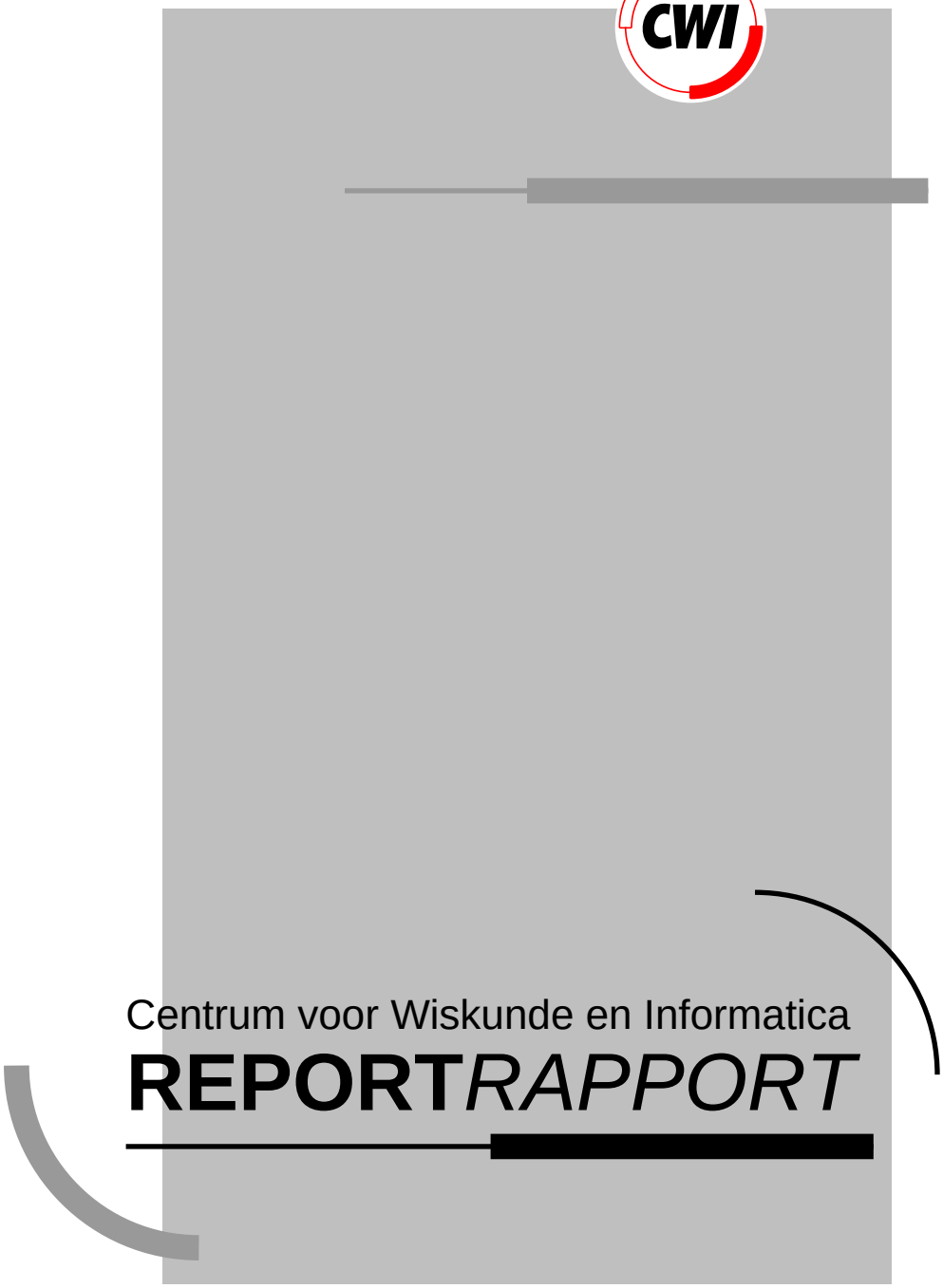

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

The Integration Project for the JACK Environement

Amar Bouali, Stefania Gnesi, Salvatore Larosa

Computer Science/Department of Software Technology

CS-R9443 1994

The Integration Project for the JACK Environment

Amar Bouali
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
amar@cwi.nl

Stefania Gnesi Salvatore Larosa
IEI-CNR
via Santa Maria, 46 I-56100 Pisa, Italy
{gnesi,larosa}@iei.pi.cnr.it

