A Perfecto Verification: Combining Model Checking with Deductive Analysis to Verify Real-Life Software*

Yonit Kesten[1], Amit Klein[2], Amir Pnueli[3], and Gil Raanan[2]

[1] Dept. of Communication Systems Engineering, Ben Gurion University, Beer-Sheva, Israel, ykesten@bgumail.bgu.ac.il
[2] Perfecto Technologies Ltd. 103 Medinat Hayehudim St. Herzelia 46733, Israel, (http://www.PerfectoTech.Com), {Amit.Klein|Gil.Raanan}@PerfectoTech.Com
[3] Weizmann Institute of Science

Abstract. The paper presents an approach to the formal verification of a complete software system intended to support the flagship product of Perfecto Technologies which enforces application security over an open communication net.
Based on initial experimentation, it was decided that the verification method will be based on a combination of model-checking using SPIN with deductive verification which handles the more data-intensive elements of the design. The analysis was that only such a combination can cover by formal verification all the important aspects of the complete system.
In order to enable model checking of large portions of the design, we have developed an assume-guarantee approach which supports compositional verification. We describe how this general approach was implemented in the SPIN framework.
Then, we explain the need to split the verification activity into the model-checking part which deals with the control issues such as concurrency or deadlocking and a deductive part which handles the data-intensive elements of the design.

Keyword: models, verification (deductive methods, assume-guarantee, compositional) model checkers (SPIN, PROMELA)*, concurrent systems, Security, safety properties, Telecommunications, Object Oriented, Network protocols*

1 Introduction

The electronic commerce market has a growth potential that may bring it to revolutionize the world economy. The potential is based on the connectivity enabled by the Internet on one hand, and the willingness of customers and clients to do business on the net, on the other hand. The issue of willingness to do business on the Internet is mainly determined by how secure the customers

* This research was supported in part by the Minerva Center for Verification of Reactive Systems

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 173–194, 1999.

feel when they engage in their business transactions. This is where security (safety, assurance) comes into play, and it can be plainly seen that as the major concerns of the clients are security, privacy and assurance, then addressing these issues in a precise and scientific manner can result in both a robust solution to the technical problems, and a high assurance and confidence level on the side of the clients. In our understanding, formal methods are an enabling technology for security of large scale systems, especially software projects. As a software analysis and improvement tool, formal methods provide the fullest and most complete scientific means for safety and assurance. From the customer point of view, formal verification means high assurance, which is a key feature in the decision to use e-commerce. The paper describes the application of formal methods for the verification of a software product, which delivers e-commerce security for the Internet. The work is an ongoing effort that combines both the company's staff, and academic researchers. Results of the research are used as a feedback into the software development process, as well as improving our understanding of formal verification of a software project.

The rest of the paper is organized as follows. In section 2 we give a short description of both the hardware platform, and the software to be verified.

In section 3 we present the verification framework, discussing our choice of verification tools, system description languages and property specification languages. We discuss problems encountered with these languages in today's existing verification tools, explaining our choice of perform deductive verification manually.

In section 4 we present a compositional model-checking verification of the Top Level Design (TLD) of our system, using the SPIN tool. We present our methodology for using the well known assume-guarantee paradigm, tailored for the specific characteristics of our system and the SPIN tool.

In section 5 we discuss the verification of the detailed design of our system, combining model-checking and deductive proof techniques.

We conclude in section 6 with a summary of the presentation and some advice to developers of tools for deductive verification.

2 Description of Application

The software to be verified is an e-commerce Internet security server ("the product"). The software is usually placed between the e-commerce server (typically a web-server which provides a web-enabled interface to the e-commerce engine - database server, application server, etc.) and the Internet, usually on the seller's premises (see Fig. 1). The Internet connection of the e-commerce server must pass through the security server (the product). Clients of the e-commerce applications access the e-commerce server across the Internet, and through the seller's Internet gateway (routers, firewalls, etc.), and finally through the security server. The product, therefore, completely controls all the e-commerce transactions. Monitoring is performed at the application level, that is, the product "understands" the protocol used for the transactions.

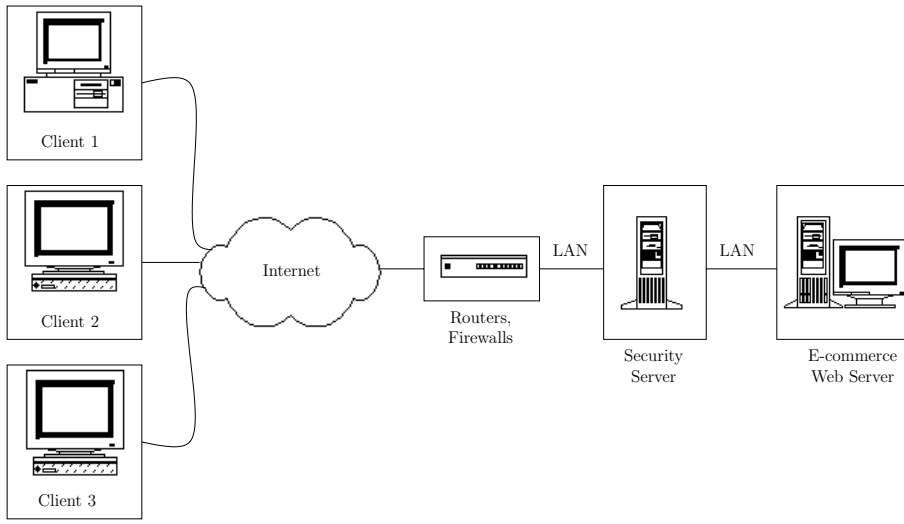

Fig. 1. Product Orientation.

The product consists only of software, running on top of an operating system. The hardware platform may consist of two CPU's, connected through a dedicated bus. A typical configuration would be two PC's connected via an Ethernet cable, but many other configurations are possible.

