RTP_Verif`	True	—		1468	23:22

Table 1: Verification results for the remote task system. *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 equivalence checking validation. A question mark indicates an inconclusive verdict due to memory/time limitations and the figures between square brackets are estimated upper-bound values.

5 Background and Related Work

The class of inductive relations was originally suggested in [10]. This reference develops the foundations of the WPS model in a slightly different setting . The WPS model results from an operational generalization of extended trace models [14, 15, 4, 3]. The semantic relations (equivalences and preorders) that underlie most of these models, as well as other semantic relations defined on the structure of a LTS, can be formulated as inductive relations using suitable transformations from LTS to WPS, provided that it is possible to give them a local characterization. Roughly a semantic relation can be characterized locally if it has a deterministic formulation dependent uniquely on properties that can be inferred locally. For example, nondeterminism, acceptance sets [14], failure sets [15], readiness sets [35], divergence (potential of an infinite internal computation), and deadlock are local properties; they can all be recorded in terms of external tests that do not require copying or branching. An example is testing equivalence [7]. However, for this relation the corresponding transformation and the underlying local relation are more complex than those for trace equivalence; see [11] for details. Likewise, many other extended trace relations can also be formulated as an inductive relation using the WPS model. For a comparative discussion of these, refer to [35] and [23, Ch. 3]. As an example of a relation which can not be formulated as such, we can give observation equivalence (or weak bisimulation) [25]—a relation which is too discriminating to have a local characterization. Observation equivalence requires a form of external testing that involves copying, which in turn gives it a global nature; see [1].

The definition of inductive relation given here is inspired by Park's elegant notion of bisimulation [28]. Similar definitions have been adopted by other semantic relations now referred collectively as simulations or bisimulations—for examples, see [25, 26, 30, 21, 2, 9]. Several algorithms have been proposed for checking such relations [9, 5, 22, 12]. As in [22] and [12], we use an on-the-fly algorithm which does not require the complete state space to be computed and stored *a priori*. The algorithm is based on the computation of a synchronized product, as done in [12], but uses a deterministic technique with explicit subset construction. Our algorithm differs from others in its generality; it treats a whole class of relations rather than a particular one. The algorithms described in [12] have been implemented in the LOTOS tool CÆSAR-ALDÉBARAN. This tool is similar to SPINe. Although its performance appears to be superior to that of SPINe—probably due to the avoidance of explicit subset construction—a detailed comparative evaluation is difficult because of the differences between LOTOS and PROMELA, but more importantly, because of the fact that their results are based on tests for stronger relations. Inductive relation checking is polynomial on the structure of a WPS, but the transformations from LTS to WPS are often exponential since they usually involve some kind of determinization to abstract from internal transitions. This complexity is not introduced unnecessarily; it is inherent. In particular, deciding trace equivalence is a PSPACE-hard problem [31]. Deciding most useful inductive relations on the structure of a LTS also turns out to be PSPACE-hard.

Compositional verification of concurrent systems based on semantic relation checking has been explored in [24], [20], and [23] in completely formal settings. In [24], abstractions of different levels are used as the basis of decomposition. In [20], a property holds true for a system (parallel composition of processes) if it holds true individually for its components, whereas in [23, Ch. 6], a property holds true for a LOTOS system defined in terms of a 'conjunction' of constraints if it holds true for each constraint individually. Here we regarded

decomposition—the construction of a modular verification strategy—as an informal design activity. As such, this activity itself is not necessarily subject to formal justification although the resulting primitive tasks involve formal proof checks. We used intuitive strategies—decompositions based on approximations and alternative tasks—when it was appropriate. Such strategies were necessary to reduce the complexity when the verdict of a primitive task was inconclusive due to extraordinary space or time requirements; the disadvantage was a lower level of confidence on the result.

Task structures based on AND/OR graphs have been used in describing development models for system specifications in Z [6]. We used them here for describing modular correctness requirements, distinguishing between horizontal and vertical decompositions in task structures. A similar distinction between vertical and horizontal steps was suggested by Turner [33] within the context of the step-wise refinement methodology for developing LOTOS specifications.

6 Conclusions and Future Work

We presented a compositional methodology for the specification and verification of concurrent or distributed systems based on deciding semantic equivalences in SPIN. The equivalence chosen was trace equivalence because our tool currently supports only this relation and its preorder, trace inclusion. It is a well-known fact that only safety properties can be reasoned about using these two relations. In practice, a more elaborate semantic relation, such as testing equivalence or weak bisimulation equivalence would be more useful. However, weak bisimulation equivalence—although it can be verified in polynomial time—does not have a local characterization, and hence, is not supported by the relation checking algorithm incorporated to SPIN. Other essentially weaker relations such as testing equivalence and failures preorder can be handled.