Abstract

JACK, standing for *Just Another Concurrency Kit*, is a new environment integrating a set of verification tools, supported by a graphical interface offering facilities to use these tools separately or in combination. The environment proposes several functionalities for the design, analysis and verification of concurrent systems specified using process algebra. Tools exchange information through a text format called FC2. Users are able to graphically layout their specifications, that will be automatically converted into the FC2 format and then minimised with respect to various kinds of equivalences. A branching time and action based logic, ACTL, is used to describe the properties that the specification must satisfy, and model checking of ACTL formulae on the specification is performed in linear time. A translator from Natural Language to ACTL formulae is provided, in order to simplify the job to describe the specification properties by ACTL formulae. A description of the graphical interface is given together with its functionalities and the exchange format used by the tools. As an example of use of JACK, we present a small case study within JACK, that covers both verification of a software system and verification of its properties.

AMS Subject Classification (1991): 68N99, 68Q10, 68Q22, 68Q60.
CR Subject Classification (1991): D.2.2, D.2.4, D.2.6, F.1.1, F.3.1.
Keywords & Phrases: Specification tools, verification tools, reactive systems, integrated environment, methodology, bisimulation, model checking.
Note: The first author started this work while visiting the *Istituto di Elaborazione dell'Informazione* of CNR in Pisa and was funded by the European Research Consortium for Informatics and Mathematics, fellowship programme. The other authors were funded by the LAMBRUSCO project, supported by the CNR Finalized Project for Information Systems and Parallel Computation, unità operativa IEI – University La Sapienza, Rome.

1. Introduction

When the verification of system properties is an important issue, automatic tools are needed. Some verification environments are now available which can be used to verify properties of reactive systems, specified by means of terms belonging to process algebrae and modelled by means of finite state Labelled Transition Systems (automata), with respect to behavioural relations and logical properties [4, 9, 28, 38, 20, 18].

Recently a new verification environment, JACK, [10] was defined to deal with reactive sytems. The purpose of JACK is to provide a general environment that offers a series of functionalities, ranging from the specification of reactive systems to the verification of

behavioural and logical properties. It has been built beginning with a number of separately developed tools that have been successively integrated.

JACK covers much of the formal software development process, including the formalization of requirements [17], rewriting techniques [12], behavioural equivalence proofs [22, 14], graph transformations [19], logic verifications [11]. The logical properties are specified by an action based temporal logic, ACTL defined in [14], wich is highly suitable to express safety and liveness properties of reactive systems modelled by Labelled Transition Systems.

The main functionalities of JACK are summarized below.

1. NL2ACTL, a generator of ACTL formulae, that produces an ACTL formula starting from a natural language sentence: NL2ACTL helps in the formalization of informal systems requirements;

2. Behavioural equivalence verification by both rewriting on terms and on equivalence algorithms for finite state LTS: a process algebra term rewriting laboratory, CRLAB, that can be used to prove equivalences of terms through equational reasoning has been integrated in JACK together with the AUTO/MAUTO/AUTOGRAPH tool set developed at INRIA; these tools can be used for the automatisation of the specification and verification of process algebra terms in finite state cases. Specifications are given following the syntax of some process algebra (AUTO/MAUTO), or by means of a graphical tool (AUTOGRAPH) that allows the user to draw automata that are translated into process algebra terms. AUTO/MAUTO can then be used for formal verification and automata analysis.

3. Model checking of properties expressed in ACTL on a reactive system modelled by a finite state LTS: a linear time model checker, AMC, for the action based logic ACTL has been defined to prove the satisfiability of ACTL formulae and consequently the properties of systems.