The product's logical architecture (patent pending) consists of a *Reducer* module and an *Enforcer* module, where the Reducer receives requests from the insecure zone (e.g. Internet) and *reduces* them into a proprietary simple protocol which represents the requests in a plain, unambiguous and robust manner; the Enforcer then *enforces* security rules on this representation and finally synthesizes the requests and transmits them over to the secure zone (the destination Web-Server). Coupling the logical architecture with the suggested physical architecture, such that the Reducer runs on one CPU, and the Enforcer runs on the other, enables the total separation of security functions (carried out by the Enforcer) and non-security functions (carried out by the Reducer), such that the non-security functions cannot compromise the security of the system. Hence, it is required to verify only the Enforcer, relieving us from the need to verify the Reducer.

As presented in Fig. 2, the security functions comprise of the following modules:

CM — This module interfaces with the physical device driver of the bus which connects the two CPU's. Data to be transmitted to the insecure CPU are handed to the CM (by the RM), and data that arrives from the insecure CPU is handled in the CM, which relays it to the RM.

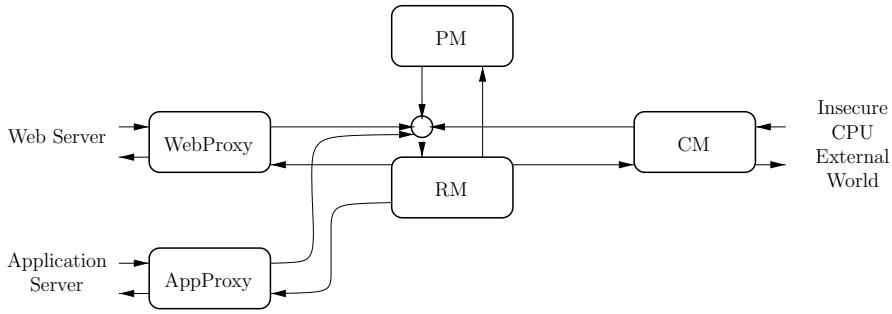

Fig. 2. System structure.

RM — The "nerve center" of the system. It is a data dispatcher, receiving data from various other modules and dispatching it to the destination modules.

PM — The security engine of the system. Processes all the transactions to/from the e-commerce server.

WebProxy — This is a proxy module which communicates with the web (e-commerce) server. The WebProxy receives data from the RM and relays it (typically over TCP/IP) to the web (e-commerce) server.

AppProxy — A proxy module which provides a function similar to that of WebProxy, for a non web-enabled server.

Incoming data (from the insecure CPU connected to the external, potentially adversary, world) arrives at the CM, which relays it to the RM, which routes it to the PM. The PM processes the data, ensures that it is safe, and returns it to the RM, which moves it to the WebProxy, which delivers it to the final destination, the e-commerce server. The software is written mostly in C++ (with some GUI functions in Java), containing tens of thousands lines of code. It is important to note that not all the code is relevant to the verification effort: there are some functions, notably the GUI (Graphic User Interface), which have nothing to do with the (security) properties to be demonstrated, thus are irrelevant to the whole verification process. These functions are not included in the above description.

3 The Verification Framework

As claimed in [MP95], a framework for formal verification should consist of the following components:

- A *computational model*, providing a common semantic base for the system and the properties we wish to establish for it.
- A *specification language* in which we can express the properties that need to be verified.

– A *system description language* in which we can describe the system whose properties need to be verified.
– A family of *verification techniques* by which such a verification can be successfully carried out. These usually include model checking, methods for deductive verification, and combinations thereof.

For our verification effort, we use the computational model of *fair discrete system* (FDS) ([KP98, KPR98]) which is slightly modified variant of the *fair transition system* (FTS), which is presented in [MP95] and underlies the *stanford temporal verifier* (STeP [BBC+95]).

The main specification language we use is (*linear*) *temporal logic* (LTL) [MP95]. However, in the context of verification by SPIN, we often express temporal properties by the corresponding automaton. In this paper we report only about the verification of *safety properties*.

For the set of verification techniques, we use model checking by SPIN, and manual deductive verification, using the deductive verification methodology expounded in [MP95] and implemented in STeP. Eventually, we intend to switch to computer-aided deductive verification by the STeP tool.

Currently, we use two system description languages according to the verification technique applied. For model checking with SPIN [Hol91], we use the SPIN system description language PROMELA. For deductive verification, the natural candidate is SPL, the system description language of STeP (and the one recommended in [MP95]).

A major problem we had to solve is how to provide an adequate and faithful representation of the concurrency programmed in our product (which is programmed in C++) within the framework of SPL, where the main difficulty was to represent dynamic creation and annihilation of processes. A solution to this difficulty is presented in the next Subsection.

While working on this representation and perfecting the deductive methods to handle our case studies within STeP, we meanwhile reverted to manual deductive verification, where we use SPL or sometimes even the actual C++ program as the system description language. As soon as we finalize the deductive methods to be used, we intend to incorporate computer-aided deduction, using the STeP tool into our process.

3.1 Dynamic Process Creation within SPL

The SPL modeling language was designed to accommodate *static* concurrency. That is, the number of processes running in parallel must be fixed at compile time, or at most, depend on an input parameter. To accommodate dynamic process creation, we declare in the SPL program an *infinite array* of processes, all of which await activation from a calling customer. A special *allocator* process hands around fresh indices in this array to all requesters. Given a process index, a requester may now communicate with the indexed server process.

To illustrate the application of this representation, consider the C++ program SUM-SQUARES, presented in Fig. 3.

```
class Number
{  // Number object, with the obvious interface
   public:
     Number(int v) // Constructor
     {  num=v; }
     // arithmetic operators, etc.
     int square()
     {  return num*num; }
   private:
     int num;
};
void main()
{  int n,sum=0;
   // Sum the first 10 squares
   for(n=1;n<=10;n++)
   {  Number x(n); //construct a Number object named x, initialized to n
      sum+=x.square(); // call the square member function
   }
}
```

Fig. 3. A C++ program SUM-SQUARES

In Fig. 4, we present the SPL representation of program SUM-SQUARES.

Note that the class *Number* is represented by a parameterized process *Number*[*i*] and the member functions of this C++ object have been implemented by the (synchronous) channels *cNumber* and *csquare*, which are local to *Number*[*i*]. While being local, implying that there are individual instances of these channels for each *Number*[*i*], they are also *public* in the sense that any external agent which knows the process-id i can send and receive messages through them referring to *Number*[*i*].*cNumber* and *Number*[*i*].*csquare*. Note that, according to this representation, processes are not really created but exist from the beginning of the run of the program.

