Verifying Hardware Components with JACK*

R. De Nicola[1], A. Fantechi[2], S. Gnesi[3], S. Larosa[3], G. Ristori[4]

[1] Dip. di Scienze dell'Informazione, Univ. di Roma "La Sapienza", Italy
[2] Dip. di Ingegneria dell'Informazione, Univ. di Pisa, Italy
[3] Istituto di Elaborazione dell'Informazione, C.N.R. Pisa, Italy
[4] Dip. di Informatica, Univ. di Pisa, Italy

Abstract. JACK (the acronym for *Just Another Concurrency Kit*) is a workbench integrating a set of verification tools for concurrent system specifications, supported by a graphical interface offering facilities to use these tools separately or in combination. The environment offers several functionalities to support the design, analysis and verification of systems specified using process algebras. In this paper we use JACK to formally specify the hardware components of a buffer system. Then we verify, by using the checking capabilities of JACK, the correctness of the specification with respect to some safety requirements, expressed in the action based temporal logic ACTL.

1 Introduction

Process algebras [18, 15] are generally recognized as a convenient tool for describing reactive systems (i.e. those systems that do not work in isolation but perform their task by interacting with others). They provide a compact linear presentation and proof methods to support verification of systems properties. The semantic models of process algebra terms are essentially finite or infinite state automata, that have often been used for specifying hardware components. Within the process algebra framework, verification of a given system specification against an implementation is usually performed by studying the behavioural relationships (i.e. equivalence or preorder) between the transition systems associated to the different descriptions of the system. Modal and temporal logics have also been proposed [10, 11, 14] as alternatives to the equivalence (preorder) based approach. Indeed logics permit more abstract specifications, since they can be used for describing systems properties rather than systems behaviours. Properties of a given system are then verified by checking whether the automaton associated to the process algebra term, describing the system, is a model for the formula expressing the desired property. Moreover, automatic tools have been devised to support both verification of behavioural equivalence of systems (see e.g. [2, 4]) and model checking of system properties (see e.g. [2, 3, 4, 7]).

* The work described was partially performed within the LAMBRUSCO project supported by C.N.R., under the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, and within the Progetto Coordinato C.N.R. Specifica ad Alto Livello e Verifica Formale di Sistemi Digitali.

In this paper we describe the architecture of the integrated verification environment JACK (Just Another Concurrency Kit) [2], that provides both behavioural and logical verification on concurrent systems defined by automata. We show how it can be used to support specification and verification of a buffer circuit. The goal of JACK is to provide a general environment that offers a series of functionalities. The JACK system has grown out of a set of tools developed separately that have been successively integrated. It covers many aspects of the formal system development process, including the formalization of requirements [12], rewriting techniques [8], behavioural equivalence proofs [19, 16], graph transformations [19], logical verifications [5, 7]. In this paper we show on a case study how JACK supports a verification methodology which takes advantage of different descriptions of a hardware component: a graphical description is used to support the behavioural specification of the system; a temporal logic description is used to express its abstract properties.

2 Background

We introduce now the action based version of CTL [11], called ACTL, defined in [9]; it is a branching time logic suitable for expressing properties of reactive systems defined by means of Labelled Transition Systems. We start introducing Labelled Transition Systems, on which reactive systems are modelled and ACTL formulae are interpreted.

Definition1 Labelled Transition System. A Labelled Transition System (LTS in short) is a 4-tuple $\mathcal{A} = (Q, q_0, Act \cup \{\tau\}, R)$, where:

1. Q is a finite set of states. We let $q, r, s, \ldots$ range over states;
2. q_0 is the initial state;
3. Act is a finite set of observable actions and τ is the unobservable action. We let $a, b, \ldots$ range over Act, and $\alpha, \beta, \ldots$ range over $Act \cup \{\tau\}$;
4. $R \subseteq Q \times (Act \cup \{\tau\}) \times Q$ is the transition relation.

Definition2 Paths. Let $\mathcal{A} = (Q, q_0, Act \cup \{\tau\}, R)$ be a LTS. Then σ is a path from $r_0 \in Q$ if either $\sigma = r_0$ (the empty path from r_0) or σ is a (possibly infinite) sequence $(r_0, \alpha_1, r_1)(r_1, \alpha_2, r_2)\ldots$ such that $(r_i, \alpha_{i+1}, r_{i+1}) \in R$ for each $i \geq 0$.