4. Analysis of concurrent systems with respect to various concurrency semantics (interleaving, partial order, multiset...): a parametric tool, the PISATOOL, has been developed that allows the user to observe many different aspects of a distributed system, such as the temporal ordering of events and their causal and spatial relations.

The paper is organized as follows: Section 2 presents an overview of the syntax and semantics of the CCS/MEIJE process algebra used to describe reactive sytems in JACK, and of the ACTL logic. An introduction to formal verification tools is given in Section 3. Section 4 gives a description of the integration project and the graphical interface of the system. In Section 5 the JACK interface is described (with an overview of the components of the JACK environment). Finally, in Section 6, we give an example of use of the JACK environment.

2. BACKGROUND

2.1 Preliminaries

We first introduce the concept of Labelled Transition System, on which reactive systems are modelled and ACTL formulae are interpreted,

Definition 2.1 (Labelled Transition System) *A Labelled Transition System is a 4-tuple*
$\mathcal{A} = (Q, q_0, Act \cup \{\tau\}, R)$, *where:*

- *Q is a finite set of states. We let $q, r, s, \ldots$ range over states;*
- *q_0 is the initial state;*
- *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 $Act \cup \{\tau\}$;*
- *$R \subseteq Q \times Act \cup \{\tau\} \times Q$ is the transition relation.* □

Note 2.2 *For $A \subseteq Act$, we let A_τ denote the set $A \cup \{\tau\}$.*
For $A \subseteq Act_\tau$, we let $R_A(q)$ denote the set $\{q'$: there exists $\alpha \in A$ such that $(q, \alpha, q') \in R\}$. We will also use the action name, instead of the corresponding singleton denotation, as subscript. Moreover, we use $R(s)$ to denote $R_{Act_\tau}(s)$.
For $A, B \subseteq Act_\tau$, we let A/B denote the set $A - (A \cap B)$. Often, we simply write $q \xrightarrow{\alpha} q'$ for $(q, \alpha, q') \in R$ □

Definition 2.3 *Given an LTS $\mathcal{A} = (Q, q_0, Act \cup \{\tau\}, R)$, we have that:*

- *a path in $\mathcal{A}$ is a finite or infinite sequence $q_1, q_2, \ldots$ of states, such that $q_{i+1} \in R(q_i)$. The set of paths starting from a state q is denoted by $\Pi(q)$. We let $\sigma, \sigma' \ldots$ range over paths;*
- *a path $\sigma \in \Pi(q)$ is called maximal if it is infinite or if it is finite and its last state q' has no successor states (i.e. $R(q') = \emptyset$);*
- *if σ is infinite, then we define $|\sigma| = \omega$; if $\sigma = q_1, q_2, \ldots, q_n$, then we define $|\sigma| = n-1$. Moreover, if $|\sigma| \geq i - 1$, we will denote by $\sigma(i)$ the i^{th} state in the sequence.* □

2.2 Process Algebrae

Process algebrae are syntaxes for the description of parallel and communicating processes. Here we give a brief presentation of CCS/MEIJE process algebra, [1, 7], which is used by the JACK system for the description of reactive systems. For simplicity, we describe the subset of MEIJE that corresponds to the CCS process algebra, following R. Milner [31]. Moreover, we adopt the syntax used in the AUTO/MAUTO tools (see section 5) and restrict it to their rules to guarantee the satisfiability of some finitary conditions of the underlying semantic model, namely automata [15].

The MEIJE Syntax: The MEIJE syntax is based on a set of actions that processes can perform and on a set of operators expressing process behaviors and process combinations. The AUTO/MAUTO CCS/MEIJE syntax permits a two – layered design of *process terms*: the first level is related to the *sequential* regular process description; the second to a network of parallel sub-processes, supporting communication and action visibility filters.
The syntax starts from a set of labels *Act* as atomic signal names ranged over by alphanumeric strings; such names represent emitted signals if they are terminated by the "!" character, or received ones if they are terminated by "?". τ denotes the special action not belonging to *Act*, symbolising the unobservable action (to model internal process communications); we let $Act_\tau = Act \cup \{\tau\}$ to denote the full set of actions that a process can perform.
The following syntax is related to the definition of regular sequential processes: R denotes a sequential process, while a matches any element of Act_τ; X is a label denoting a process variable;

$$R \ ::= \ \textbf{stop} \ \mid \ a : R \ \mid \ R + R \ \mid \\ \textbf{let rec} \ \{X = R \ [\,\textbf{and} \ X = R\,]\ \} \ \textbf{in} \ X$$