In addition, there is an allocation process *allocate* whose role is to keep supplying new process-id's. The act of process creation is thus translated to making a possible client process aware of the name of a new process, by providing the client process with the index of that process. Most server processes keep waiting for some client to invoke their methods and until that happens they take no action.

The C++ invocation `Number x(n)` by the client (*main*) of the principal method *Number* has been separated into an allocation call providing the server-id which *main* saves in the local variable x, followed by a synchronous output of the value of n to channel *cNumber*[x]. Similarly, the C++ invocation `x.square()` has been translated to a synchronous input from channel *csquare*[x].

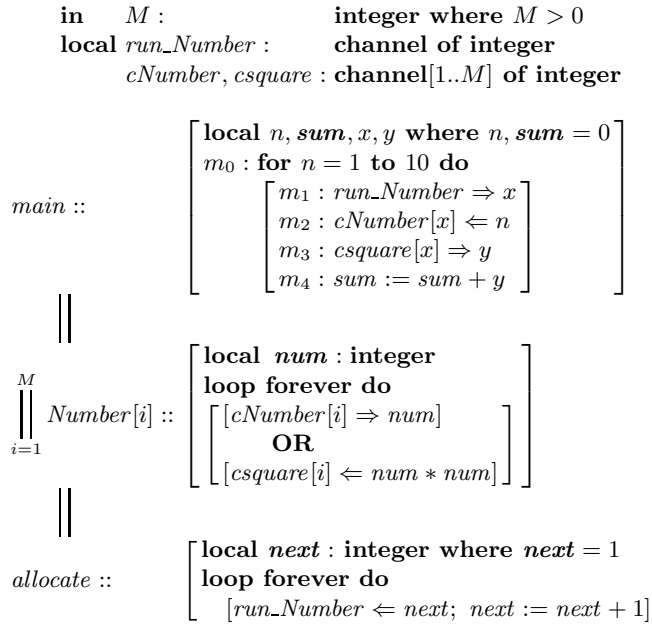

Fig. 4. The SPL representation of program SUM-SQUARES

4 Model Checking the TLD

Our first formal verification effort concentrated on the verification of the top level design (TLD) of the system. Through this experiment, we hoped to identify a method powerful enough to handle the verification of the complete system. In this experiment, we chose to use the model checker SPIN which was advertised as being specially designed for the verification of software and communication protocols, in particular.

Unfortunately, in spite of the high abstraction we applied in deriving the TLD view of the system, it was still too big to be completely verified in one go. This forced us to revert to compositional model checking based on the assume-guarantee paradigm. This paradigm is very well known and many variants have been developed over the years, e.g., [CM81], [Jon83], [BK85], [Pnu85], [dR85], [Zwi89], [PJ91], [AL93], [Jon94], [KM95], [AL95], [CC95], [CGL96], and [XdRH97]. Yet, for the use of this paradigm in our context we had to develop our own variant (heavily inspired by all this previous work).

In this section, we report about our approach to the verification of the top-level design of our system, using compositional model checking with the SPIN tool.

4.1 Systems and Their Safety Properties

We will illustrate our approach to compositional verification on the simple case of two processes communicating by synchronous channels, as depicted in Fig. 5.

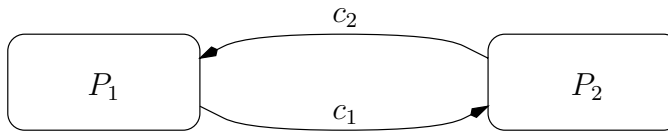

Fig. 5. Two Processes.

We assume that processes in the system are presented as PROMELA programs, where communication is restricted to synchronous (rendezvous) channels. The computational model we use for representing the behavior of systems and their specifications assumes V a finite set of typed *state variables*. In particular, for each channel c_i with range of messages R_i, the set V includes a corresponding variable C_i which ranges over $R_i \cup \perp$, where the special value $\perp$ denotes a state in which there was no communication over channel c_i.

A system *state* is any valuation of the state variables consistent with their types. For a state s and a state variable $x \in V$, we denote by $s[x]$ the value assumed by x in state s. A *run* of a system S is a finite non-empty sequence of states $r : s_0, s_1, \ldots$ such that s_0 satisfies the initial condition, and for every j, $0 < j < |r|$, the state s_j can be obtained from s_{j-1} by executing one of the statements in the program for the system S which is enabled on s_{j-1}. In particular, $C_i = m \neq \perp$ in state s_j iff the statement executed in passing from s_{j-1} to s_j sent the message m on channel c_i, e.g., by a sender executing $c_i!\, m$ jointly with a receiver executing $c_i?\, y$.

For a subset of the state variables $U \subseteq V$ and a state s, we denote by $s{\downarrow_U}$ the *projection* of the state s on the subset U. That is, $s{\downarrow_U}$ is the U-state obtained by removing from the state s the valuation of the variables which belong to $V - U$. In particular, $s{\downarrow_{C_i}}$ retains only the value assumed by the variable C_i, namely, the message sent on channel c_i in the step leading to this state (we write $s{\downarrow_{C_i}}$ as an abbreviation for $s{\downarrow_{\{C_i\}}}$). For a run $r : s_0, s_1, \ldots$, we denote by $r{\downarrow_U}$ the projected run $s_0{\downarrow_U}, s_1{\downarrow_U}, \ldots$. Finally, for a run r and a channel variable C_i, we denote by $r{\Downarrow_{C_i}}$ the *compressed projected run* obtained by removing from $r{\downarrow_{C_i}}$ all the bottom elements. For a run r, $r{\Downarrow_{C_i}}$ represents the list of messages emitted on channel c_i during the run in the order of their emission. In the CSP terminology, this is called the c_i-restricted trace of r [Hoa84].