ACTL is a temporal logic of state formulae (denoted by ϕ), in which a path quantifier prefixes an arbitrary path formula (denoted by γ). ACTL models are all *total*, i.e. each of their paths has *infinite* length; this is not a limitation, since each finite length path can be transformed into an infinite one by adding an edge from its last state into itself. The definition of ACTL includes the logic for the definition of action formulae (denoted by χ). Some of the ACTL operators we used in the case study and their informal semantics are reported in Table 1, while the formal semantics of the full set of the ACTL operators is described in [7, 9].

Several modalities can be defined starting from the basic ones. We will write $< \alpha > \phi$ for $E[true\{true\}U\{\alpha\}\phi]$ and $[\chi]\phi$ for $\sim< \chi >\sim \phi$. Moreover, by using

the until operator it is possible to define the derived modality $AG\phi$, that means *from now on ϕ is always true.*

Finally, we note that the ACTL logic allows one to simply express safety and liveness properties in terms of the actions a system can perform. Safety properties permit to say that nothing bad does happen. Liveness properties state that something good eventually happens.

Table 1. ACTL operators

Action formulae	
$\chi ::= true$	"any action"
$false$	"no action"
α	"α action"
$\neg\chi$	"not χ"
$\chi \mid \chi'$	"χ or χ'"
State formulae	
$\phi ::= T$	"any behaviour"
F	"no behaviour"
$\sim \phi$	" not ϕ"
$\phi \& \phi'$	"ϕ and ϕ'"
$E\gamma$	"there exists a path in which γ"
$A\gamma$	"for all paths γ"
Path formulae	
$\gamma ::= [\phi\{\chi\}U\{\chi'\}\phi']$	"ϕ is true for the states of the path until a state that satisfies ϕ' is reached by executing an action satisfying χ'. Before it, only actions satisfying χ or τ can be executed."
$X\{\tau\}\phi$	"the next state of the path satisfies ϕ and is reached by executing a τ action"
$X\{\chi\}\phi$	"the next state of the path satisfies ϕ and is reached by executing an action satisfying χ"

3 The JACK Environment

The idea behind the JACK environment was to combine different specification and verification tools, independently developed at different sites.

A first experiment in building verification tools, starting from existing ones, is described in [7]. Following this attempt, we have developed an environment

(whose structure is shown in Fig. 1) that exploits the FC2 format [17] for automata. We had the objectives to provide an environment in which a user can choose between several verification tools by a simple, user–friendly graphic interface and to create a general system for managing any tool that has an input or output based on FC2 format files. Such tools can be easily added to the JACK system, thus extending its potentiality.

Some of the tools within JACK allow a process specification to be built. This can be done both by entering a specification in a textual form and by offering sophisticated graphical procedures to build a specification as an automaton.

Other tools provide different verification strategies ranging from behavioural equivalence proofs to logical verifications by model checking algorithms. The access to both specification and verification tools is realized by means of a user-friendly interface (Fig. 2). Below, we introduce the JACK tools that are exploited for our case study.

• NL2ACTL is a specification tool. It provides a prototype translator from Natural Language expressions to ACTL formulae [12]. NL2ACTL has a friendly interface and makes easier the expression of system properties in the logic. Actually, NL2ACTL deals with sentences describing occurrence of actions performed by systems. A precise semantic meaning in terms of ACTL formulae is associated with each sentence. If the sentence does not have an ambiguous interpretation, an immediate ACTL translation is provided; otherwise, an interactive dialog with the user is estabilished to solve ambiguities.

• ATG is a graphic specification tool [19] for the design of parallel and communicating processes that provides functionalities for a compositional development of a specification. Process construction starts by drawing the automata that represent single sequential processes. Processes surrounded by boxes are said to be *networks* and boxes are used to hide details of low–level specification and to represent parallel composition. If two networks are drawn at the same level, they can synchronize via the signals they emit. A network could simply be an empty box: it is sufficient to specify its external synchronization signals; this permits a top–down approach in the ATG specification process. ATG can translate a graphic specification into a custom format, the FC2 format, or Postscript.

• FC2LINK is a linker for FC2 files; it inputs a set of files (that describe the components of a system) and a network description file (that describes how such components interact and synchronize together) and outputs a single FC2 network (that describes the global behaviour of the system).

• HOGGAR is a tool that offers minimization procedures (based on bisimulation) for systems described as a single transition system, or networks of transition systems. The algorithms are based on a symbolic representation of global transition systems by means of a *Binary Decision Diagram* (BDD) and permits the analysis of very large systems with a reasonable cost in terms of time and space.

