Proc. of *International Workshop on Current Trends in Applied Formal Methods*, Boppard,
Germany, 7-9 October 1998, Lecture Notes in Computer Science, Springer-Verlag

A Symbolic Model Checker for ACTL*

A. Fantechi[1,2], S. Gnesi[2], F. Mazzanti[2], R. Pugliese[1] and E. Tronci[3]

[1] Dip. di Sistemi e Informatica, Univ. di Firenze, Italy
[2] Istituto di Elaborazione dell'Informazione, C.N.R. Pisa, Italy
[3] Dip. di Matematica Applicata, Univ. dell'Aquila, Italy

Abstract. We present SAM, a symbolic model checker for ACTL, the action-based version of CTL. SAM relies on implicit representations of Labeled Transition Systems (LTSs), the semantic domain for ACTL formulae, and uses symbolic manipulation algorithms. SAM has been realized by translating (networks of) LTSs and, possibly recursive, ACTL formulae into BSP (Boolean Symbolic Programming), a programming language aiming at defining computations on boolean functions, and by using the BSP interpreter to carry out computations (i.e. verifications).

1 Introduction

The increasing reliance of many aspects of human society on highly complex computer systems requires the adoption of innovative validation techniques. In the validation of software difficulties arise from the discontinuous nature of the software behaviour. This behaviour is based on sequences of discrete transitions, with such a high number of possible evolution paths and failure modes, that exhaustive testing becomes impossible. Moreover, testing can provide information only on the tested paths. Hence, due to the lack of continuity, we cannot infer the behaviour of untested sequences from that of the tested ones.

Formal methods are mathematically based techniques that can offer a rigorous and effective way to model, design and analyze computer systems. It is increasingly accepted that the adoption of formal methods in the life cycle development of embedded systems would guarantee higher levels of dependability. It appears that, due to the lower costs of training and innovation, industries are more keen to accept formal validation techniques assessing the quality attributes of their products, obtained by a traditional life cycle, rather than a fully formal life cycle development. However, to achieve an efficient use of formal methods in industry, such methods need to be better integrated with traditional software engineering. Formal "validation" and "verification" techniques and automated support tools need to be improved so that they could be easily used by non-expert staff.

Model checking techniques [8] has been defined to verify system properties, expressed as temporal logic formulae, on finite state models of the behavior

* This work was partly supported by the ESPRIT project GUARDS and by Progetto Coordinato CNR "Metodologie e strumenti di analisi, verifica e validazione per sistemi software affidabili".

of systems. Once a model of a system has been generated, the properties are automatically verified by model checking tools and therefore these kind of tools can be easily used also by non-expert users. This in general is not true for verification techniques based on theorem proving approaches [5]. In this case, the system state is modeled in terms of set-theoretical structures, and operations are modeled by specifying their pre- and post-conditions in terms of the system state. Properties are described by invariants that must be proved to hold through the system execution by means of a theorem prover, usually with the help of the user. Due to the above reasons model checking has been preferred in industries, especially for formal verification of hardware components.

Many "prototipal" verification environments are currently available which can be used to automatically verify behavioural and logical properties of reactive and concurrent systems specified by means of process algebrae. Most of these environments (e.g. [9, 10, 17, 11, 4]) use finite state systems to model the systems under investigation and formulae of temporal logics to express properties [19, 26]. Usually, given a system, a so called "generation phase", based on the operational semantics of the language, allows the corresponding LTS to be derived. When real world applications are considered a main problem arises due to their extremely large state-spaces.

To cope with the "state-explosion" in the model generation phase some work has been recently done for the logic CTL [14], a branching-time temporal logic whose interpretation domains are Kripke structures. Indeed, the SMV model checker has been developed [7], which uses symbolic manipulation algorithms to check the satisfiability of CTL formulae. In SMV the transition relations are represented implicitly by means of boolean formulae and are implemented by means of Binary Decision Diagrams (BDDs, [6]). This usually results in a much smaller representation for the systems' transition relations thus allowing the maximum size of the systems that can be dealt with to be significantly enlarged.

CTL formulae allow properties of systems to be expressed in terms of their states. In case of systems which are described in terms of actions and state changes, such as concurrent (e.g. control) systems, it is more natural to use an action-based temporal logics to express their properties. For instance, one can use ACTL [13], the action-based version of CTL. Our effort has then been to build efficient model checking tools for action-based logics by directly relying on implicit BDD-based descriptions of systems' state spaces. In particular, we have built SAM, a symbolic model checker for μ-ACTL that relies on implicit representations of LTSs and symbolic manipulation algorithms. As logic language we have used μ-ACTL [16] since ACTL, although quite expressive (e.g. it permits to express safety and liveness properties, as well as certain "cyclic" properties), lacks of a "real" fixpoint operator which permits to express general recursive properties (whereas this is possible by using, e.g., the μ-calculus [20]). The symbolic model checker has been realized by translating (networks of) LTSs and μ-ACTL formulae into Boolean Symbolic Programming (BSP, [27]), a programming language aiming at defining computations on boolean functions, and by using the BSP interpreter to carry out computations (i.e. verifications).