In this study, we were mainly interested in the study of *safety properties* [Lam77]. This is why it is sufficient to consider the semantics of a system as given by the set of all of its runs, and consider as properties to be verified only *safety properties*. The main feature of a safety property φ is that if it is violated by a run r then it cannot be satisfied by any run extending r. A typical example

is the property specifiable by the formula $\varphi : \Box(x > 0)$ stating that, at all states of all runs, the value of x is always positive. It is obvious that if a run r violates φ then one of the states in r must have a non-positive value of x and, therefore, no extension of r can satisfy the requirement "x is always positive".

We write $r \models \varphi$ to denote that the run r satisfies the property φ, and write $S \models \varphi$ if φ is *S-valid*, i.e., all runs of S satisfy r.

A safety formula φ is said to be a *channel property* of channel c if

C1. $r \models \varphi$ for every r such that $r\Downarrow_C$ is the empty sequence. That is, φ holds over all runs which did not send even a single message on channel c.

C2. If r_1 and r_2 are runs such that $r_1\Downarrow_C = r_2\Downarrow_C$, then $r_1 \models \varphi$ iff $r_2 \models \varphi$. That is, the truth value of φ on a run r is fully determined by $r\Downarrow_C$, the sequence of messages transmitted by the run r on channel c.

4.2 A Compositional Proof Rule

Normally, there is no chance of being able to verify, using model checking techniques, any of the properties of the system by submitting the entire system to a model checker such as SPIN. A frequently used approach, to which we refer as the *compositional approach* is to consider modules (processes) of the system separately and verify every property by considering only the processes which are responsible for the variables on which the property depends.

For example, in the system of Fig. 5 we may wish to prove two safety properties: φ_1 and φ_2, where each φ_i depends only on the variables determined by process P_i, for $i = 1, 2$. This suggests that property φ_1 should be model-checked on a model consisting of process P_1 alone. Unfortunately, while the property φ_1 may depend only on variables manipulated by process P_1 the range of behaviors of P_1 when run alone may differ radically from its behaviors when coupled with P_2. In particular, when run in isolation it may produce a behavior which violates the property φ_1, while such a behavior is impossible in the real system due to the interaction with P_2. There is a danger that we may erroneously conclude that property φ_1 is not valid over the system while, in fact, it is valid.

To overcome this difficulty, we never study any of the processes in complete isolation. Instead, we identify channel properties, say I_1 and I_2 which capture the properties of the communication on channels c_1 and c_2 which (for our application) is the only way the two processes can interact with one another.

As a first step in the application of this idea, it is necessary to confirm that the proposed channel properties are indeed valid for all computations of the joint system $P_1 \| P_2$. This can be done using rule COMP presented in Fig. 6.

$$\begin{array}{ll} \text{L1.} & P_1 \models (I_2 \rightarrow I_1) \\ \text{L2.} & P_2 \models (I_1 \rightarrow I_2) \\ \hline & P_1 \| P_2 \models I_1 \wedge I_2 \end{array}$$

Fig. 6. Rule COMP.

Such a rule is often described as an *assume-guarantee* paradigm. Premise L1 of the rule can be interpreted by saying that, under the *assumption* that the environment maintains the property I_2 on channel c_2, process P_1 *guarantees* to maintain the property I_1 on channel c_1. Premise L2 states the symmetric obligation for process P_2. The rule claims that if these two obligations hold then both I_1 and I_2 will be maintained in all runs of the combined system $P_1 \| P_2$.

This rule is not sound for arbitrary properties I_1 and I_2. In the general case, the most general conclusion that can be inferred from premises L1 and L2 is $(I_1 \rightarrow I_2) \wedge (I_2 \rightarrow I_1)$ which does not necessarily imply $I_1 \wedge I_2$. However, in our case we are guaranteed of the following assumptions:

A1. $s_0[C_1] = s_0[C_2] = \perp$, for every initial state s_0. That is, no message has been transmitted on entering the initial state.
A2. I_1 and I_2 are safety channel properties for channels c_1 and c_2, respectively.
A3. At most one message can be transmitted at any execution step. That is, either $s[C_1] = \perp$ or $s[C_2] = \perp$ for every state s appearing in a run of $P_1 \| P_2$.

As we will now show, these assumptions guarantee that rule COMP is sound.

Claim. Under the assumptions A1–A3, rule COMP is sound.

Proof: Assume, to the contrary, that rule COMP is unsound. Let $r : s_0, s_1, \ldots, s_n$ be one of the shortest counter examples to the rule. That is, r satisfies premises L1 and L2 but does not satisfy the conclusion $I_1 \wedge I_2$.

For $k \leq n$, we denote by $r^{(k)}$ the k-prefix $r^{(k)} : s_0, \ldots, s_k$ of r.
Obviously, $n > 0$. This is because, due to assumption A1, $r^{(0)}\Downarrow_{C_1} = r^{(0)}\Downarrow_{C_2}$ is the empty sequence, and by clause C1 of the definition of a channel property, both I_1 and I_2 should hold over $r^{(0)}$.

Since r is one of the shortest counter-examples, we can assume that $r^{(n-1)} \models I_1 \wedge I_2$. By assumption A3, either $s_n[C_1] = \perp$ or $s_n[C_2] = \perp$, and we assume, with no loss of generality, that $s_n[C_2] = \perp$. Consequently, $r^{(n)}\Downarrow_{C_2} = r^{(n-1)}\Downarrow_{C_2}$ and, due to clause C2 of the definition of a channel property and the fact that $r^{(n-1)}$ satisfies I_2, it follows that also $r^{(n)} = r$ satisfies I_2. Applying premise L1 to r, we conclude that I_1 holds over r. Thus, both I_1 and I_2 are satisfied by r, contradicting our hypothesis that r does not satisfy $I_1 \wedge I_2$. □

Once we establish I_1 and I_2 as valid channel properties for the system $P_1 \| P_2$ we can use them for model-checking any local property φ_1 which only refers to the variables manipulated by process P_1. To do so, we model check the validity of the implication $I_2 \rightarrow \varphi_1$ over process P_1. Note that all necessary verification tasks apply model checking to a single process rather than to the complete system. This is the main advantage of the compositional approach.