• AMC, the model checker for ACTL logic formulae [7, 13] permits checking validity of ACTL formulae over an LTS in linear time on the size of the model. Whenever an ACTL formula φ does not hold, the model checker exhibits also a path of the LTS given in input, that falsifies φ, (called *counterexample*) and provides useful information on how to modify the LTS to satisfy the formula φ.

3.1 Other Tools

There are various other tools within JACK that we did not use in our case study. MAUTO [19], a tool for the specification and the verification of concurrent systems described by process algebra terms such as MEIJE and CCS [18]; PMC [5], a parallel model checker for CTL and (via translations) for ACTL. NSS [6], a tool providing a verification methodology to check ACTL properties on full CCS terms, i.e. also those corresponding to infinite automata. CRLAB [8], a system based on rewriting and on proof strategies for algebraic terms. PISATOOL [16], a verification tool for checking equivalence of CCS terms that is parametric with respect to the granularity of the observations chosen or defined by the user.

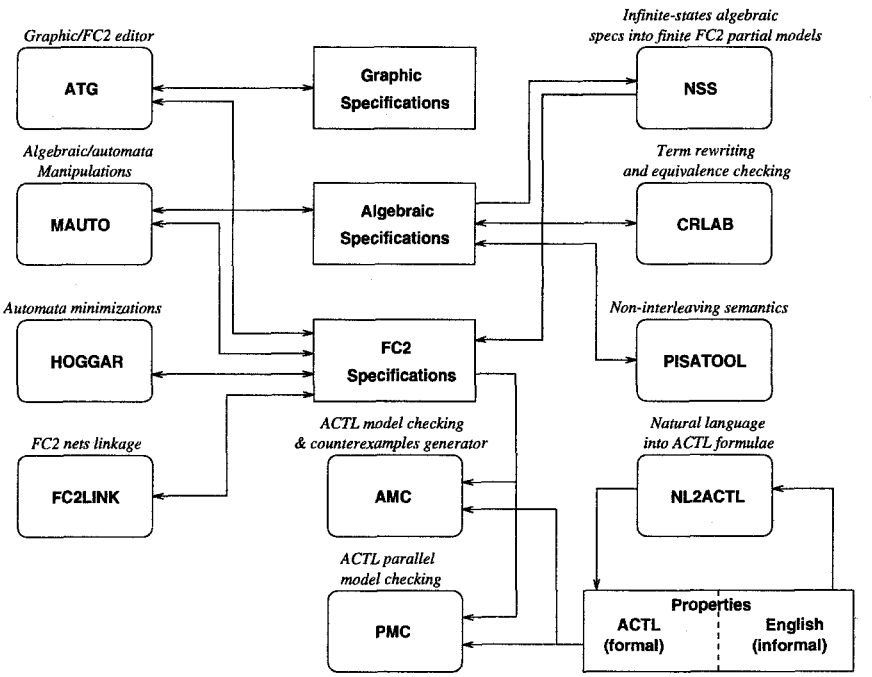

Fig. 1. A general view of the JACK toolset.

4 Case Study: a Hardware Buffer

In this section we study the specification and verification of a hardware buffer and some other related units. First we will present the informal requirements specification of the units composing the system, then we will proceed using JACK as follows:

