Model Checking in Practice: An Analysis of the ACCESS.bus Protocol using SPIN

Bernard Boigelot and Patrice Godefroid

Proceedings of Formal Methods Europe'96, Oxford, March 1996. Lecture Notes in Computer Science, vol. 1051, pages 465-478, Springer-Verlag.

Model Checking in Practice: An Analysis of the ACCESS.bus™ Protocol using SPIN

Bernard Boigelot[1]* and Patrice Godefroid[2]**

[1] Université de Liège
Institut Montefiore, B28
B-4000 Liège Sart-Tilman, Belgium
boigelot@montefiore.ulg.ac.be

[2] AT&T Bell Laboratories
1000 E. Warrenville Road
Naperville, IL 60566, U.S.A.
god@research.att.com

Abstract. This paper presents a case study of the use of model checking for analyzing an industrial protocol, the ACCESS.bus™ protocol. Our analysis of this protocol was carried out using SPIN, an automated verification system which includes an implementation of model-checking algorithms. A model of the protocol was developed, and properties expressed by linear-time temporal-logic formulas were checked on this model. This analysis revealed subtle flaws in the design of the protocol. Developers who worked on implementations of ACCESS.bus™ were unaware of these flaws at a very late stage of their development process. We also present suggestions for solving the detected problems.

1 Introduction

State-space exploration techniques are increasingly being used for debugging and proving correct finite-state concurrent reactive systems (cf. [Rud87, Liu89, HK90, Hol91, DDHY92, FGM+92]). These techniques consist of exploring a global state graph, called the *state space*, representing the combined behavior of all concurrent components in the system. This is done by recursively exploring all successor states of all states encountered during the exploration, starting from a given initial state, by executing all enabled transitions in each state. Many different types of properties of a system can be checked by exploring its state space: deadlocks, dead code, unspecified receptions, buffer overruns, etc. Moreover, the range of properties that state-space exploration techniques can verify has been substantially broadened during the last decade thanks to the development of

* "Aspirant" (Research Assistant) for the National Fund for Scientific Research (Belgium). The work of this author was done in part while visiting AT&T Bell Laboratories.

** This work was carried out in part while this author was with the University of Liège.

model-checking methods for various temporal logics (e.g., [CES86, LP85, QS81, VW86]).

In this paper, we present an application of model checking for the analysis of the ACCESS.bus$^{\mathrm{TM}}$protocol. The ACCESS.bus$^{\mathrm{TM}}$ protocol is a serial communication protocol aimed at providing a simple, uniform, and inexpensive way to connect peripheral devices (such as keyboards, mice, modems, monitors, and printers) to a host computer. It has been developed and standardized by an industrial consortium of computer and peripheral manufacturers, referred to as the *ACCESS.bus*$^{\mathrm{TM}}$ *Industry Group* [ACC94]. At the time of this writing, implementations of the ACCESS.bus$^{\mathrm{TM}}$ protocol already exist, and are expected to be commercialized soon.

Our analysis of the correctness of the ACCESS.bus$^{\mathrm{TM}}$protocol was performed using the automated protocol verification system *SPIN* [Hol91]. SPIN checks properties of communication protocols, modeled in the *Promela* language, by exploring their state space. Promela is a nondeterministic guarded-command language for modeling systems of concurrent processes that can interact via shared variables and message channels. Interaction via message channels can be either synchronous (i.e., by rendez-vous) or asynchronous (buffered) with arbitrary (user-specified) buffer capacities, and arbitrary numbers of message parameters. Given a concurrent system modeled by a Promela program, SPIN can check for deadlocks, dead code, violations of user-specified assertions, and temporal properties expressed by linear-time temporal-logic formulas. When a violation of a property is detected, SPIN reports a scenario, i.e., a sequence of transitions, violating this property.

Our analysis of the ACCESS.bus$^{\mathrm{TM}}$protocol pointed out several ambiguities in the standardized document specifying the protocol. Moreover, it revealed subtle and potentially harmful flaws in the design of the protocol itself. Developers who worked on implementations of ACCESS.bus$^{\mathrm{TM}}$ were still unaware of these flaws at a very late stage of their development process.