Obviously, the method described here can be applied to a system consisting of an arbitrary number of processes, as long as every channel connects a unique sender to a unique receiver.

4.3 Implementing the Compositional Verification in SPIN

Since our application involves more than two communicating modules, we decompose the TLD into clusters of processes, each cluster small enough so that it can be locally verified by SPIN. Having defined such a cluster, we identify the incoming and outgoing channels, connecting the cluster to the rest of the design.

In Fig. 7 we present the general setup of a typical cluster.

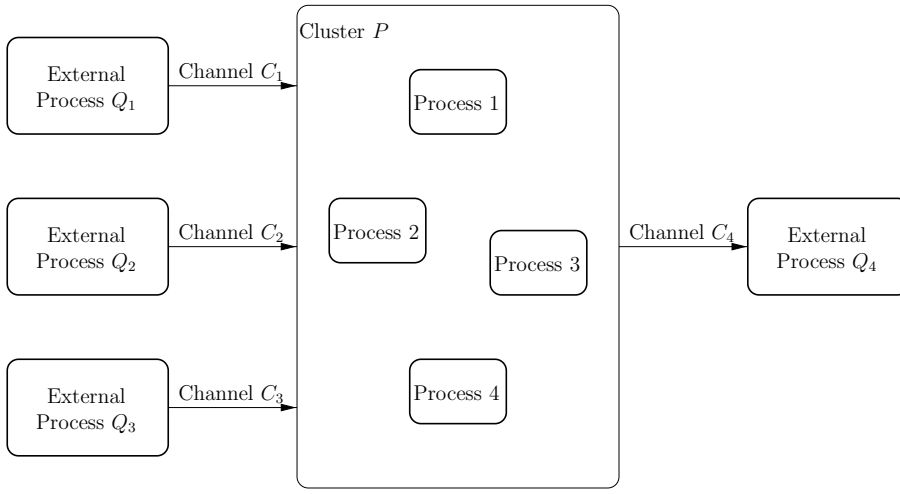

Fig. 7. Decomposition

The general form of the local verification that has to be applied to each of the clusters is

$$P \models (I_1 \wedge I_2 \wedge \ldots \wedge I_n \rightarrow J), \tag{1}$$

where $I_1, \ldots, I_n$ are the *assumptions* for the behavior on the incoming channels, such as $C_1, \ldots, C_3$ in Fig. 7, and J is the property that the cluster *guarantees* on its output channel, such as C_4 in Fig. 7. It is assumed that $I_1, \ldots, I_n, J$ are channel safety properties. The composition rule requires that we associate a unique invariant $I_{i,j}$ with every channel $C_{i,j}$ connecting cluster P_i to cluster P_j. The invariant $I_{i,j}$ will appear as an assumption (one of the I_k's) in the verification task for cluster P_j and as a guarantee (the J) in the verification task for cluster P_i.

Let us consider how to represent the verification task (1) for a representative cluster P to the SPIN tool. Obviously, we can represent the cluster of processes P as a set of concurrent PROMELA processes. The remaining question is how to represent the assumptions $I_1, \ldots, I_n$ and the guarantee J where, up to now, we considered these specifications as temporal formulas.

In theory, SPIN provides a special mechanism for representing non-trivial temporal properties. This is the *never* claim which identifies a single automa-

ton (represented as a PROMELA process) which runs in synchronous parallelism to the application and monitors its behavior. Unfortunately, this mechanism is too restricted for us since we need to attach to the cluster a set of automata corresponding to the assumptions and the guarantee.

Consequently, we decided to construct our own processes which represent automata and monitor for the satisfaction of their corresponding temporal properties. Unlike the single *never* automaton, these automata processes run in asynchronous parallelism (interleaving) to the rest of the system, but communicate with the cluster via synchronous channels.

For the guarantee property J, we construct an *acceptor process* A, which is an automaton *accepting* precisely the set of sequences satisfying J.

For each assumption property I_i we construct a *generator process* G_i, which is an automaton *generating* precisely the set of all sequences satisfying I_i.

We refer to the set of processes P, G_i and A as the PROMELA *model* for the verification task (1). The verification is performed by running the PROMELA model within SPIN. Each generator and acceptor is connected to the cluster by a single channel. In Fig. 8, we present such a PROMELA model.

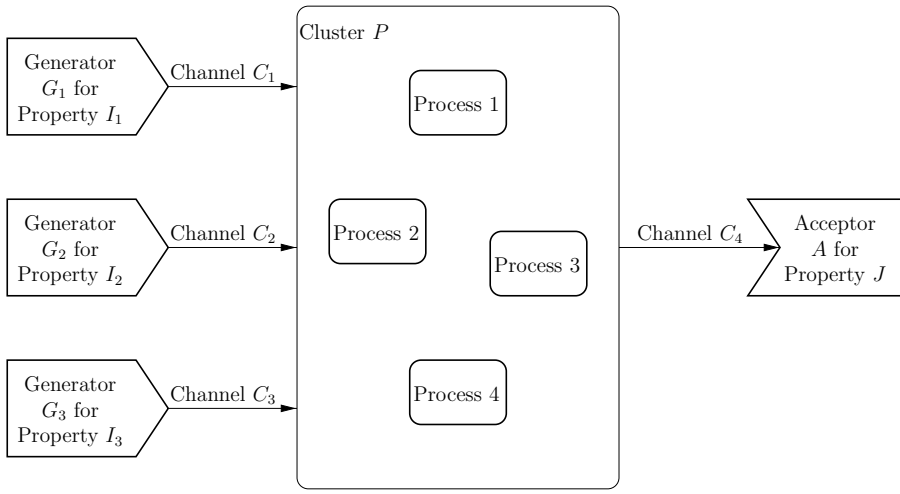

Fig. 8. A PROMELA model for a cluster.

4.4 Construction of Acceptors and Generators

Since each channel $C_{i,j}$ is associated with a single channel safety property $I_{i,j}$, we have to construct for such a channel first an acceptor $A_{i,j}$ which accepts all the behaviors satisfying $I_{i,j}$, and then a generator $G_{i,j}$ which generates precisely the set of all sequences satisfying $I_{i,j}$.