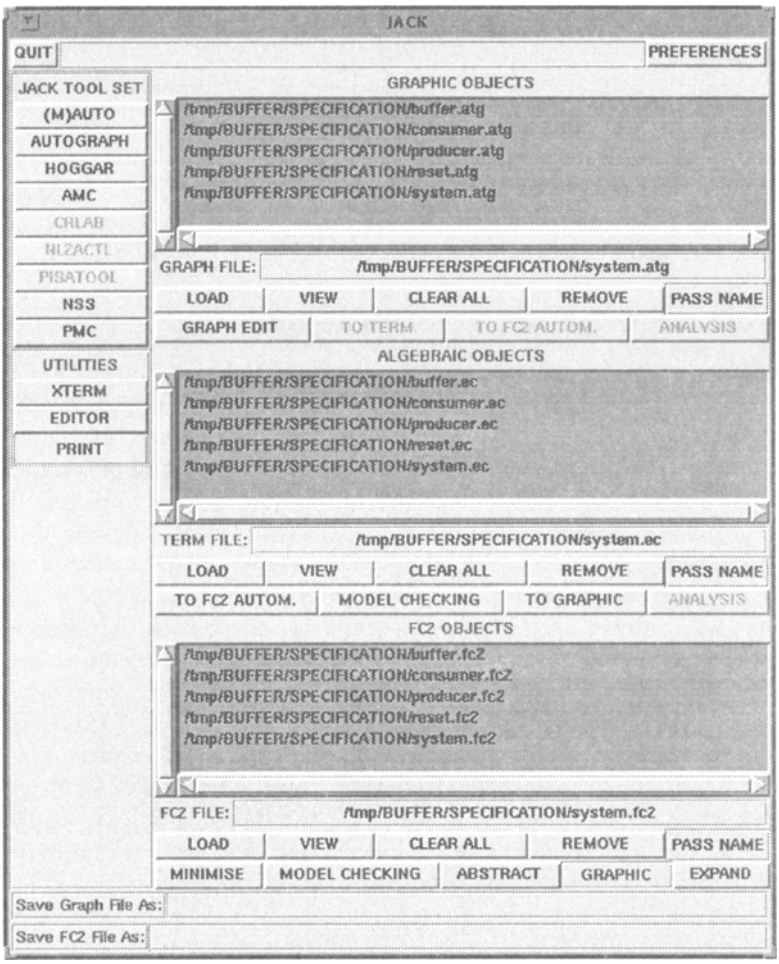

Fig. 2. The JACK interface

1. We will use ATG for providing the formal specification of the buffer and of the other units. Each unit will have a separate FC2 specification file, automatically generated by ATG from the graphic specification. Then, by means of FC2LINK we will get a single FC2 model of our system.
2. By means of HOGGAR we will perform an automatic minimization of the number of states of the buffer global system, getting an automaton that is equivalent to the original one with respect to the weak bisimulation.
3. We will use NL2ACTL to formalize the properties our system has to meet; this is useful to avoid some of the errors that are often made when one tries to go from natural language propositions to ACTL formulae.
4. Finally, we will use AMC providing it with the global automaton and a list of formulae, to check whether the properties represented by the formulae are

satisfied by our specification or not. In case an error forbids the satisfiability of a logical requirement, we have to face the problem of discovering *why* the checked formula is false. Such a task will be accomplished by the facilities of AMC in a semi–automatic way. Once the error in the specification has been recognized, a correct automaton can be defined, going back to phase 1.

To better exploit JACK functionalities we will introduce an error on the formal specification of our buffer system, that will be discovered by using the model checker AMC.

4.1 Hardware Requirements Specification

The hardware units that we are going to describe are a two positions *FIFO buffer*, a *producer*, that writes data in the buffer and a *consumer*, that reads data from the buffer. Only data instability is considered as a relevant event (*datain*). The level signals *wr_req, rd_req* and *end_rd* are raised by either the consumer or the producer, whilst *wr_ack, rd_ack, end_wr, bf_full* and *bf_empty* are raised by the buffer. For each of these signals, we consider their transition from 0 to 1 and from 1 to 0 as relevant event. The three units mentioned above can be reset by the asynchronous signal *rst_req*, that is an external request.

When the producer has to write a datum into the buffer, it waits for such a datum to become stable, then forces a transition from 0 to 1 of the *wr_req* signal. When the buffer is ready to serve the request, it raises the signal *wr_ack* and then, if the buffer is empty, the *bf_empty* signal goes to 0. After that *wr_ack* has gone to 1, the write operation can start. When the writing has been completed, the buffer should make a transition from 0 to 1 on the signal *end_wr* to signal the end of the write operation to the producer; if the buffer is full, the *bf_full* signal goes to 1. In the end, *wr_ack* returns to 0. Notice that the data are required to be stable during the time interval between the service request and the end of the write operation.

When the consumer has to read a datum from the buffer, it forces a transition from 0 to 1 on *rd_req*. When the buffer is ready to serve this request, it acknowledges the consumer via a transition from 0 to 1 on *rd_ack*. The end of the read operation is signaled by the consumer with a transition from 0 to 1 on *end_rd*; then the *bf_empty* signal goes to 1 if the buffer is empty and *rd_ack* goes to 0. Depending on the state of the buffer before the reading, a transition from 1 to 0 of the *bf_full* signal is done.

Notice that the read and the write operations are executed sequentially; hence, at most one operation can be issued at a given time.

The reset operation is requested by the *rst_req* asynchronous signal; its effect is to empty the buffer and stop the execution of its current operation. In particular, *wr_ack, rd_ack, end_wr, bf_full* go to 0 and *bf_empty* goes to 1.

4.2 Phase 1: Formal Specification with ATG

The first phase in the specification of the system consists of building the automata describing the components of the system. The actions beginning with a

"!" prefix are *outputs* of a component automaton, while the actions that begin with a "?" prefix are *inputs*. The initial states of the automata are the double–circled ones.