This paper is organized as follows. In the next section, we present an overview of the ACCESS.bus$^{\mathrm{TM}}$ protocol. Next, we describe our model for this protocol, and discuss our assumptions. In Section 4, we specify two basic properties that the protocol has to satisfy. Then, we turn to the verification of these properties. For both of these properties, SPIN reported scenarios violating the property. These scenarios are presented in Sections 5 and 6. By analyzing these scenarios, causes of errors have been identified, and suggestions for solving the detected problems are presented.

2 ACCESS.bus$^{\mathrm{TM}}$ Protocol

The ACCESS.bus$^{\mathrm{TM}}$ protocol is a serial communication protocol. Its purpose is to provide a simple, uniform and inexpensive way to connect peripheral devices to a host computer. The analysis presented in this paper is based on release 2.2 of the protocol specifications [ACC94].

An important feature of ACCESS.bus$^{\text{TM}}$ is that it supports dynamic reconfiguration, which means that devices can be connected to the bus while the system is operating, and can become operational without the system being rebooted. The overall structure of ACCESS.bus$^{\text{TM}}$ is illustrated in Figure 1. The protocol is composed of a hardware layer based on the I^{2}C protocol developed by Philips, and of two software layers referred to as "Base Protocol" and "Device Drivers". Device Drivers are controlled by user applications running on the host.

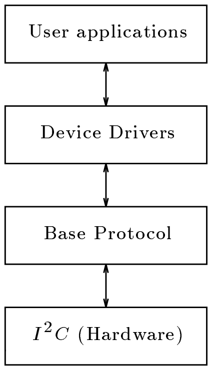

Fig. 1. Structure of ACCESS.bus$^{\text{TM}}$.

The I^{2}C protocol is a serial protocol that is used for interconnecting IC's inside electronic appliances such as TV's and VCR's. It uses a *bus* composed of two wires, *serial data* (SDA) and *serial clock* (SCL), which connects the host and the devices together. Each *component* (i.e., device or host) has an 8-bit *I^2C address* which is not necessarily unique and may change over time. When a component is plugged in, its address becomes the *default I^2C address*. Information is transmitted on the I^{2}C bus by means of *messages* composed of an *address part* and a *data part*. Since the bus is synchronous, there is no propagation delay. Any I^{2}C component may try to send a message at any time over the bus. Although a *same* message can be sent simultaneously by several components, an *arbitration mechanism* ensures that two *different* messages are never sent at the same time. This mechanism is deterministic: whenever two or more components attempt to simultaneously send different messages over the I^{2}C layer, the conflict is resolved in favor of the same message. A transmitted message is received by a component if and only if the address of the component matches the address part of the message, and the component is not simultaneously transmitting the same message over the bus. A message can thus be *lost* if its recipient is simultaneously transmitting the same message, or if there is no recipient. Each time a message is sent over the I^{2}C layer, its sender receives a *Positive Acknowledgment* from the I^{2}C layer if the message is received by another component, and a *Negative Acknowledgment* otherwise.

The Base Protocol aims at ensuring that every device will always be recog-

Message Types	**Purpose**
Reset()	Force a device to its power-up state and to the default I^2C address. This message is sent by the host on power-up to all the I^2C addresses. A device also sends this message to its address right after being assigned a new address.
Attention()	Inform the host that a device has finished its power-up/reset test and needs to be configured.
IdentificationRequest()	Ask a device for its *identification string*, which is a sequence of bytes describing the hardware composing the device. This message is issued by the host after reception of an *Attention* message from a device.
IdentificationReply(Id)	Reply to *IdentificationRequest* with the identification string *Id* of the device.
AssignAddress(Id, Addr)	Ask all the devices with a matching identification string *Id* to turn their address into *Addr*.
PresenceCheck()	Check if a device is present on the bus at a specific address (specified in the address part of the message). This message is sent by the host at regular intervals of time in order to detect new and missing devices.
CapabilitiesRequest(Offset)	Ask a device to send a fragment (specified by *Offset*) of its *capabilities string*, which is a sequence of bytes describing the functional characteristics of the device.
CapabilitiesReply(Offset, Data)	Reply to *CapabilitiesRequest* with a fragment of the capabilities string of the device.
EnableApplicationReport()	Enable or disable a device to send application reports, that is, device-dependent functional information, to the host.
ApplicationReport(Data)	Send device-dependent functional information.