The integration of SAM in JACK [4], an environment for the specification and formal verification of concurrent systems that also includes a model checker for ACTL using explicit state space representation, is in progress. A number of formal validation projects using SAM are under progress too. In [24] model checking of a hydroelectric power plant with (potentially) 10^{52} states has been carried out.

2 Background

2.1 JACK

Our starting point has been JACK (*Just another Concurrency Kit*) [4], that so far is the only verification environment including an ACTL model checker (AMC, [18]). JACK is an environment based on the use of process algebrae, automata and temporal logic formalisms, which supports many phases of the system development process. The idea behind the JACK environment is to integrate different specification and verification tools, independently developed at different research institutes (I.E.I.- C.N.R. and the University of Rome "La Sapienza" in Italy, and INRIA in France), to provide an environment in which a user can choose from several verification tools by means of a user-friendly graphic interface.

The FC2 format [21], i.e. the common representation format for data, makes it possible to exchange information among the tools integrated in JACK and to easily add other tools to the JACK environment, thus extending its potential. The FC2 format allows a Labeled Transition System (i.e. an automaton) to be represented by means of a set of tables that keep the information about state names, arc labels, and transition relations between states. The format allows nets of automata to be represented as well.

Some of the tools in JACK allow a process specification to be built. This can be done both by entering a specification in a textual form (i.e. a process algebraic term) by using MAUTO, or by drawing the automaton that describes the behavior of the process by using ATG [25]. Moreover, sophisticated graphical procedures, provided by ATG, allow a specification to be built as a network of processes (or networks). Hence, a hierarchical approach in the specification activity is also possible.

Once the specification of a system has been written, JACK permits the construction of the automaton corresponding to the behaviour of the overall system, by using either MAUTO or FC2LINK and HOGGAR (which is a BDD-based tool); this is the *model* of the system. Moreover, by using MAUTO or HOGGAR, automata can be minimized with respect to various (bisimulation) equivalences. ACTL can be used to describe temporal properties and *model checking* can be performed, by using AMC, to check whether systems (i.e. their models) satisfy the properties.

JACK has been successfully used in several case studies. In [12] JACK was used to formally specify the hardware components of a buffer system, and to verify the correctness of the specification with respect to some safety requirements. In [3] the verification of an interlocking safety critical system developed by Ansaldo Trasporti was presented.

Model checking by using JACK has a major limiting factor, namely the state space explosion problem. Indeed, AMC can perform model checking only onto a single automaton (i.e. AMC cannot take a network of automata as a model); thus it is always necessary to generate the global automaton of the system. SAM, the extension of JACK presented in this paper, is aimed at solving the state space explosion problem.

2.2 CCS/Meije

Process algebrae [22] are generally recognized as a convenient tool for describing reactive systems at different levels of abstraction. They rely on a small set of basic operators, used to build complex descriptions from more elementary ones, and on behavioral equivalences (e.g. bisimulation) or preorders (e.g. testing), used to study the relationships between descriptions of the same system at different levels of abstraction (e.g., specification and implementation).

In the JACK environment, the process algebra used to define processes is CCS/Meije [1]. The syntax of the language is based on a set of elementary and uninterpreted actions that processes can perform and on a set of operators that permit to build complex processes from simpler ones. The syntax permits a two-layered design of *process terms*. The first level is related to *sequential regular terms*, the second one to *networks* of parallel sub-processes supporting communication and action renaming or restriction. The two-layered structure of CCS/Meije descriptions is reflected also by the graphical methodology that can be used in connection with ATG and that will be illustrated in Section 3.2.

For the syntax and the operational semantics of CCS/Meije, we refer the interested reader to [1].

2.3 μ-ACTL

In this section we briefly present the syntax of μ-ACTL [16], an extension of ACTL [13], the temporal logic used in the JACK environment, with a fixpoint operator. For the semantics of formulae, we refer the interested reader to [13] and [16].

μ-ACTL is suitable to express properties of reactive systems whose behaviour is characterized by the actions they perform and whose semantics is defined by means of LTS's. The logic can be used to define both *liveness* (something good eventually happen) and *safety* (nothing bad can happen) properties of reactive systems. Moreover, μ-ACTL is *adequate* with respect to strong bisimulation equivalence, namely if $p \sim q$, then p and q satisfy the same set of μ-ACTL formulae.