In Fig. 3, Fig. 4 and Fig. 5 the automata describing the consumer, the producer and the buffer modules are shown. Notice that three states of the buffer automaton have been labelled by *buffer_0*, *buffer_1* and *buffer_2* respectively, to emphasize how many positions of the buffer are occupied.

Moreover, to specify the reset operation requested by the *rst_req* signal, a new automaton has been created (Fig. 6) that describes a reset event handler. Then suitable synchronization actions have been added to the automata describing the other components of the system, to realize the communication between them and the reset automaton. In its initial state, the reset automaton waits first for a *?rst_req* external request, then for the end–of–reset signal *?end_rst* from the buffer and finally sends the *!on_{prod,con}* signals to restart the producer and the consumer.

To build the automaton corresponding to the whole system, we have to link together all the automata described above. This is done by drawing a *network* of automata with ATG (Fig. 7), in which each box of the net represents an automaton, and the labels of the ports (the small circles that are on the sides of the boxes) represent the actions that the automaton is able to perform. Whenever two ports are linked by an edge, the actions at these ports are synchronized; in the automaton corresponding to the system described by the net, synchronizations will be represented by means of the name of the synchronization event: this event can be observable, and in this case it takes the name labelling the edge, or unobservable, if no name labels the edge. Ports without links denote the asynchronous actions issued by the components, and they will appear with the same name in the automaton corresponding to the system described by the net. For example, we synchronize the action *!wr_req_set* of the producer with the action *?wr_req_set* of the buffer simply by drawing an edge between the related ports; in the automaton corresponding to the system described by the net, the synchronization between the two action takes the name *wr_req*, meaning that the *wr_req* signal is raised.

Notice that between the reset box and the buffer box there is a small disk (called *web*) having four outgoing edges; it is used to synchronize together such four edges: all the actions that are related to ports linked to the same web must occur at the same time; this corresponds to have a "line" that is used to send the reset signal to the whole system.

As we said previously, we have introduced an error in the specification of the system. The error is contained in the automata specifying the buffer; when the second position of the buffer is filled, no *!end_wr_set* signal is sent by the buffer in order to signal the end of the write operation. This may cause the producer waits indefinitely for the end of the write operation, unless a reset operation reinitializes the whole system.

The FC2 file that contains the global automaton of our specification is obtained by using FC2LINK to link the FC2 files (previously saved by ATG) in a single *network* that is contained in a single FC2 file (`system_lk.fc2`).

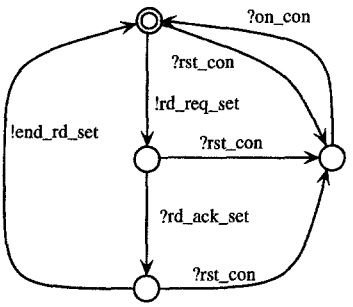

Fig. 3. The consumer

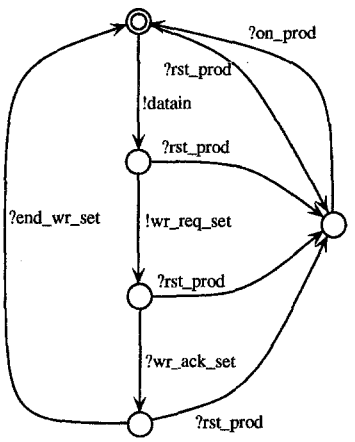

Fig. 4. The producer

4.3 Phase 2: Minimizing our Model with HOGGAR

The file `system_lk.fc2` obtained by FC2LINK is passed to HOGGAR to transform the automata network into the corresponding *single* automaton, that is the global model of the system that will be used (after a minimization step) for model checking. HOGGAR generates a model that is *reduced* with respect to the weak bisimulation. In our case the global buffer system model has 53 states before reduction, while the weak bisimulation reduction leads to a model with 38 states by stripping off most of the silent actions.