Fig. 2. Base Protocol Message Types.

nized by the host within a finite amount of time after being plugged in, that it will be assigned a unique I^2C address, and that its Device Drivers will be able to send and receive device-dependent functional data (such as mouse moves). The Base Protocol defines a set of message types that can be sent over the I^2C layer. These message types are listed in Figure 2. When a device is plugged in, it sends an *Attention* message to the host, which should reply with an *Identification-Request* message, which should itself be replied to with an *IdentificationReply* message from the device. Then, the host should send an *AssignAddress* message containing a new I^2C address for the device. When processing this message, the device updates its address, and sends a *Reset* message to this address.

3 Design of the Model

Our analysis of ACCESS.bus^{TM} focused on the power-up/reset and identification phases, that is, the part of the Base Protocol dealing with *Reset, Attention, IdentificationRequest, IdentificationReply, AssignAddress* and *PresenceCheck* messages. We made the following assumptions in order to resolve ambiguities in the specification document.[3]

- At the Base Protocol layer, every message received from the I^2C layer is stored in a bounded fifo buffer while waiting to be processed. If the buffer is full, new incoming messages are lost.
- The processing of a *Reset* message by a Base Protocol entity empties its associated fifo buffer.

Moreover, the following features of ACCESS.bus^{TM} were not modeled:

- the deterministic nature of the I^2C arbitration mechanism,
- the timing constraints defined in the specification document, and
- the possible corruption of messages sent over the I^2C bus.

Consequently, if a message is sent over the I^2C layer at the same time by one or more components, it is always correctly received exactly once by every component with a matching address that does not belong to the set of the senders.

The overall structure of the model is shown in Figure 3. Each component is modeled by two processes: a *microcontroller* and an *Upper Base Protocol* (UBP). The I^2C bus is modeled by shared variables, since message broadcasting is not a basic communication primitive in Promela. A set of semaphores is used to control the access to the bus.

Each microcontroller continually listens to the bus, grabs messages destined to its corresponding UBP, and appends them to a bounded fifo buffer, which is a basic Promela data type. Each UBP takes messages from its associated fifo buffer, and processes them according to the protocol rules (cf. Figure 2). It can send messages directly (without queuing) over the I^2C layer. Moreover, it can nondeterministically switch between two modes, plugged and unplugged,

[3] These assumptions match those made by the developers we had contacts with.

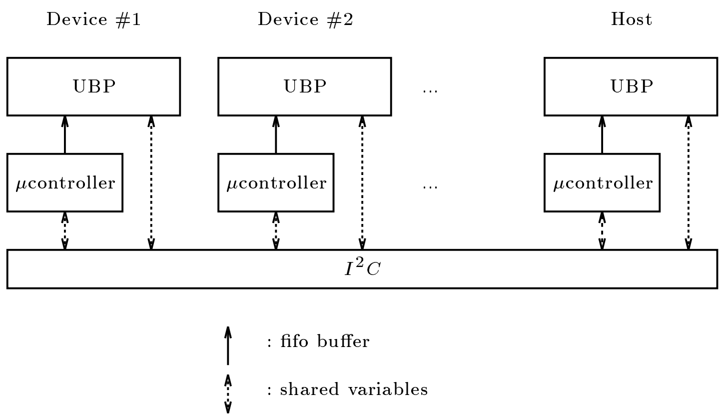

Fig. 3. Structure of the model.

in order to simulate repeated pluggings and unpluggings of the corresponding component.

Initially, all devices are assumed to be unplugged. To keep the state space of the Promela model as small as possible, the maximum size of each fifo buffer was set to two elements, and the number of devices was limited to two. The complete Promela model contains about 200 lines of code.

4 Properties

The specification document [ACC94] does not contain a precise and complete description of the service provided by the Base Protocol to the Device Drivers. Two basic properties that have to be satisfied by the Base Protocol were extracted from the document.