The definition of μ-ACTL relies on an auxiliary logic of action. The collection $\mathcal{AF}$ (ranged over by χ) of *action formulae* over a set of (visible) actions *Act* (ranged over by α) is defined by the following grammar:

$\chi ::= \alpha | \neg\chi | \chi \wedge \chi$.

We write *ff* for $\alpha_0 \wedge \neg\alpha_0$, where a_0 is some chosen action, *tt* for $\neg$*ff* and $\chi \vee \chi'$ for $\neg(\neg\chi \wedge \neg\chi')$. Intuitively, action formulae express sets of (observable) actions. Given an action formula χ, the set of the actions satisfying χ is $\kappa(\chi) = \{\alpha | \alpha \models \chi\}$.

The syntax of μ-ACTL formulae is defined by the following grammar:

$\phi ::= tt \mid \phi \wedge \phi \mid \neg\phi \mid E\pi \mid A\pi \mid \mu Y.\phi(Y) \mid Y$

$\pi ::= X_\chi \phi \mid X_\tau \phi \mid \phi \,_\chi U \, \phi \mid \phi \,_\chi U_{\chi'} \, \phi$

where Y ranges over a set Var of variables, *state formulae* are ranged over by ϕ, *path formulae* are ranged over by γ, E and A are *path quantifiers*, X and U are the *next* and *until* operators indexed by action formulae.

Several useful derived modalities can be defined, starting from the basic ones. In particular, we will write:

- $< \chi > \phi$ for $E[tt \,_{ff} U_\chi \phi]$ and $[\chi]\phi$ for $\neg < \chi > \neg\phi$.
- $EF\phi$ for $E[tt \,_{tt} U \, \phi]$, and $AF\phi$ for $A[tt \,_{tt} U \, \phi]$; these are called the *eventually* operators.
- $EG\phi$ for $\neg AF\neg\phi$, and $AG\phi$ for $\neg EF\neg\phi$; these are called the *always* operators.
- $\nu Y.\phi(Y)$ for $\neg\mu Y.\neg\phi(\neg Y)$; ν is called the maximal fixpoint operator.

2.4 Boolean Symbolic Programming (BSP)

BSP is a programming language aimed at defining, possibly fixpoint, computations on boolean functions [27]. BSP primitives include boolean operations, quantifiers, arithmetic operations (ALU-like). Defining processes just using boolean operations is a tedious and error prone task. Thus BSP has primitives to define processes as well. The output of a BSP program is formed by "answers" to BSP queries. Essentially BSP queries correspond to the usual queries on BDDs. E.g.: "Is a given boolean function identically 1 (true)?", "Print the set of satisfying assignments of a given boolean function". Shortly, BSP is a programming language to define symbolic computations at a logical level rather than at the BDD level.

The BSP compiler translates a BSP program into a sequence of calls to BDD primitives (essentially `if_then_else` and `compose`). This translation is done in time $O(s)$ where s is the size of the BSP program being translated. During such translation some optimization is also carried out (e.g. when possible BDD calls are moved outside of loops computing fixpoints, BDD calls are rearranged in an attempt to free BDDs as soon as possible). BSP tries to efficiently execute a given symbolic program. However, as for any programming language, it is the user responsibility to write efficient (symbolic) programs.

Implementations as well as specifications can both be defined using BSP. E.g.: a netlist of size n can be translated into a BSP program of size $O(n)$; a μ-calculus formula of size n can be translated into a BSP program of size

$O(n)$ [27]. These features make BSP suitable as a low level language to define finite state verification tasks. Moreover having implementation and specification defined using the same language enables the use of rewriting techniques to speed up verification [27]. Note that BSP is a programming language rather than a Model Checker. Thus BSP can be used for other purposes as well (e.g. see [28]).

3 The extended version of JACK

In this section we comment on the architecture of the extended version of JACK, which is depicted in Figure 1, by especially pointing out the new features introduced with respect to JACK. Then, by means of a simple example, we will show how the new environment can be used as a support for the specification and verification of reactive systems.