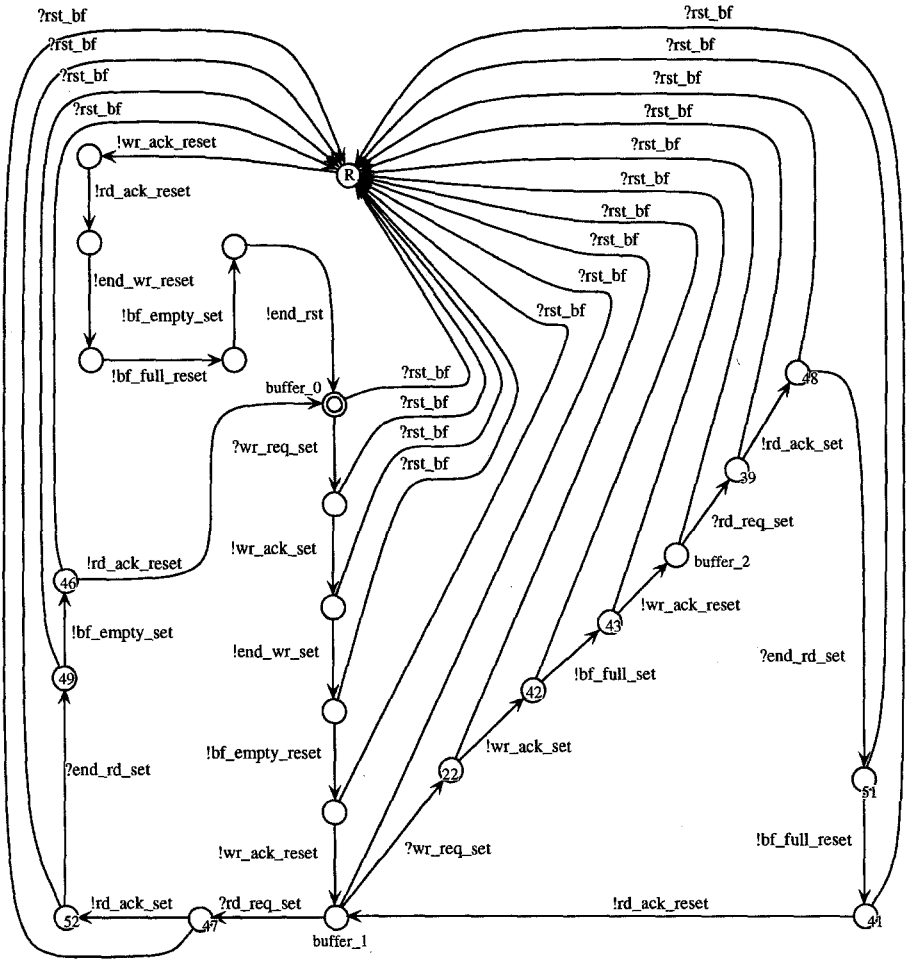

Fig. 5. The buffer

4.4 Phase 3: the Properties and their Formalization

Several properties of the buffer can be defined in terms of the sequence of event occurring at its input/output pins. Below we give three representative example requirements.

Data stability. *Data instability must not occur between the write service request and the end of the write operation.*
It is important to check this property on the whole system to guarantee that possibly detected malfunctions are not due to a misuse of the buffer. Moreover, we have also to consider that a reset operation can interrupt the write operation and release the stability requirement. Hence, the property can be expressed in terms of actions by:

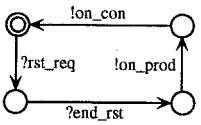

Fig. 6. The reset

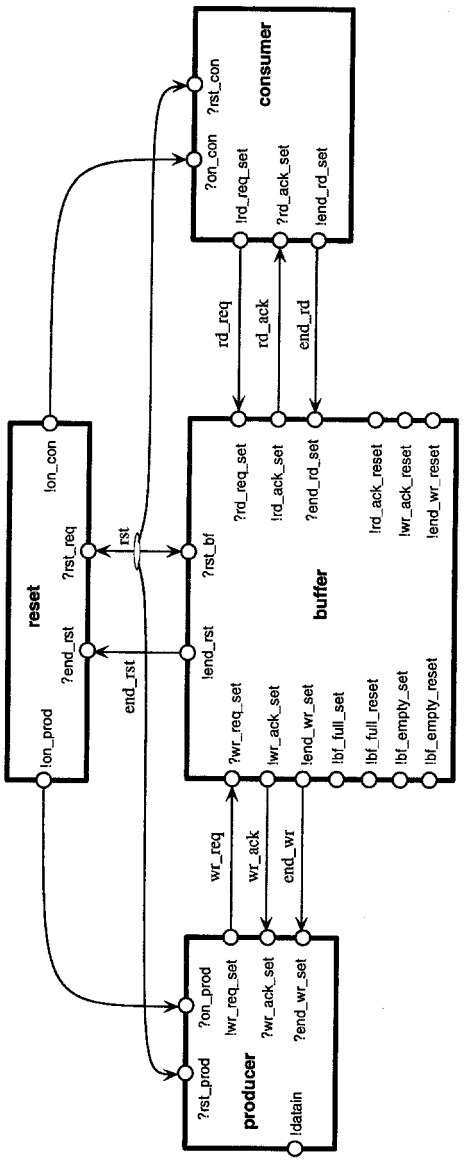

Fig. 7. The whole system

- *For all states, after* wr_req *it is not possible* !datain *until* end_wr *or* rst.