Property 1. A device d_i is said to be *operational* when it has an I^{2}C address $addr(d_i)$ different from the default I^{2}C address, and it has sent a *Reset* message to the address $addr(d_i)$. At any time, all devices that are operational must have different I^{2}C addresses.

This property can be formalized by using linear-time propositional temporal logic [MP92]. Linear-time temporal logic can be used for specifying properties of infinite sequences of states. Propositions in the logic correspond to boolean conditions on variables and process states of the program. Formulas are constructed over propositions using the classical boolean connectives ($\neg$, $\vee$, ...) and the temporal operators $\Box$ (always), $\Diamond$ (eventually), and $\bigcirc$ (next). Formulas are interpreted on *infinite* sequences $s_0 s_1 s_2 \ldots$ of states: given a particular infinite sequence of states, the formula is either satisfied or falsified by this sequence. Informally, one has:

- $\Box p$ holds in state s_i if p holds in s_i and in all successor states of s_i in the sequence on which the formula is interpreted;
- $\Diamond p$ holds in s_i if p holds in some successor state of s_i or in s_i itself;
- $\bigcirc p$ holds in s_i if p holds in the next state of the sequence.

We refer the reader to [MP92, Eme90] for a detailed presentation of the syntax and the semantics of linear-time temporal logic.

For a pair of devices d_1 and d_2, Property 1 can be formalized by the following linear-time temporal-logic formula:

$$\Box((oper(d_1) \wedge oper(d_2)) \Rightarrow (addr(d_1) \neq addr(d_2))),$$

where $oper(d_i)$ is true if device d_i is operational, and $(addr(d_1) \neq addr(d_2))$ is true if devices d_1 and d_2 have different $\mathrm{I}^2\mathrm{C}$ addresses ($\Rightarrow$ denotes logical implication).

The second property of the Base Protocol we consider is the following.

Property 2. Whenever a device is plugged in, it will eventually become operational, provided that it remains plugged.

This property can be formalized by the following linear-time temporal-logic formula:

$$\Box(plugged(d_1) \Rightarrow \Diamond(oper(d_1) \vee \neg plugged(d_1))),$$

where $plugged(d_1)$ is true if device d_1 is plugged.

Given the finite state space A_G of a system and a linear-time temporal-logic formula f, checking that all infinite sequences of states defined by transitions in A_G satisfy f is known as the *model-checking problem*. Various techniques have been proposed for solving this problem [LP85, VW86, CVWY90, GH93, GPVW95]. SPIN includes an implementation of the algorithms presented in [GH93] and [GPVW95], which are based on a depth-first search in the state space of the system (see [GH93] for details). When SPIN detects a sequence of states that violates the property to be checked, it stops its search, and exhibits this scenario (formed by all states and transitions currently stored in the depth-first-search "stack") to the user.

Let us now turn to the results obtained by SPIN with the Promela model described in the previous section and the two properties defined above.

5 Verification of Property 1

5.1 First Flaw

After a few seconds of computation, SPIN detected that the first property was not satisfied.

Figure 4 depicts a first sequence of transitions leading to a state where two devices are operational while having been assigned the same $\mathrm{I}^2\mathrm{C}$ address. In this diagram, a thin vertical *time line* is associated with each plugged component. Time increases from the top to the bottom of the time lines. The sending of a

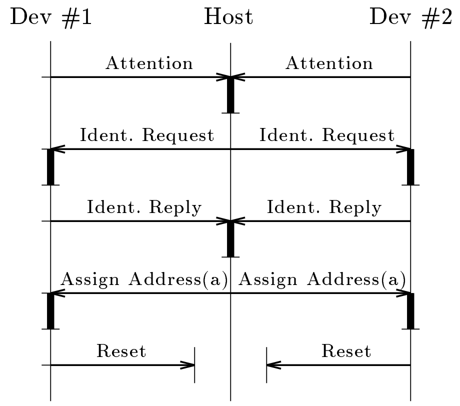

Fig. 4. First flaw.

message through the I^2C layer is represented by an horizontal arrow drawn from the time line of the sender to the time line of the receiver. (Indeed, there is no delay between the sending and the reception of a message.) The head and the tail of the arrow correspond to the exact moment when the message starts to be transmitted on the I^2C bus. If a message is lost (i.e., is not received by any component), the corresponding arrow does not reach any time line. A message is lost when its recipient does not exist, when its recipient is sending the same message over the bus, or when the fifo buffer of the recipient is full. Thick vertical lines represent the delay between the moment when the message is appended to the input buffer of its recipient and the moment when the message is actually processed by its recipient.