Acceptors: Construction of acceptors is fairly easy. Assume the channel name is "c", and it should comply with a channel safety property J. We assume that we know how to construct for the property J a finite state deterministic automaton over the finite alphabet of c, with a single error state E (where all other states are accepting), which accepts precisely the sequences satisfying J. The translation of this automaton to PROMELA is straightforward. Every non-error node of the automaton is represented by the input statement $c?y$, followed by a case selection statement branching to different locations according to the value of the input y. The error state is represented by the statement `assert(0)` which aborts the computation, announcing an error.

For the simplest cases, this *explicit state* representation of the automaton for J and its PROMELA translation is adequate. However, in many cases, the alphabet is structured, such as the integers in the range 0..255 and the conditions are often expressed succinctly by predicates over the alphabet, such as $y < 128$. In these cases, the PROMELA acceptor is represented more compactly using auxiliary variables to represent part of the state.

For example, in Fig. 9, we present a PROMELA acceptor that accepts all sequences of permutations of $\{1, 2, 3\}$.

```
active proctype accept_c()
{       byte x,y,z;
        atomic {
        do
        :: c?x -> assert((1<=x) && (x<=3));
           c?y -> assert((1<=y) && (y<=3) && (y!=x));
           c?z -> assert((1<=z) && (z<=3) && (z!=x) && (z!=y));
        od;    }
        unless end_ver;
}
```

Fig. 9. An acceptor for all permutations over $\{1, 2, 3\}$.

Note that this automaton may reject the input at three locations. It rejects after the first input iff this input is not in the range [1..3]. It rejects after the second input iff this input is not in the range [1..3] or it is equal to the first input. Finally, the last input is rejected iff it is not in the range or it equals one of the previous input. Note that the explicit state automaton corresponding to this PROMELA process will have at least 8 states and 21 edges.

Generators Construction of Generators is less straightforward. An important element in the construction of generators is that we should have a systematic way of translating an acceptor to a generator, automatically if possible. This is important because, as we have already mentioned, for each channel $C_{i,j}$ we need an acceptor $A_{i,j}$ and a generator $G_{i,j}$. The soundness of the composition rule

hinges on the assumption that the set of sequences generated by $G_{i,j}$ is equal to the set of sequences accepted by $A_{i,j}$, i.e. that they refer to the same property.

For the case that the acceptor is derived from an explicit-state automaton, the conversion from acceptor to generator is straightforward. Every edge connecting automaton state q_i to $q_j \neq E$ and labeled by the letter A, should be translated to the PROMELA statement $c!A$ connecting the locations corresponding to q_i to q_j. In the case that q_j equals E, the error state, we omit the corresponding output statement. Such a generator will never generate a wrong output.

The situation is more involved in the case the acceptor process uses auxiliary variables, such as the acceptor of Fig. 9. Due to space limitations, we will only present in Fig. 10 the generator obtained by applying our systematic conversion to this acceptor.

```
active proctype generate_c()
{       byte x,y,z;
        atomic {
        do
        :: x = random(1,3); c!x;
           y = random(1,3); end_var = (y==x); c!y;
           z = random(1,3); end_var = ((z==x) || (z==y)); c!z;
        od;    }
        unless end_ver;
}
```

Fig. 10. A generator for all permutations over $\{1, 2, 3\}$.

The way this generator operates is that it first draws a random number in the range 1..3 and outputs it. Next it draws a second random number y in the same range. After drawing it, the generator applies the same test to y relative to x. If y should be rejected, the automaton does not cause an error abortion of the complete system by asserting *false*. Instead, it raises a special boolean flag `end_var` which causes the complete system to terminate immediately but in a non-error state. Technically, immediate termination is ensured by enclosing all processes in the system by the `unless end_var` clause which interrupts and terminates all processes as soon as the flag `end_var` is raised.

4.5 Results of the TLD Verification

The case study described below served to demonstrate the strength of the method together with its limitations. Eventually, it was decided to discontinue the application of this method, although the techniques developed may be of some use in further developments.

The case study included the 5 modules presented in Fig. 2 : WebProxy, PM, CM, RM, and AppProxy, plus an additional trivial module that has been added for technical reasons. The data transmitted over the channels connecting

these modules contains 3 or 4 fields (depending on the channel): a "destination" field, a "session" field, and a "data" field, with sometimes a "from" field. In the table below, we summarize for each module the number of incoming and outgoing channels, and the number of overall processes (including acceptors and generators) which were involved in its modular verification:

Module Name	Incoming Channels	Outgoing Channels	Overall Processes
CM	2	2	7
RM	1	4	7
PM	1	1	4
WebProxy	2	2	7
AppProxy	2	2	7

The results from the verification, carried out by SPIN, were all positive. However, the execution (on a Pentium-II/333MHz, 100MHz bus, 512KB L2 cache, 256MB SDRAM, SPIN-3.2.3/Linux-2.0.29, -DSAFETY, -DMA=117 used) took too long (6 hours and 46 minutes for the most complex module - RM). The experiment is considered, therefore, to be unsuccessful in that it appears to reveal a scalability problem inherent to the implementation of the Acceptors/Generators scheme. While we gained a lot of expertise and insight by developing and applying this technique, and managed to verify a certain portion of the TLD, we decided not to use this method for the verification of the detailed design.

5 Detailed Design Verification: Combining Model Checking with Deductive Methods

The results reported in the last section demonstrated that SPIN alone cannot meet the challenge posed by a complex software system. At the beginning we thought that the problem was inherent to an explicit-state model checker such as SPIN, and that switching to a symbolic model checking, such as SMV [BCM+92], may solve the scalability problem. Indeed, we tried SMV on a subset of the case studies and observed a speedup factor of about 1000:1. However we soon realized that no matter how fast they are, no model checking tool is able to handle the complete problem, in particular, it cannot handle data intensive problems.

5.1 Why Combine the Methods?

It appears that the most versatile and powerful method of handling data-intensive problems is by using a deductive method. The main candidate tools for supporting deductive verification are STeP [BBC+95] and PVS [OSR93]. After some preliminary attempts we ran into the problem that none of these tools provided us with a system description language that fits our needs, namely analyzing multi-threaded programs written in C++.