Here, [...] denotes an optional and repeatable part of the syntax. We now explain the CCS/MEIJE sequential part semantics:

- **stop** is the process without behavior;
- $a : R$ is the action prefix operator;
- $X = R$ bounds the process variable X to the process R;
- $R + R$ is the non deterministic choice operator.
- The **let rec** construct allows recursive definitions of process variables.

The second level of process term definition is used to design networks of parallel sub components denoted here by P, where R is a sequential regular process:

$$P \quad ::= \quad R \quad | \quad P \parallel P \quad | \quad P \setminus a \quad | \quad P[a/b] \quad | \quad a * P \quad | \\ \textbf{let } \{X = P \; [\textbf{and } X = R \;]\; \} \; \textbf{in } X$$

- $\parallel$ is the parallel operator;
- $P \setminus a$ is the action restriction operator, meaning that a can only be performed within a communication;
- $P[a/b]$ is the substitution operator, renaming b into a.
- $a * P$ is the ticking operator, driving process P by performing action a simultaneously with any behavior of P. This means that any time that process P performs an action, then process $a * P$ performs in parallel both this action and action a.
- The **let** construct bounds non recursive definitions of process variables.

The Figure 0.1 shows the structural operational semantics of some CCS/MEIJE operators previously described, in terms of labelled transition systems [34]: note that the CCS/MEIJE parallel operator operational rules are those of the CCS parallel operator, whereas the MEIJE parallel operator, instead, has an additional rule allowing product of actions that are not necessarily co–names (i.e. $a!$ and $a?$).

Operator	Operational rules
$a : P$	$\dfrac{}{a : P \xrightarrow{a} P}$
$P + Q$	$\dfrac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \dfrac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$
$P \parallel Q$	$\dfrac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \qquad \dfrac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'} \qquad \dfrac{P \xrightarrow{a?} P', Q \xrightarrow{a!} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$

Figure 0.1: Operational semantics of some MEIJE operators

2.3 Bisimulation Semantics

We give here the definition of the bisimulation equivalence over LTSs, due to Park, [33].

Definition 2.1 (Bisimulation) *Given an LTS* $\mathcal{A} = (Q, q_0, Act \cup \{\tau\}, R)$*, a bisimulation over* $Q \times Q$ *is a binary symmetric relation* $\mathcal{R}$ *such that, for any* $(p, q) \in Q \times Q$*, we have* $p\mathcal{R}q$ *iff:*

$$\forall \alpha \in Act_\tau, (p \xrightarrow{\alpha} p' \Rightarrow (\exists q', q \xrightarrow{\alpha} q' \wedge p'\mathcal{R}q'))$$

We note by $\sim$ *the largest such relation, that is the union of all bisimulations definable over* S*.*

Observational bisimulations, first introduced by Milner [31], are defined through the notion of an unobservable action τ, considered as a silent step in the system behavior. To abstract unobservable moves during observation, we shall use the weak transition relation defined as follows:

Definition 2.2 (The Weak Relation)

$$\begin{array}{rll} & \stackrel{\varepsilon}{\Longrightarrow} & \stackrel{\text{def}}{=} (\xrightarrow{\tau})^\star \\ \forall a \in Act, & \stackrel{a}{\Longrightarrow} & \stackrel{\text{def}}{=} \stackrel{\varepsilon}{\Longrightarrow} \xrightarrow{a} \stackrel{\varepsilon}{\Longrightarrow} \end{array}$$