The methodology described is indeed independent of the semantic relation and the specification language used. In principle, it is applicable to a variety of other specification languages. In our opinion, LOTOS [19] would be a good candidate because of strong tool support behind it.

Using preorders instead of equivalences to capture the notion of one system implementing, or refining, another can also be envisioned. In this case, a functional group could be defined as a partial order of modules as opposed to an equivalence class.

So far the translation from the block diagram notation to PROMELA skeletons has been done manually. This process can be automated. Proof obligations can also be derived automatically from such diagrams. These two points are considered as future research directions.

Unfortunately, state explosion limits severely the usability of the current tool. Without clever decomposition strategies, large systems cannot be handled. Approximative techniques such as SPIN's bit-state hashing [16] and state space reduction techniques as proposed in [34], [13], and [29] are in principle applicable to semantic relation checking. An approximative method has been described in [5] and a reduction technique based on partial orders has recently been suggested in [8]. These techniques should be investigated in more detail to make semantic relation checking in SPIN a feasible complementary method to the more traditional temporal logic model checking and invariant analysis by state space exploration.

The SPINe tool is available through anonymous FTP. Please contact the author at **erdogmus@iit.ncr.ca**.

References

[1] S. Abramski. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[2] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced: preliminary report. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*, 1988.

[3] E. Brinksma. On the existence of canonical testers. Memorandum INF-87-5, Department of Informatics, University of Twente, Netherlands, 1987.

[4] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, number 197 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[5] 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.

[6] R. Darimont and J. Souquières. A development model: Application to Z specifications. In *Proceedings of IFIP WG8.1 Working Conference on Information System Development Process*. North-Holland, 1993.

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

[8] M. L. de Souza and R. de Simone. Using partial-order methods for the verification of behavioral equivalences. In *Proceedings of 8th International Conference on Formal Description Techniques*, pages 59–75, 1995.

[9] D. Dill, A. Hu, and H. Wong-Toi. Checking for language inclusion using simulation preorders. In *Proceedings of 3rd Workshop on Computer-Aided Verification*, 1991.

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

[11] H. Erdogmus. Verifying semantic relations in SPIN. In *Proceedings of 1st SPIN Workshop*, Verdun, Québec, Canada, Oct. 1995. INRS-Télécommunications.

[12] J. Fernandez and L. Mounier. Verifying bisimulations on the fly. In *Proceedings of 3rd International Conference on Formal Description Techniques*, 1990.

[13] P. Godefroid. *Using partial order methods to improve automatic verification methods*, pages 176–185. Number 531 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

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

[15] C. A. R. Hoare. A model of Communicating Sequential Processes. Technical Report PRG-22, Oxford University Programming Research Group, England, 1981.

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

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

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

[19] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS—A Formal Description Technique based on the temporal ordering of observational behavior, IS-8807*, 1988.

[20] B. Jonsson. Modular verification of asynchronous networks. In *Proceedings of 6th Annual Symposium on Principles of Distributed Computing*, Aug. 1987.

[21] K. G. Larsen. A context dependent bisimulation between processes. *Theoretical Computer Science*, 49:185–215, 1987.

[22] K. G. Larsen. Efficient local correctness checking. In *Proceedings of 4th Workshop on Computer-Aided Verification*, pages 35–47, 1992.

[23] 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.

[24] A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th Annual Symposium on Principles of Distributed Computing*, Aug. 1987.

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

[26] F. Orava. Verifying safety and deadlock properties of networks of asynchronously communicating processes. In *Proceedings of 9th International Workshop on Protocol Specification, Testing, and Verification*. IFIP, June 1989.

[27] F. Pachl. XMS remote communication protocols. Internal Report 2X830810JP/2, BNR, Montréal, 1983.

[28] 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.

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

[30] W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56(1):135–154, 1988.

[31] 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.

[32] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, N.J., 1981.

[33] K. Turner. A LOTOS-based development strategy. In *Proceedings of 2nd International Conference on Formal Description Techniques*, pages 135–191, 1989.

[34] A. Valmari. *A Stubborn Attack on State Explosion*, pages 156–165. Number 531 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[35] 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.