The PVS tool requires translating the program into formulas of high-order logic, and if we wish to perform verification of temporal properties, it is also necessary to include the theory of temporal logic.

At first glance it seems that STeP is more user friendly since it has its own system description language SPL. When we tried to use STeP, it became clear very quickly that, while SPL is quite adequate for dealing with distributed systems, the representation of dynamic object creation, as explained in Section 3, requires a special translation which hinders its application for verifying C++ programs.

Obviously, the problems we complain about are purely technical, and all that is needed to solve them is a translator from C++ to either high-order logic or to SPL. Given enough time, we probably would have constructed such a translator and then continued to use these powerful tools. Being under intense time pressure, our final decision was to use the temporal deductive methodology proposed in [MP95] but conduct the proofs *manually*.

The strategy we have formulated and applied so far consists of a combination of *model checking* with *deductive verification*, where

- *Model checking* is used for handling *control-related* issues of deadlocks, mutual exclusion, and non-interference. It is applied to a simplified model of the system in which almost all data has been abstracted away.
- *Deductive verification* is used for analyzing the data-intensive parts of the system.

Besides separation of concerns, the main interaction between the two methods is that a thorough analysis of non-interference can lead to a significant simplification of the deductive verification task. In particular, when a certain segment in a given object's member function is known to be "isolated" from outside interference, we can verify its data transformations as though it were a sequential program.

5.2 Augmenting Deductive Verification by Model Checking

The chief purpose of employing Model Checking in our scheme is to resolve concurrency issues automatically. The immediate candidates are the following:

– Mutual Exclusiveness (Non-Interference)
– Deadlocks

For the purpose of verifying the security of our system, we are not interested in deadlock freedom properties, since if the system enters a deadlock state, security is not breached (although performance drops to zero). However, in order to verify that while the software is running, it does only what it is supposed to do, we lean heavily on non-interfernce properties. Non-Interference is crucial for the deductive phase: then, it is very helpful to know that some variables can change only by the thread being analyzed (and not by any concurrent thread). To some extent, non-interference can be thought of as "serializing" threads, and making them independent of each other (in limited code segments), with the obvious benefits for the deductive analysis.

The basic temporal formula that represents non-interference is:

$$\Box(at\_\ell_i \wedge at'\_\ell_i \rightarrow x = x') \tag{2}$$

This can be proven using SPIN in the following way: the system to be analyzed is represented as a set of PROMELA processes. Then, for each variable `x` for which the above formula should hold (at label `l` of process `P` with process identity `i`) an analysis is carried out, and each transition that writes on `x` is marked by a label. The SPIN equivalent of the above formula then takes the following form (for each label a in process Q with process identity j):

`[](P[i]@l -> !`Q`[`j`]@`a`)`

This formula can be checked as a `never` claim by SPIN , for each (Q,j,a) tuple. Note that we abstract away all the data manipulations.

The analysis can be extended naturally to cases in which even *access* to a variable should not occur while a certain thread is in a certain state.

Below, we illustrate the use of model-checking for establishing non-interference. The example is a system of threads accessing a shared data object, using locking mechanism to synchronize the access to the object. Threads can read, write or destroy the data object. An object can be read concurrently by several threads. However, when a data object is written or destroyed by one thread, it must not be read by other threads. A "group" is a special object, containing validity information for the data object. Due to implementation restrictions, the data object's validity must be ensured via a lookup in the group object before the data object is accessed. The group object itself is accessible directly, and can be both read and written.

There are K "user" threads in the system, which write the data object, and a "garbage collection" thread, which destroys inaccessible objects. These are, of course, simplified versions (for the sake of the example) of the actual threads, that may read and write nondeterministically to several objects, repeatedly.

Model checking provides the necessary non-interference assurance, stating that the data object cannot be written (or destroyed) while it is read, nor can it be destroyed and written at the same time.

The PROMELA code is as follows:
The "user" threads:

```
active [K] proctype user()
{
        ReadLock(group);
        /* pointer to object */
        if
        :: object_exists ->
           WriteLock(object);
           ReadUnlock(group);
           do
           :: skip -> /* may choose this branch */
```

```
write:        skip; /* perform writing */
              WriteUnlock(object);
              /* give up control */
              /* resume control */
              WriteLock(object);
           :: skip -> /* may choose this branch */
              WriteUnlock(object);
              break;
           od;
        :: else ReadUnlock(group);
        fi;
}
```

The "garbage collection" thread:

```
active proctype remove()
{
        WriteLock(group);
        WriteLock(object);
        object_exists=false;
        WriteUnlock(group);
        WriteUnlock(object);
        /* perform removal */
destroy: skip;
}
```

This code uses the four following macros for semaphore simulations:

```
#define ReadLock(var) atomic {!var.WRITE -> var.READ++ }
#define ReadUnlock(var) {var.READ--}
#define WriteLock(var) atomic {!var.WRITE && !var.READ ->   \
                               var.WRITE=true}
#define WriteUnlock(var) {var.WRITE=false}
```

Note that the "user" threads release and re-acquire their lock on the object as necessary. This is an optimization applied to the original locking design, wherein the threads held their lock on the object throughout the writing phase. The outcome of this optimization is revealed once SPIN is used to re-assess the safety properties (assume `K=2`):

```
[] !(user[1]@write && user[2]@write)
[] !(user[1]@write && remove[0]@destroy)
[] !(user[2]@write && remove[0]@destroy)
```

Contrary to the original design, SPIN reports a `never` claim violation for this system. Examining the error trail uncovers the following flaw: the basic assumption of the design is that a "user" thread accesses an object only after checking that its existence bit (in the group object) is up. This no longer holds

for the modified system: the `remove` thread may engage in deletion operation (once a "user thread" gives up his lock on the object), with a "user" thread re-acquiring the lock and writing data on the object, a scenario which violates the second or the third LTL formula of the above.

This resulted in a complete redesign of the whole locking scheme.

5.3 Deductive Verification of Data Transformations

Having verified non-interference between threads (equation 2), we can now verify heavy data transformations in single threads, as though they are sequential programs. We give two examples using the deductive verification rules presented in [MP95].