Weak bisimulation is then defined upon this relation, and called the weak $\longrightarrow$. We denote the largest one by $\approx$. Branching bisimulation, first introduced in [39] and denoted by $\sim_b$, is a particular observational bisimulation refining the notion of unobservable moves taking into account the internal nondeterminism. Its scheme is given by:

Definition 2.3 (Branching Bisimulation) *A branching bisimulation is a binary symmetric relation* $\mathcal{R} \subseteq Q \times Q$ *such that* $p\mathcal{R}q$ *iff:*

$$\begin{array}{ll} \forall \alpha \in Act_\tau, & (p \xrightarrow{\alpha} p' \Rightarrow \\ & (\alpha = \tau \wedge p'\mathcal{R}q) \text{ or } (\exists q_1, \ldots, q_n, \text{ such that} \\ & \qquad (1)\ q = q_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_n \xrightarrow{\alpha} q' \text{ and} \\ & \qquad (2)\ \forall i \in [1 \ldots n], p\mathcal{R}q_i, p'\mathcal{R}q') \end{array}$$

Bisimulations are used to minimise transition systems, as they define a minimal canonical form, and also to compare systems. Two systems are considered equivalent if and only if their respective initial states are related within some bisimulation over the product of the disjoint union of the sets of states of the two systems to compare. Verification with automata widely uses these concepts, for instance, to check partial properties of systems and to compare an implementation with a particular specification.

2.4 The ACTL Logic

We now define the temporal, action–based and pure branching time logic ACTL [14]; a logic of this type is appropriate to express properties of LTSs because its operators are based on actions. Moreover, ACTL is a temporal branching time logic, as it has both operators for quantification over paths and linear time operators. ACTL is a *pure* branching time logic because in its syntax each linear operator must be preceded by a branching one, and vice versa; this implies that only branching time properties are expressible. Furthermore, ACTL has an auxiliary calculus of actions embedded. Here below we present the action calculus:

Definition 2.4 (Action formulae syntax and semantics) *Given a set of observable actions Act, the language* $\mathcal{AF}(Act)$ *of the action formulae on Act is defined as follows:*

$$\chi ::= tt \mid b \mid \neg\chi \mid \chi \vee \chi$$

where b ranges over Act.
The satisfaction relation $\models$ for action formulae is defined as follows:

$a \models tt$ *always;*
$a \models b$ *iff* $a = b$;
$a \models \neg\chi$ *iff not* $a \models \chi$;
$a \models \chi \vee \chi'$ *iff* $a \models \chi$ *or* $a \models \chi'$.

□

From now on, we let ff abbreviate the action formula $\neg tt$ and $\chi \wedge \chi'$ abbreviate the action formula $\neg(\neg\chi \vee \neg\chi')$.
Given an action formula χ, the set of the actions satisfying χ can be characterized as follows.

Definition 2.5 ($\kappa : \mathcal{AF}(Act) \to 2^{Act}$) *We define the function $\kappa : \mathcal{AF}(Act) \to 2^{Act}$ as follows:*

- $\kappa(tt) = Act$;
- $\kappa(b) = \{b\}$;
- $\kappa(\neg\chi) = Act/\kappa(\chi)$;
- $\kappa(\chi \vee \chi') = \kappa(\chi) \cup \kappa(\chi')$. □

Theorem 2.6 *Let $\chi \in \mathcal{AF}(Act)$; then $\kappa(\chi) = \{a \in Act : a \models \chi\}$.*

Sketch of the proof: The proof of this Theorem can be given by structural induction on χ. □

Definition 2.7 (ACTL syntax) *ACTL is a branching time temporal logic of state formulae (denoted by ϕ), in which a path quantifier prefixes an arbitrary path formula (denoted by π). The syntax of ACTL formulae is given by the grammar below:*

$$\begin{aligned} \phi &::= tt \mid \phi \wedge \phi \mid \neg\phi \mid E\pi \mid A\pi \\ \pi &::= X_\chi\phi \mid X_\tau\phi \mid \phi\ {}_\chi U\ \phi \mid \phi\ {}_\chi U_{\chi'}\ \phi \end{aligned}$$

where χ, χ' range over action formulae, E and A are path quantifiers, and X and U are next and until operators respectively. □

We now describe the conditions under which a state s (a path σ) of an LTS satisfies an ACTL formula ϕ (a path formula π), written $s \models \phi$ ($\sigma \models \pi$).