In the scenario of Figure 4, two devices with the same identification string are plugged in at the same time. If both devices send and receive simultaneously all the messages shown in Figure 4, it is impossible for the host to distinguish them. Moreover, the self-addressed *Reset* message is not received by any device if they both send this message at the same time.

This problem is a direct consequence of the properties of I^2C, and is not surprising. However, it is worth noticing that having two devices sending the same message at exactly the same time is not an unlikely event. Two devices that wait for sending a message will synchronize on the message frame currently being transmitted on the I^2C bus, and will both start trying to transmit their message at the exact end of this frame.

5.2 Second Flaw

A more complex sequence of events violating Property 1 is given in Figure 5. As in the previous scenario, two devices are plugged in and have the same identification string. The first device finishes its internal initialization process, and sends an *Attention* to the host at time t_0. At time t_1, the host assigns the address a to this device by sending an *AssignAddress* message, which is stored in the input

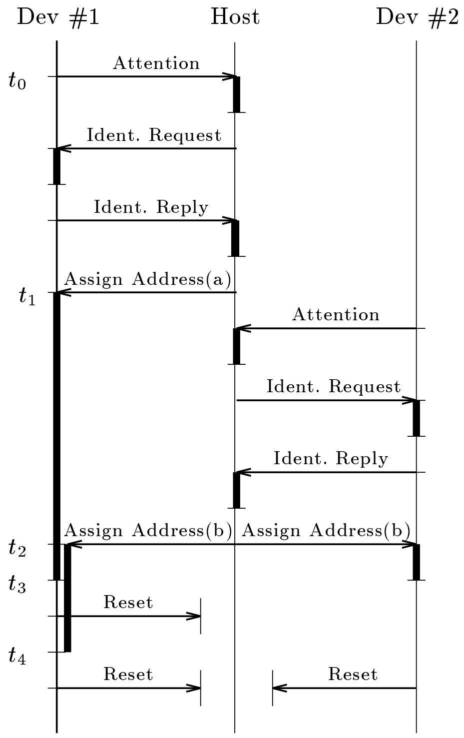

Fig. 5. Second flaw.

fifo buffer of the device. Then the second device sends an *Attention* to the host, and enters its identification phase. When the host assigns the address b to the second device at time t_2, the *AssignAddress* message is also received by the first device, since its address is still the default address at that time, because the request to change its address to a is still waiting in its buffer and has not been processed yet. When the first device finally processes its incoming messages, it changes its address to a at time t_3, sends a self addressed *Reset*, and sets its address to b at time t_4. The self addressed *Reset* messages are then sent simultaneously by both devices, and are thus lost, allowing the two devices to become operational with the same address b.

This scenario reveals another problem in the protocol: here, the erroneous situation results not only from losing two *Reset* messages, but also from delaying an *AssignAddress* in a fifo buffer.

5.3 Third Flaw

When observing the first two flaws, one could wonder if Property 1 is violated only when two devices may share the same identification string. SPIN can easily show that this is not the case. A scenario resulting in the assignment of the same

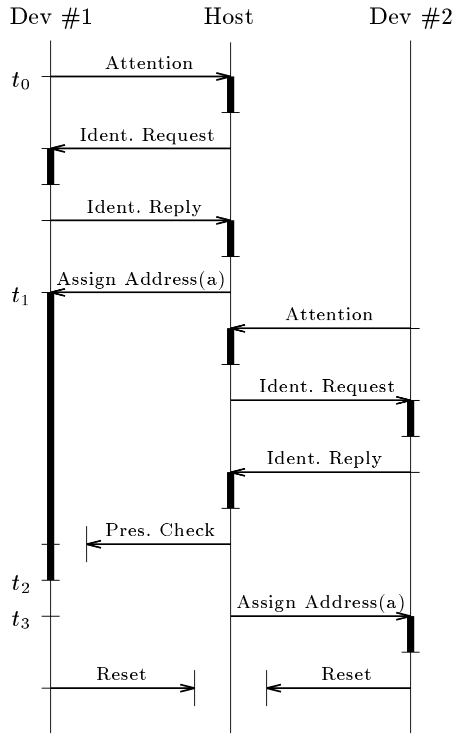

Fig. 6. Third flaw.