Example 1: Using the WAIT rule to ensure a proper implementation of a procedure. Assume a procedure of the following structure:

$$\begin{bmatrix} E : (entry \quad point) \\ B : \qquad \cdots \\ X : \end{bmatrix}$$

We write $\Box((at\_E \wedge precondition) \rightarrow (at\_E \vee at\_B)\, \mathcal{W}\, (at\_X \wedge postcondition))$ to state the fact that once control is in the routine then control remains within the procedure body B until it gets to the exit point X, with the postconditions satisfied, thus guaranteeing the proper ($precondition \rightarrow postcondition$) action of the procedure.

For instance, this rule can be used to verify a sort algorithm, provided the array it sorts does not change (while program counter is inside the sort code) by other threads. The *precondition* may be identically T in this case, and the *postcondition* would require a sorted array which is a permutation of the original array.

Example 2: Using the BACK-TO rule to ensure that if a procedure terminates successfully, then some expected action has been previously executed. Assume a procedure of the following structure:

$$\left[\begin{array}{l} \qquad\qquad \cdots \\ \textbf{if } (\cdots) \textbf{ then} \\ \quad \left[\begin{array}{l} G_1 : \text{an action, in case of } true \\ G_2 : \cdots \\ G_3 : return\_code := GOOD; \\ G_4 : \textbf{go to } X; \end{array}\right] \\ \textbf{else} \\ \quad \left[\begin{array}{l} B_1 : \cdots \\ B_2 : return\_code := BAD; \\ B_3 : \textbf{go to } X; \end{array}\right] \\ X : \end{array}\right]$$

We write:

$$\Box((at\_X \wedge return\_code = GOOD) \rightarrow ((at\_G_3 \vee at\_G_4)\, \mathcal{B}\, at\_G_2))),$$

Then we write:

$$\begin{array}{ll} \Box\,(at\_G_2 & \rightarrow\ postcondition) \\ \Box\,(at\_G_2 \wedge postcondition) & \rightarrow\ (postcondition\ \mathcal{W}\ at\_X \wedge postcondition) \end{array}$$

Combining them into the desired property:

$$\Box((at\_X \wedge return\_code = GOOD) \rightarrow postcondition)$$

If *postcondition* involves global (shared) variables, then for the third property to hold, it is necessary to establish that these variables are protected from interference by other threads.

We used the above rules, in the manner described in the examples, to verify non-trivial data transformations, some as complex to analyze as the sort algorithm. The verification relied heavily on the non-interference LTL formulae verified by SPIN . The combination of the two methods resulted in a core module of the product being successfully verified.

6 Conclusions and Further Research

In this paper, we have reported about the experience of Perfecto Technologies in the application of formal verification to their security server software product, and the lessons we learned from this experience.

Starting with model checking by SPIN, we succeeded to verify a certain portion of the TLD of the system. To do so, we have developed a compositional approach, specially geared to verification by SPIN. One of the conclusions we reached was that model checking alone cannot scale up to verify the detailed design, but may suffice for the verification of the top-level design, provided one uses compositionality and abstraction.

For verification of the detailed design, we finally settled on a combination of model checking for the control-intensive part and manual deductive verification for the data-intensive part. The two methods interact by the deductive verification benefiting from proofs of non-interference established by model checking.

The preference for manual verification was forced on us because none of the existing support tools for deductive verification provides a system description language fully adequate for modeling multi-threaded C++ programs. We indicate in Section 3 how such a system description language may be developed with minimal extensions. We call upon developers of tools for deductive verification to pay more attention to providing a convenient interface language if they wish to attract users from the C++ community.

Acknowledgements

The authors would like to thank Eran Reshef (Perfecto Technologies Ltd.) and Eilon Solan (Northwestern University, IL., USA) for their continuous help and advice, and Elad Shahar (The Weizmann Institute of Science) for his help with the SMV system.

References

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Prog. Lang. Sys.*, 15:73–132, 1993.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Sys.*, 17(3):507–534, 1995.

[BBC+95] N. Bjørner, I.A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report, Stanford University, 1995.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BK85] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In *Proc. of Seminar on Concurrency*, LNCS 197, 1985.

[CC95] P. Collete and A. Cau. Parallel composition of assumption-commitment specifications: A unifying approach for shared variables and distributed message passing concurrency. *Acta Informatica*, 1995.

[CGL96] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking. In *Model Checking, Abstraction and Composition*, volume 152 of *Nato ASI Series F*, pages 477–498. Springer-Verlag, 1996.

[CM81] K.M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Trans. Software Engin.*, 7(4):417–426, 1981.

[dR85] W.-P. de Roever. The quest for compositionality — a survey of assertion-based proof systems for concurrent programs, part i: Concurrency based on shared variables. In *The Role of Abstract Models in Computer Science*, pages 181–206. IFIP, North Holland, 1985.

[Hoa84] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Engelwood Cliffs, NJ, 1991.

[Jon83] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Trans. Prog. Lang. Sys.*, 5(4):596–619, 1983.

[Jon94] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. Prog. Lang. Sys.*, 16(2):259–303, 1994.

[KM95] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KP98] Y. Kesten and A. Pnueli. Deductive verification of fair discrete systems. Technical report, Weizmann Institute, 1998.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In ICALP'98 pages 1–16.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engin.*, 3:125–143, 1977.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proc. 9th ACM Symp. Princ. of Dist. Comp.*, pages 377–408, 1990.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[OSR93] S. Owre, N. Shankar, and J.M. Rushby. User guide for the PVS specification and verification system. SRI International, Menlo Park, CA, 1993.

[PJ91] P.K. Pandya and M. Joseph. P-A logic – a compositional proof system for distributed programs. *Dist. Comp.*, 5:37–54, 1991.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, sub-series F: Computer and System Science, pages 123–144. Springer-Verlag, 1985.

[XdRH97] Q.W. Xu, W.-P. de Roever, and J.-F. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[Zwi89] J. Zwiers. *Compositionality Concurrency and Partial Correctness*, volume 321 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1989.