Definition 2.8 (ACTL semantics) *The satisfaction relation for ACTL formulae is defined in the following way:*

$s \models tt$ *always;*
$s \models \phi \wedge \phi'$ *iff* $s \models \phi$ *and* $s \models \phi'$;
$s \models \neg\phi$ *iff not* $s \models \phi$;
$s \models E\pi$ *iff there exists a path* $\sigma \in \Pi(s)$ *such that* $\sigma \models \pi$;
$s \models A\pi$ *iff for all maximal paths* $\sigma \in \Pi(s)$, $\sigma \models \pi$;
$\sigma \models X_\chi\phi$ *iff* $|\sigma| \geq 1$ *and* $\sigma(2) \in R_{\kappa(\chi)}(\sigma(1))$ *and* $\sigma(2) \models \phi$;

$\sigma \models X_\tau \phi$ *iff* $|\sigma| \geq 1$ *and* $\sigma(2) \in R_{\{\tau\}}(\sigma(1))$ *and* $\sigma(2) \models \phi$;
$\sigma \models \phi \, {}_\chi U \phi'$ *iff* *there exists* $i \geq 1$ *such that* $\sigma(i) \models \phi'$,
and for all $1 \leq j \leq i-1$:
$\sigma(j) \models \phi$ *and* $\sigma(j+1) \in R_{\kappa(\chi)_\tau}(\sigma(j))$;
$\sigma \models \phi \, {}_\chi U_{\chi'} \phi'$ *iff* *there exists* $i \geq 2$ *such that* $\sigma(i) \models \phi'$, $\sigma(i-1) \models \phi$,
$\sigma(i) \in R_{\kappa(\chi')}(\sigma(i-1))$ *and for all* $1 \leq j \leq i-2$:
$\sigma(j+1) \in R_{\kappa(\chi)_\tau}(\sigma(j))$.

□

Several useful modalities can be defined, starting from the basic ones. We write:

- $E\widetilde{X}_\chi \phi$ for $\neg AX_\chi \neg\phi$, and $A\widetilde{X}_\chi \phi$ for $\neg EX_\chi \neg\phi$; these are called the *weak next* operators.
- $EF\phi$ 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.
- $< \chi > \phi$ for $E(tt_{ff} U_\chi \phi)$, if $\chi \neq ff$;
- $<> \phi$, for $E(tt_{ff} U \phi)$,
- $[\chi]\phi$ for $\neg < \chi > \neg\phi$;
- $[\,]\phi$ for $\neg <> \neg\phi$.

3. Formal Verification Tools

Formal verification for reactive systems usually consists of two important stages:

1. the system design specification stage;
2. the properties checking stage.

The tools have been built following this scheme. Here below we thus describe specification tools and properties checking tools.

3.1 Specification Tools

These tools offer functionalities to build a process specification. They are often process algebra syntax compilers and make it possible to compositionally design a process term, by first specifying the sub terms separately and then putting just the sub term process names in a higher term. This can be done in two ways:

- by allowing the designer to enter a specification in a textual form;
- by offering sophisticated graphical procedures to construct a process specification. The tool then automatically translates the drawings into a process term.

At this level, other functionalities can be offered such as:

- term rewriting;
- finite–state conditions checking.

3.2 Property Checking Tools

Logics have been intensively used to check the behavioural and logical properties of programs. In particular, in the concurrent systems area, temporal (or modal) logics have been introduced in order to be able to express properties in a logical language that permits system behavior to be described and to verify these properties on some system model (e.g. a Kripke structure or a transition system) [16, 21, 2, 26]. These logics are such that if a property holds in a system, then that system is a *model* for the formula representing the property.