address to two devices with different identification strings is given in Figure 6. As in the previous scenario, the first device starts its identification phase at time t_0, and the processing of the *AssignAddress* message received from the host at time t_1 is delayed until time t_2. In the meantime, the second device sends an *Attention* to the host, and proceeds with its identification phase. On reception of an *IdentificationReply* from the second device, the host issues a *PresenceCheck* aimed at checking if the first device is still plugged. The first device does not receive this *PresenceCheck* message, since it is still using the default bus address. Therefore, the host receives a *Negative Acknowledgment* from the I^2C layer. It concludes that the first device is not present anymore, and that address a is available. It then assigns address a to the second device at time t_3. Again, if the self-addressed *Reset* messages are sent simultaneously by the two devices, they are lost, and both devices become operational with the same address a, thus violating the first property.

5.4 Suggestions

The three scenarios presented in Figures 4, 5, and 6 reveal the existence of three causes of errors for Property 1 in the Base Protocol:

– a *Reset* message is lost,
– two devices share the same identification number, and
– the processing of an *AssignAddress* is delayed.

For eliminating these causes of errors, we suggest the following modifications to the Base Protocol.

In order to avoid losing *Reset* messages, these messages should not be appended to the input buffers of their recipients, but should rather be processed immediately upon reception, for instance by issuing hardware interrupts to notify immediately the corresponding UBP of the arrival of such a message. In such a way, *Reset* messages will not be lost when the input buffers of their recipients are full. Moreover, the loss of a *Reset* message due to the fact that it is simultaneously sent by two (or more) components can be avoided by adding a unique firmware number of the sender to each *Reset* message frame. If this radical solution is too expensive to be implemented, adding a random number (e.g., the value of an internal clock) to each *Reset* message frame will strongly reduce the probability of losing a *Reset* message because of simultaneous transmissions of it. This probability can be further reduced by waiting for a random amount of time before trying to send a *Reset* message.

Concerning the second cause, preventing identical identification strings can be done by using a unique firmware number in the identification string of each device.

Finally, the problems resulting from delaying an *AssignAddress* message can be avoided by the two following modifications. First, *AssignAddress* messages should be processed immediately upon reception, for instance by using hardware interrupts as indicated above. Second, whenever a component receives an *AssignAddress* message requesting a modification of its current I^2C address, its fifo buffer should be emptied.

Once we have modified our Promela model by following the above suggestions, SPIN proved in about 30 minutes of computation on a SPARC20 workstation with 256 Megabytes of RAM that Property 1 was satisfied by all possible executions of the model. Note that the correctness proof of our model does not guarantee that the modifications suggested above are sufficient for avoiding the reported problems in practice.

It is worth noticing that the timing constraints defined in the specification document do not prevent any of the three scenarios discussed in this section from occurring. Indeed, each of these scenarios can easily be annotated with timestamps satisfying these timing constraints.

6 Verification of Property 2

6.1 Fourth Flaw

SPIN quickly found scenarios violating Property 2 as well. Indeed, two or more devices can hold the I^2C bus for an unbounded amount of time, and thus prevent other components from sending messages. The timing constraints described in the protocol specification help to prevent such situations, but it is easy to show that these constraints are not sufficient to completely solve the problem. Moreover, the deterministic nature of the I^2C arbitration mechanism (which we did not model) does not help to solve this problem. Indeed, if two devices alternatively send *ApplicationReport* messages over the bus, it can be deduced from the arbitration rules that, if their addresses have high-priority values (i.e., 02 and 03), it is impossible for a third device with an address of lower priority (i.e., FE) to be granted the bus at any time. In this scenario, the third device will never be able to send an *Attention* message to the host to signal its presence in the system, and hence will never become operational.