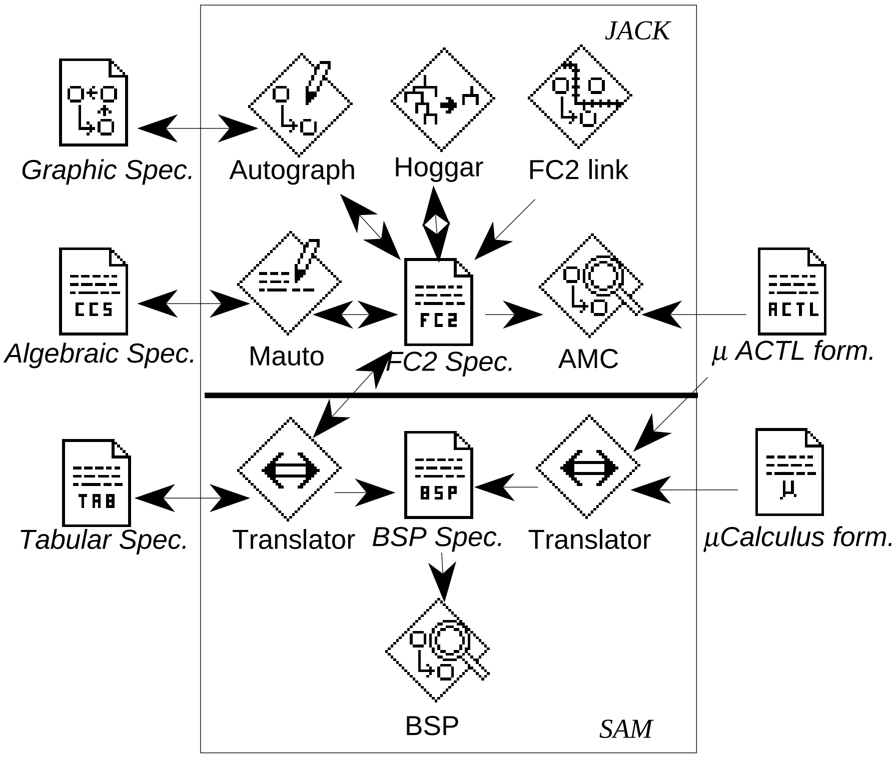

Fig. 1. The architecture of the extended version of JACK

3.1 The architecture

The user can use three different formalisms for defining the reactive systems used as models for checking logical properties. A part from textual (CCS/Meije) and graphical (ATG) representations, that both are translated first into the FC2 format and then into BSP, a *tabular* representation can be used as well, that is directly translated into BSP.

Tabular representations have the same overall structure of FC2 specifications (i.e. the description of a system is composed of the descriptions of the system components and of the descriptions of their synchronizations), but have simplified syntax and semantics so that the structure of systems is described in a more straightforward way. The tabular representation aims at helping the user to better understand system descriptions as well as to make the user able to generate by himself system descriptions directly without using graphical representations.

The user can express logical properties that must be checked as μ-calculus and μ-ACTL formulae.

Both the formalisms for specifying processes, basically FC2 and tabular representations, and the formalisms for specifying properties, namely μ-calculus and μ-ACTL, are translated, in linear time complexity, into BSP. Therefore, checking whether a reactive system s verifies a property p consists in querying the BSP interpreter if the boolean function "$\mathrm{tr}(s)$ implies $\mathrm{tr}(p)$" is identically one, where $\mathrm{tr}(s)$ is the BSP program that represents s and $\mathrm{tr}(p)$ is the BSP program that represents p.

3.2 Symbolic model checking in practice

System specification

Let us consider a level crossing that can be crossed at a given time either by a train or by at most two cars. A train asks for the permission to cross by using action `approaching_t` and signals that it has crossed by using action `leaving_t`. A car behaves similarly but uses actions `approaching_c` and `leaving_c`, respectively. These are the only visible actions of the system. Normally, the barriers are kept open, thus cars can cross. The railway signal allows a train to proceed only if the barriers can go down safely, namely when no car is crossing. After a train has crossed, the barriers will go up again.

The specification of the system is a net, called **crossing**, composed of two automata, **barrier** representing the controller of the barrier and **signal** representing the controller of the railway signal. Figure 2 shows their graphical ATG representations. While the two component automata (top of Figure 2) should be self-explicative, here we comment on the network that describes the overall system (bottom of Figure 2). Boxes (e.g. **barrier** and **signal**) can be processes or networks, thus allowing a top-down approach in the specification activity. Ports at the border of boxes are their interconnection places. If two boxes are drawn at the same level, they can synchronize via the actions that label linked ports. In addition to CCS/Meije, a multiway synchronization operator, called *wedge*, is available: more than two processes can synchronize by executing an action (e.g **barrier** and **signal** synchronize on `approaching_t` which labels a wedge). As in the case of a textual specification, the behaviour of a graphical specification can be defined in terms of an LTS.