Another approach to system verification has been studied. This approach is based on automata observations and analysis: properties are also expressed as automata, and equivalence notions such as bisimulation are used to check whether a given system possesses some (un)desired property or not.

In both cases, methods automatization has played an important role. Due to computability problems, automatic methods can only be proposed for finitely represented systems. Model checking is the name for automatized verification with logics. For the automata based methods, a set of algorithms on graphs, such as bisimulation equivalence checking, forms the kernel of verification tools based on transition systems.

In the next section, we will present the JACK system and the "glue" of the tools integration project: the FC2 automata description format.

4. The JACK Integration Project

The idea behind the JACK environment was to put together different specification and verification tools developed separately at two research sites: IEI–CNR in Italy and INRIA in France.

A first experiment in building verification tools, starting from existing ones, is described in [11]. Following this first attempt, we have developed an environment based on the links proposed in [11], and on new links that exploit the FC2 format [27] (see Figure 0.2). We had the following objectives:

- to provide an environment in which a user can choose between several verification tools; this environment will have a simple, user–friendly graphic interface;
- 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. In this sense, the FC2 format acts as a system "glue".

Now, we briefly introduce the tools that are used in the JACK system, dividing them into *specification* tools and *verification* tools.

4.1 Specification Tools

AUTOGRAPH **(INRIA):** This is a graphic specification tool for the design of parallel and communicating processes [35] that provides functionalities for a compositional development of a specification. As a general rule, a window is a process specification. Process construction starts from automata which represent single sequential processes. Processes surrounded by boxes are said to be *networks* and are used to hide information on low–level details of a specification and to represent parallel composition. If two networks are drawn at the same level, they can synchronise the signals they emit, and thus representing communicating processes. There is no need for a network to be fully specified, it could simply be an empty box. It is sufficient to specify its external synchronisation signals, and this permits a top–down approach in the AUTOGRAPH specification process.

Another feature of AUTOGRAPH is the automata interactive exploration: starting from an initial state, an user can just unfold the paths (s)he is interested in. AUTOGRAPH provides

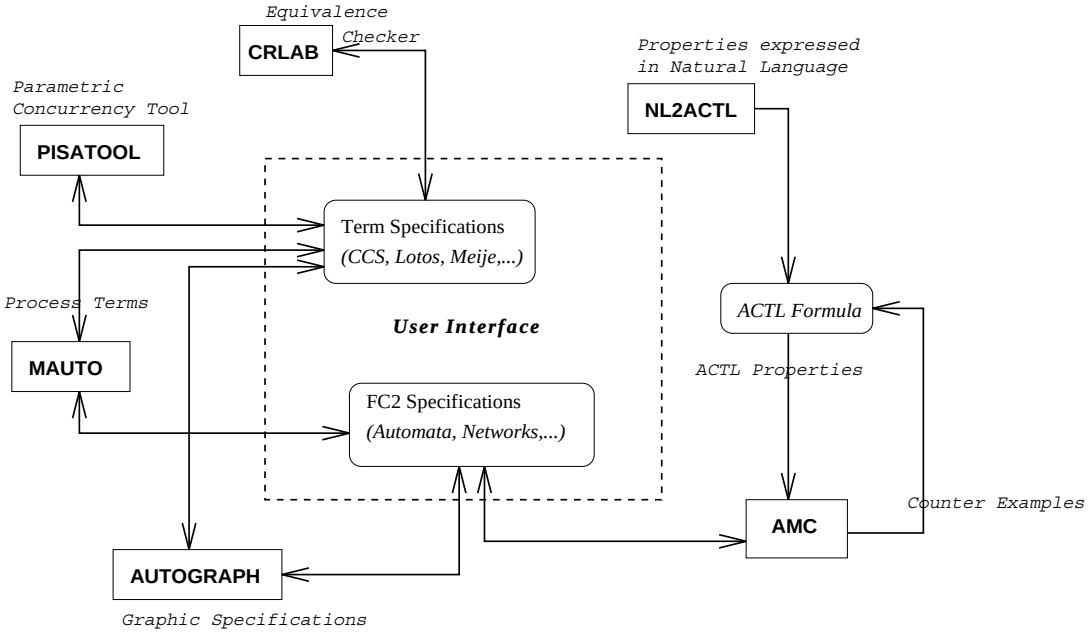

Figure 0.2: The integration project

several output formats in which a graphical specification can be translated, including FC2, the MAUTO syntax for terms and Postscript.