6.2 Suggestions

The probability of occurrence of such scenarios can be reduced by modifying the structure of the message frames in order to give a higher priority to protocol messages, as opposed to application reports, with respect to the I^2C arbitration mechanism. One could use for this purpose the least significant bit of the first byte of the frame (0 for protocol message frames, 1 for application reports). The protocol specification also includes an optional *Device Bandwidth Management* system, which could help avoiding the problem.

7 Conclusions

We have presented the main stages and the results of an analysis of an industrial protocol, the ACCESS.bus^{TM} protocol. The analysis of this protocol was performed using SPIN, an automated protocol verification system including state-space exploration and model-checking algorithms. Our analysis revealed subtle flaws in the design of this protocol, which were not found by simulating or testing the existing prototype implementations. We have also presented suggestions for solving the detected problems. During this work, SPIN repeatedly proved to be a powerful and efficient verification tool.

Model checking is an effective and simple method for verifying that a concurrent reactive system satisfies a temporal logic formula. It makes it possible to reason about programs without having the burden of carrying out correctness proofs by hand. Indeed, model checking is *fully automatic*: no intervention of the user is required. This is a crucial feature for a verification technique to be used in industry, since products are often (read always) developed under time pressure, and therefore verification steps that would be too time consuming are likely to be skipped.

Although model checking is fully automatic, applying model checking for the analysis of communication protocols is not yet a systematic activity. The ability of quickly modeling a system at the "right" level of abstraction requires training, experience, and some knowledge of how model-checkers work: oversimplifying the model of the system should be avoided in order to be able to detect potential problems in the actual system, while abstracting enough irrelevant details is needed in order to keep automatic verification computationally tractable. Moreover, ingenuity and tenacity are often necessary for expressing interesting properties (i.e., those that might reveal significant errors) and for filtering error traces when looking for plausible scenarios (i.e, those that may occur in a realistic environment). In summary, verification is and remains a discipline in itself, even with the help of powerful verification tools such as model-checkers. Therefore, we believe that the most promising and pragmatic way for introducing formal verification in existing development processes is by forming groups of "validation engineers" who are specially trained for this task.

Another analysis of the Base Protocol can be found in [Hoo95]. It was carried out by using an assertional method with the help of the interactive proof checker included in the verification system PVS [ORS92]. Hooman proved manually that the Base Protocol satisfies Property 1 and 2 provided that all the devices have a different identification string, that messages between base-protocol components are not buffered, and that whenever a component wants to transmit a message over the I^2C layer, this message is transmitted within a bounded amount of time. If one of these (strong) assumptions is not satisfied, no information about the correctness of the protocol is provided. In contrast, our analysis was based on a more detailed model, i.e., on weaker assumptions, and produced counter-examples violating Property 1 and 2. This enabled us to precisely identify the causes of these errors, and to suggest implementable solutions for these problems. Finally, all counter-examples mentioned above and the proof of correctness of our modified model were produced automatically by SPIN.

8 Acknowledgments

We wish to thank Didier Pirottin, who contributed to the results presented in this paper. We are also grateful to Ron Koymans (Philips Research) for challenging us to analyze the ACCESS.bus^{TM} protocol and for fruitful discussions, and to Mark Staskauskas for helpful comments on a preliminary version of this paper.

References

[ACC94] ACCESS.bus Industry Group. Access.bus specifications, version 2.2. 370 Altair Way, Suite 215, Sunnyvale, California 94086, USA, 1994.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, Rutgers, June 1990.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, Cambridge, MA, October 1992. IEEE Computer Society.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier/MIT Press, Amsterdam/Cambridge, 1990.

[FGM+92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. of the 14th International Conference on Software Engineering ICSE'14*, Melbourne, Australia, May 1992. ACM.

[GH93] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proc. 13th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 109–124, Liège, May 1993. North-Holland.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

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

[Hoo95] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. To appear in the Proceedings of Foundations of Software Technology and Theoretical Computer Science, December 1995.

[Liu89] M.T. Liu. Protocol engineering. *Advances in Computing*, 29:79–195, 1989.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. 11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rud87] H. Rudin. Network protocols and tools to help produce them. *Annual Review of Computer Science*, 2:291–316, 1987.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.