Proper reset. *After the* rst *asynchronous signal the level signals* wr_ack, rd_ack, end_wr, bf_full *should go to 0 and* bf_empty *should go to 1.*
Since we are dealing with an automaton in which actions are executed sequentially, we can express the fact that all these actions should be performed in a sequence immediately after the reset action:

- *For all states, after* rst *it is possible only* !end_wr_reset *or* !bf_full_reset *or* !wr_ack_reset *or* !bf_empty_set *until* !rd_ack_reset.

This property express only the fact that four of the five actions can be performed before *!rd_ack_reset* occurs: to match the proper reset requirement we have to add to it other four properties, each with the same structure of the first one, but with a different action (taken from the above mentioned five ones) after the until. The reset requirement will be the logical conjunction of such five properties. However, for the sake of space, we will consider only one of them.

Liveness. *Under the assumption that the units are not reset, we want that the system has always the possibility to get and process a new datum.*
We don't want that the processing of a new datum is bound to a reset operation, since we want that liveness is ensured by the pure hardware units data exchange protocols. Since the event related to the fetch of a datum is signalled by a *!datain* action, our property becomes:

- *For all states, there exists a path in which it is not possible* rst *until* !datain.

All the informal requirements above can be formalized using NL2ACTL: the three sentences above can be given as input to the tool that, after some interaction with the user, gives the following results:

```
For all states, after wr_req it is not possible !datain until !end_wr
or rst
 Parsed in: 11.400  secs.
 AG [wr_req] A[true {~!datain} U {(end_wr | rst)} true]
For all states, after rst it is possible !end_wr_reset or !bf_full_reset
or !wr_ack_reset or !bf_empty_set until !rd_ack_reset
 Parsed in: 36.583  secs.
 AG [rst] A[true {!end_wr_reset | !bf_full_reset | !wr_ack_reset
                 | !bf_empty_set} U {!rd_ack_reset} true]
For all states, there exists a path in which it is not possible rst until
!datain
 Parsed in: 9.437  secs.
 AG( E[ true {~ rst}U{!datain} true ] )
```

4.5 Phase 4: Formal Verification of the Properties

In this phase we use AMC to verify the properties described in Sect. 4.4 for the model that was generated in Sect. 4.2 and 4.3. The outputs of NL2ACTL are given to AMC, to perform the model checking; the AMC output is shown below:

```
|=  AG([wr_req]A[true {~ !datain} U {end_wr | rst} true])
  The formula is TRUE in state 10 time: (user: 0.02 sec, sys: 0.00 sec)
|=  AG([rst] (A[true {!end_wr_reset | !bf_full_reset | !wr_ack_reset |
                      !bf_empty_set}U{!rd_ack_reset} true]))
  The formula is TRUE in state 10 time: (user: 0.02 sec, sys: 0.00 sec)
|= AG ( E[ true {~ rst}U{!datain} true ] )
  The formula is FALSE in state 4 time: (user: 0.00 sec, sys: 0.00 sec)
```

We have that the third formula is not verified: this means that our system is not able to stay "alive". The error is in our buffer design. Now we will show how we can trace it, by taking advantage of the AMC model counterexample facilities. We simply ask AMC to tell us *why* the liveness formula is false: the tool exhibits us a path in which the formula is falsified; if there is more than one path that falsify the formula, AMC asks us to pick one of them; this happens also if there are more than one subformula that are falsified by a path. However, users can disable such AMC interactions and ask it to go through a path by performing some default choices. Below a subset of the semi–automatic explanations of AMC are reported:

```
why: (E[ true {(~ "rst")} U {!"datain"} true ]) false 22
   22 :
   |   labelled by : "wr_ack"  which satisfy : (~ "rst") or tau
   42 :
   |   labelled by : !"bf_full_set"  which satisfy : (~ "rst") or tau
   43 :
   |   labelled by : !"wr_ack_reset"  which satisfy : (~ "rst") or tau
   50 :
   |   labelled by : "rd_req"  which satisfy : (~ "rst") or tau
   39 :
   |   labelled by : "rd_ack"  which satisfy : (~ "rst") or tau
   48 :
   |   labelled by : !"bf_full_reset"  which satisfy : (~ "rst") or tau
   51 :
   |   labelled by : "end_rd"  which satisfy : (~ "rst") or tau
   41 :
   |   labelled by : !"rd_ack_reset"  which satisfy : (~ "rst") or tau
   45 :
   |   labelled by : "rd_req"  which satisfy : (~ "rst") or tau
   47 :
   |   labelled by : "rd_ack"  which satisfy : (~ "rst") or tau
   52 :
   |   labelled by : "end_rd"  which satisfy : (~ "rst") or tau
   49 :
```

```
|   labelled by : !"bf_empty_set"  which satisfy : (~ "rst") or tau
46 :
|   labelled by : !"rd_ack_reset"  which satisfy : (~ "rst") or tau
44 :
|   (~ (EX{!"datain"} true | EX{(~ "rst")} true)) is true
|   labelled by : "rst" which don't satisfy : ~"rst"| !"datain" or tau
14 : END
```