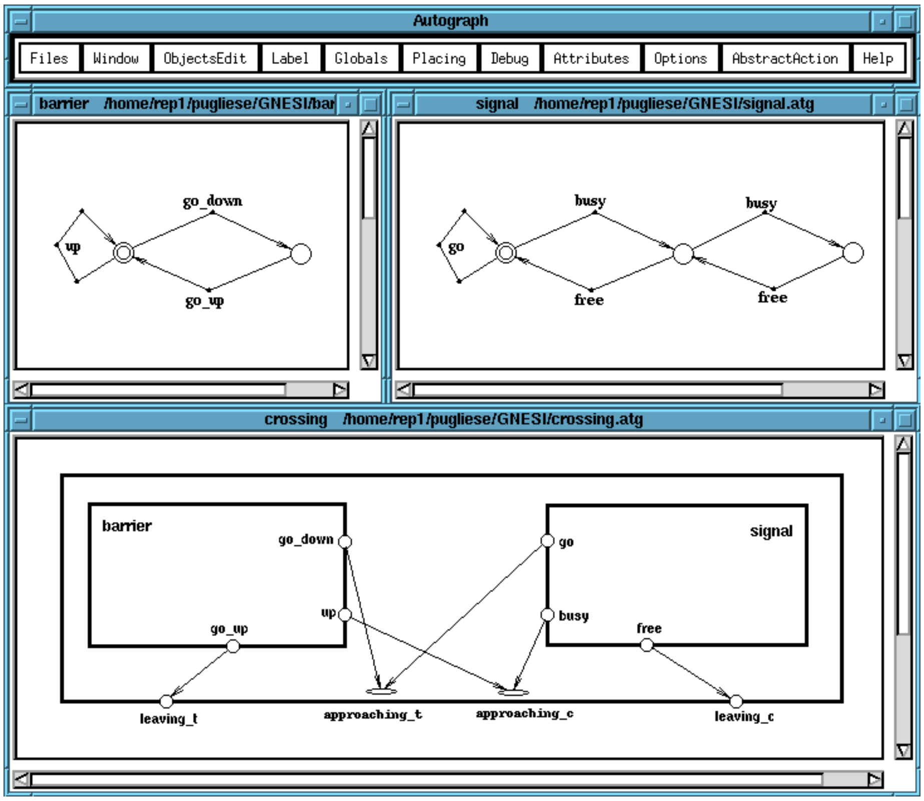

Fig. 2. The level crossing: a pictorial representation

Once the FC2 representation of **crossing** has been obtained by using ATG and FC2LINK, we can construct the corresponding tabular representation by using the utility `totab`.

This construction, shown in Figure 3, is not needed and has been done only to give a flavor of what the tabular representation is. In this representation, all the possible actions that an automaton or a net can do are explicitly enumerated. The tabular description of an automaton, other than the possible actions, specifies the possible transitions that the automaton can do. In defining the transitions, action indexes are used in place of action names. More specifically, transitions are specified as triples of numbers of the form: $n0$: $n1$ `->` $n2$. Such a triple indicates that the automaton can evolve from state $n1$ to state $n2$ by performing action $n0$. The tabular description of a net with n components specifies

```
Networks: 3
Main: 1

System: 1 "crossing"     Automaton: 2 "barrier"    Automaton: 3 "signal"
Actions: 4               Actions: 3                Actions: 3
 1: approaching_t         1: go_down                1: go
 2: approaching_c         2: up                     2: busy
 3: leaving_c             3: go_up                  3: free
 4: leaving_t            States: 2                 States: 3
Components: 2 3          Initial: 1                Initial: 1
Synchronizations:        Transitions:              Transitions:
 1: 1 1                   2: 1 -> 1                 1: 1 -> 1
 2: 2 2                   1: 1 -> 2                 2: 1 -> 2
 3: 0 3                   3: 2 -> 1                 2: 2 -> 3
 4: 3 0                                             3: 2 -> 1
                                                    3: 3 -> 2
```

Fig. 3. Tabular representation of **crossing**

the structure of the global system by means of the n-uple of numbers following the keyword `Components`. This n-uple indicates the templates of automata that constitute the system. In the case of **crossing**, the tuple `2 3` says that the system is composed by one instance of the automaton `2`, i.e. **barrier** and one instance of the automaton `3`, i.e. **signal**. The synchronizations among the components are specified as n+1-uple of numbers. Hence, in our case we have triples of numbers. A triple of the form $n0$: $n1$ $n2$ indicates that if the first component can perform action $n1$ and the second component can perform action $n2$ then the system can perform action $n0$. A 0 value for ni means that the corresponding component does not take part into the synchronization. The major simplification with respect to FC2 specifications is that both the transition definitions and the synchronization definitions use "plain" action indexes rather than generic expressions with action indexes as operands.

Both the FC2 and the tabular representations can be given as an input to the utility `tobsp` that returns a BSP program that represents the system.

BSP program of the specification

The BSP program that represents the **barrier** is given in Figure 4. Hereafter, we explain the main features of the language BSP. Examples refer to Figures 4, 5 and 7.

Comments are C-like, i.e. started with `/*` and ended with `*/`.

A *declaration* of the form `(def` *id* `(array` *n*`))` defines the identifier *id* to be a vector of *n* boolean variables. We will refer to an identifier declared in this way as an *array*. Thus arrays denote vectors of boolean variables. For example, `(def C1_a (array 2))`, `(def C1_px (array 1))` declare arrays ranging, respectively,

```
(def C1_a (array  2))
(enum  2 0 (C1_a0 C1_a1 C1_a2 C1_a3))
(def C1_px (array  1))
(def C1_nx (array  1))
(enum  1 0 (C1_s1 C1_s2))
(def C1_S ( eqv C1_px C1_s1))
(def C1_R
  (defprocess
    (present_state C1_px)
    (next_state C1_nx)
    (transition
      (update C1_px)
      (eqv C1_px C1_s1)
      (eqv C1_a C1_a2)
      (eqv C1_nx C1_s1))
    (transition
      (update C1_px)
      (eqv C1_px C1_s1)
      (eqv C1_a C1_a1)
      (eqv C1_nx C1_s2))
    (transition
      (update C1_px)
      (eqv C1_px C1_s2)
      (eqv C1_a C1_a3)
      (eqv C1_nx C1_s1)) ))
```

Fig. 4. BSP program of **barrier**

on actions and present states of `C1_`. Boolean variables are represented with BDD variables. Unless otherwise instructed BSP uses as BDD variable ordering the ordering in which boolean variables (arrays) are declared. It is possible to override this behaviour by instructing BSP to follow a user given BDD variable ordering.

A *declaration* of the form `(def` id `(record` $X_1 \ \ldots X_n$`))` defines identifier id as the vector of boolean variables obtained by concatenating the vectors of boolean variables $X_1 \ldots X_n$. We will refer to an identifier declared in this way as a *record*. Note that no new BDD variable is created by such declaration. For example, `(def px (record C1_px C2_px))` gives name `px` to the vector of boolean variables formed by the variables in `C1_px` or in `C2_px`.

A *declaration* of the form `(enum` $size$ $offset$ `(`$id_0 \ \ldots id_{k-1}$`))` defines identifiers $id_0 \ldots id_{k-1}$ to be vectors of $size$ boolean values. Identifier id_0 is the boolean representation of $offset \bmod 2^{size}$, id_1 is the boolean representation of $(offset + 1) \, mod \, 2^{size}$, etc. For example, `(enum 2 0 (C1_a0 C1_a1 C1_a2 C1_a3))` assigns to `C1_a0`, `C1_a1`, `C1_a2`, `C1_a3` respectively the vectors [0 0], [1 0], [0 1], [1 1] (note: the leftmost bit is the least significant bit).

A *term* of *size* n is a vector of n boolean expressions. For example, `C1_a`, C1_s1, px, are terms of size, respectively, 2, 1, 3. More complex terms can be built using boolean operators and quantifiers. For example, if a is a record or an array and R and F_OF_1 are terms of size n then the following are terms of size n: (and R F_OF_1) (semantics: bitwise *and* of R and F_OF_1), (exists a (and R F_OF_1)) (semantics: ($\exists$ a (R $\wedge$ F_OF_1))). If t_1, t_2 are terms of size n then (eqv t_1 t_2) is a term of size 1 evaluating to 1 iff terms t_1, t_2 are bitwise equal.

A *definition* (def *id* *t*) assigns to *id* the value of term *t*. For example, C1_S denotes a boolean function which is 1 (true) iff C1_px is bitwise equal to C1_s1.

A *term* of the form (defprocess ...) is used to define the transition relation of a process (LTS). For example, C1_R is a boolean function which is 1 iff in the LTS C1_ there is a transition labeled with C1_a from state C1_px to state C1_nx.

The transition relation for the parallel composition of two LTSs is obtained by defining in BSP the semantics of the parallel composition operator. For each automaton constituting the system (i.e. **barrier** and **signal**), a boolean function modeling its set of transitions is constructed. Similarly, the global system (i.e. **crossing**) transitions are modeled as a boolean function constructed starting from the transitions of the system components. Part of the BSP program that represents the **crossing** is given in Figure 5.

The properties

We have verified the following properties:

1. if a train leaves the level crossing (event leaving_t) then, first, it has to approach to the level crossing (event approaching_t);
2. if a train approaches to the level crossing then it must immediately leave the level crossing;
3. if a car approaches the crossing (event approaching_c) then no train can approach the crossing until a car leaves (event leaving_c) the crossing;
4. it is possible to have any number of cars approaching and leaving the crossing.

The μ-ACTL formulae that correspond to the previous properties are shown in Figure 6.