Numbers represent states. To make reading easy we have reported such numbers in Fig. 5. The (sub)path 22—44 falsifies the liveness property. By analyizing the counterexample, we are able to recognize where the error is, and eventually to correct it. After having fixed the bug the user should repeat the phases 1–3, in order to check if the new specification meets the requirements.

In our case, to fix the error we need to add a new state between the states 42 and 43 of the buffer automaton, causing the execution of a *!end_wr_set* action between the *!wr_ack_set* action and the *!bf_full_set*: in this way, the producer can be acknowledged of the end of the write operation and fetch new data to store in the buffer. As usual, in such a new state there should be an edge leading to the reset state R by the action *?rst_bf*.

Acknowledgements

The authors would like to thank C. Bernardeschi, A. Bouali, N. De Francesco, R. de Simone, G. Ferro, P. Inverardi, E. Madelaine, A. Masini, M. Nesi, S. Polverini, C. Priami, V. Roy, D. Yankelevich, the AiTech team and the Ansaldo Trasporti team for their contribution to the development and the experimentation of the JACK environment.

References

1. A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Fourth Workshop on Computer-Aided Verification*, Montreal, 1992, LNCS 663, Springer-Verlag.
2. A. Bouali, S. Gnesi, S. Larosa. The integration Project for the JACK Environment. Bulletin of the EATCS, n.54, October 1994, pp. 207-223.
3. E.M. Clarke, E.A. Emerson, A.P. Sistla: *Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications*, ACM Toplas, 8 (2), 1986, pp. 244-263.
4. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, Springer-Verlag, 1989, pp. 24–37.
5. M. Danelutto, G. DiCaprio, and A. Masini. Parallelizing a Model Checker. submitted for publication, 1995.
6. N. De Francesco, A. Fantechi, S. Gnesi, P. Inverardi. Model Checking of non-finite state processes by Finite Approximations. TACAS Workshop, LNCS, Springer-Verlag. May 1994.

7. R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. Computer Network and ISDN systems, Vol. 25, No.7, North Holland, 1993, pp. 761-778.
8. R. De Nicola, P. Inverardi, and M. Nesi. Equational reasoning about LOTOS specifications: A rewriting approach. In *Sixth International Workshop on Software Specification and Design*, 1991, pp. 54-67.
9. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science*, LNCS 469, Springer-Verlag, 1990, pp. 407–419.
10. R. De Nicola and F.W. Vaandrager. Three Logics for Branching Bisimulation. *Journal of ACM*, Vol. 42, N. 2, 1995, pp. 458-487.
11. E. A. Emerson, J. Y. Halpern. "Sometimes" and "Not Never" Revisited: on Branching Time versus Linear Time Temporal Logic. Journal of ACM, 33 (1),1986, pp. 151-178.
12. A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods in Systems Design*, 4(2), Elsevier Science Publisher, 1994, pp. 243-263.
13. G. Ferro. AMC: ACTL Model Checker. Reference Manual. IEI-Internal Report, B4-47 December 1994.
14. M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, **32**, 1985, pp. 137-161.
15. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.
16. P. Inverardi, C. Priami, and D. Yankelevich. Verifing concurrent systems in SML. In *SIGPLAN ML Workshop*, San Francisco, June 1992.
17. E. Madelaine and R. De Simone. The fc2 reference manual. Technical report, INRIA, 1993.
18. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.
19. V. Roy and R. de Simone. AUTO and autograph. In R. Kurshan, editor, *Workshop on Computer Aided Verification*, New-Brunswick, June 1990. LNCS 531, Springer-Verlag, pp. 65-75.