BSP programs of the properties

The translation from μ-ACTL to BSP is done by defining with BSP the semantics of μ-ACTL formulae. The translation of a single μ-ACTL formula into a list of boolean function occurs in two steps. First the μ-ACTL formula is translated into the μ-calculus. Then, the resulting μ-calculus formula is compositionally translated into a list of boolean functions (BSP program). The standard translation from μ-ACTL to μ-calculus can be found in [16]. Figure 7 shows the (simplified) BSP translation of the first formula in Figure 6.

```
/* definition of missing components */
(def a (array  3))
(enum  3 0 (a0 a1 a2 a3 a4))
(def px (record C1_px C2_px))
(def nx (record C1_nx C2_nx))
(def aa (record C1_a C2_a))
(def S (and C1_S C2_S))
(def R (or
  (exists aa (and
    (eqv a a1)
    C1_R
    (eqv C1_a C1_a1)
    C2_R
    (eqv C2_a C2_a1)))
  (exists aa (and
    (eqv a a2)
    C1_R
    (eqv C1_a C1_a2)
    C2_R
    (eqv C2_a C2_a2)))
  (exists aa (and
    (eqv a a3)
    (eqv C1_px C1_nx)
    C2_R
    (eqv C2_a C2_a3)))
  (exists aa (and
    (eqv a a4)
    C1_R
    (eqv C1_a C1_a3)
    (eqv C2_px C2_nx)))  ))
```

Fig. 5. Part of the BSP program of **crossing**

Results of model checking

Once that the BSP programs of the specification and of the logical formulae have been produced, for each formula that must be checked, i.e. for each boolean function `F_*_i`, a query of the form

```
(def check_fun_i (forall px (imp S F_*_i)))
(isone check_fun_i)
```

is added to the file containing the translations, and the BSP interpreter is called on the file. The BSP term `(isone check_fun_i)` asks BSP to check whether the boolean function `check_fun_i` is identically 1, in that case the formula is true. The results of the model checking of the formulae in Figure 6 are in Figure 8. Note that all the formulae are true.

```
~ (EF  <~ approaching_t> <leaving_t> true).

AG ( [approaching_t] AX {leaving_t} true).

AG ([approaching_c] A [true{~approaching_t} U  {leaving_c} true]).

nu FORM : <approaching_c | leaving_c> FORM.
```

Fig. 6. μ-ACTL formulae

```
(def F_3FFB_1 (exists nx (exists a (and R (eqv a a4)))))

(def F_2FF_1 (exists nx (and (exists a (and R (not (eqv a a1))))
                                       (compose F_3FFB_1 px nx))))

(def F_1F_1 (or F_2FF_1  F_0E_1))
(def F_0E_1 (exists nx (and (exists a (and R b1)) (compose F_1F_1 px nx))))

(def F_3F_1 (not F_0E_1))
```

Fig. 7. BSP translation of the first formula in Figure 6

4 Conclusions

We have presented a symbolic model checker, SAM, for an action-based temporal logics that directly performs verification over Labeled Transition Systems. To our knowledge, this is the first attempt in this direction, since previous symbolic model-checkers have been defined for state-based temporal logics such as CTL.

SAM is currently used in a number of formal validation projects, among which we recall, in particular, the validation of fault tolerance mechanisms defined for an architecture for dependable systems, developed inside the european project GUARDS [2]. Three fault-tolerant mechanisms (namely, Inter-Consistency algorithm, Fault-Treatment mechanism and Multi-Level Integrity mechanism), have been modeled using the tools of the JACK environment; the possible occurrence of faults has been modeled by introducing explicit "fault" actions in the model, and different fault assumptions have been modeled, in order to study the behaviour of the mechanism under different fault hypotheses. The satisfaction of some critical properties of the mechanism, both in presence of faults or not, is the object of the on-going validation effort. In a first phase of the project, we were able to verify the properties of the Inter-node consistency algorithm when designed for a three node GUARDS architecture. The model exhibited (after some abstraction) around 70000 states and was affordable by AMC, the traditional model checker for ACTL. When the algorithm for a four-node GUARDS architecture has been considered in the next phase of the project, it turned out

```
isone check_fun_1 AutVarOrd "check_fun_1" =
function check_fun_1 is identically 1

isone check_fun_2 AutVarOrd "check_fun_2" =
function check_fun_2 is identically 1

isone check_fun_3 AutVarOrd "check_fun_3" =
function check_fun_3 is identically 1

isone check_fun_4 AutVarOrd "check_fun_4" =
function check_fun_4 is identically 1
```

Fig. 8. Answers of the BSP interpreter

that it was no more tractable by AMC, since the size was grown to 10^7 states. The fault treatment mechanism is the largest model we have generated inside this project, amounting to $2 * 10^9$ states. Verification of critical properties is now in progress using SAM.

The integration of SAM within the JACK environment is under development, together with a friendly user interface.

Acknowledgments We thank Cinzia Bernardeschi and Antonella Santone for interesting discussions about the realization of this tool.

The development of SAM has seen at its beginning the contribution of Gioia Ristori, who has recently left us for a better world. She will however remain with us for ever.

References

1. D. Austry, G. Boudol. Algebre de processus et synchronization, *Theoretical Computer Science*, 1(30), 1984.
2. C. Bernardeschi, A. Fantechi, S. Gnesi. Formal Specification and Verification of the Inter-Channel Consistency Network, GUARDS Esprit project, Technical Report D3A4/6004c, 1997.
3. C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, D. Romano. A Formal Verification Environment for Railway Signaling System Design, in Formal Methods in System Design 12, 139-161, 1998.
4. A. Bouali, S. Gnesi, S. Larosa. The integration Project for the JACK Environment, Bulletin of the EATCS, n.54, pp.207-223, 1994. (see also http://rep1.iei.pi.cnr.it/projects/JACK.)
5. R.S. Boyer, J.S., Moore. "A Computational Logic", ACM Monograph Series, Academic Press, 1979.
6. R.E. Bryant. Graph Based algorithms for boolean function manipulation, *IEEE Transaction on Computers*, C-35(8), 1986.

7. J.R. Burch, E.M.Clarke, K.L. McMillan, D. Dill, J. Hwang. Symbolic Model Checking 10^{20} states and beyond, in Proceedings of LICS, 1990.
8. Clarke, E.M., Emerson, E.A., Sistla, A.P., "Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specification," ACM Transaction on Programming Languages and Systems, 8(2):244-263, 1986.
9. R. Cleaveland, J. Parrow, B. Steffen. The Concurrency Workbench, in Proc. of Automatic Verification Methods for Finite State Systems, LNCS 407, pp. 24-37, 1990.
10. R. Cleaveland, S. Sims. The NCSU Concurrency Workbench, in Proc. of Computer Aided Verification, LNCS 1102, pp. 394-397, 1996.
11. R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems, *Computer Networks and ISDN Systems*, 25(7):761-778, 1993.
12. R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. Verifying Hardware Components within JACK, in Proceedings of CHARME '95, LNCS 987, pp. 246-260, 1995.
13. R. De Nicola, F. W. Vaandrager. Action versus State based Logics for Transition Systems, Proceedings Ecole de Printemps on Semantics of Concurrency, LNCS 469, pp. 407-419, 1990.
14. E.A. Emerson, J. Halpern. ″Sometime″ and ″Not never″ revisited: On branching versus linear time temporal logic, *JACM* 33:151-178, 1986.
15. E.A. Emerson, C. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus, in Proceedings of LICS, pp. 267-278, 1986.
16. A. Fantechi, S. Gnesi, G. Ristori. From ACTL to μ-Calculus, ERCIM Workshop on Theory and Practice in Verification, Pisa, December 9-11, 1992.
17. J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox, CAV'96, LNCS 1102, pp. 436-440, 1996.
18. G. Ferro. "AMC: ACTL Model Checker. Reference Manual", IEI-Internal Report B4-47, December 1994.
19. M. Hennessy, R. Milner. Algebraic Laws for Nondeterminism and Concurrency, *JACM* 32:137-161, 1985.
20. D. Kozen. Results on the Propositional μ-calculus, *Theoretical Computer Science*, 27:333-354, 1983.
21. E. Madelaine, R. de Simone. The FC2 Reference Manual, Available by ftp from `cma.cma.fr:pub/verif` as file *fc2refman.ps.gz*, 1993.
22. R. Milner. Communication and Concurrency, *Prentice-Hall International*, Englewood Cliffs, 1989.
23. G.D. Plotkin. A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Aarhus University, Dep. of Computer Science, Denmark, 1981.
24. R. Pugliese, E. Tronci. Automatic Verification of a Hydroelectric Power Plant. FME'96, LNCS 1051, pp. 425-444, 1996.
25. V. Roy, R. De Simone. AUTO and Autograph, in Proceedings of the Workshop on Computer Aided Verification, LNCS 531, 65-75, 1990.
26. C. Stirling. An Introduction to modal and temporal logics for CCS, In Concurrency: Theory, Language, and Architecture, LNCS 391, 1989.
27. E. Tronci. Hardware Verification, Boolean Logic Programming, Boolean Functional Programming, in Proceedings of LICS, 1995.
28. E. Tronci. On Computing Optimal Controllers for Finite State Systems, Proc. of the 36th IEEE Conf. on Decision and Control, 1997.