AUTO/MAUTO **(INRIA):** MAUTO [15] (a generalisation of AUTO) is a tool for both the specification and the verification for concurrent systems described by process algebrae. Actually MAUTO can deal with MEIJE [1], CCS [31], LOTOS [25] and ESTEREL [3] process algebrae (while AUTO was just designed for the first two).

MAUTO is a command interpreter manipulating two classes of objects: the class of process algebrae specifications and that of specification automata. Each algebraically specified process is called *term*. During the specification stage, the user deals with the terms in the MAUTO environment; terms are parsed and syntax errors are reported. In the parsing stage it is detected whether terms represent finite state systems or not (just finite state systems can be studied). Sufficient syntactic conditions are studied in [29]. The user can then translate a term (the main specification term, or another one) into either the FC2 format, or the AUTOGRAPH format to graphically view the specification, or into other formats suitable as inputs for various other tools, to carry on with the next specification and verification phases. Many translation functions from algebraic objects into automata are also available, so that the user can enter the MAUTO verification framework. This will be descibed in the Verification Tools section.

NL2ACTL**, an automatic translator from Natural Language to Temporal Logic (IEI–CNR):** NL2ACTL, a prototype translator from Natural Language expressions to Temporal Logic formulae, has been developed and integrated in JACK, in order to test the use of Natural Language in a friendly interface to make the expression of properties in the logic easier for the user. NL2ACTL deals with sentences expressing the occurrence of actions performed by reactive systems. A precise semantic meaning in terms of ACTL formulae is associated with each sentence. If this semantics is not ambiguous, an immediate ACTL translation is provided; otherwise, an dialog with the user is started in order to solve the ambiguity.

In fact, in our experience in the specification and verification of properties using temporal logics, we have found that imprecisions frequently occur in the passage from the informal

expression of properties in natural language to their formulation in temporal logics, due to the inherent ambiguities in many natural language expressions. We have thus attempted to identify a solution to this problem in the current state of the art in Natural Language Processing, looking for a formal method, that can help to generate logic formulae, which correspond as closely as possible to the interpretations an ACTL expert would give of the informal requirements.
NL2ACTL has been developed using a general development environment, PGDE, aimed at the construction, testing and debugging of natural language grammars and dictionaries, which permits us to build an application recognizing natural language sentences and producing their semantics [30].

4.2 Verification Tools

AUTO/MAUTO **(INRIA):** As we stated above, AUTO/MAUTO can also be used in verification, because it permits automata to br reduced in various ways. Commands allow process algebra term verification of partial properties based on observations of underlying automata.
Further automata analysis is available, such as abstraction, minimisation, and diagnostics on equivalence failure.
Using MAUTO in the verification phase, the user can manipulate the automata with respect to abstraction criteria, or can perform their minimisation and/or comparison with respect to behaviour, or can produce diagnostics of equivalence checkings.
The verification principle can be generalised as follows. First define an implementation of a system as a process algebra term involving several communicating processes running in parallel. Then translate the term into a global automaton capturing all its possible computations. For partial property verification, define properties with abstraction criteria[1]: abstracting the global automaton helps to verify whether the expressed property is satified by the implementation. Users can also define a specification with respect to the desired external behavior, using abstracted actions. The global system is abstracted to meet the implementation: the answer is given by bisimulation checking between the implementation and the specification.
The ACTL Model Checker (IEI–CNR): AMC, the model checker for ACTL logic formulae, permits the validity of an ACTL formula to be verified on a labelled transition system in a linear time. Whenever an ACTL formula φ does not hold, the model checker produces a path from the LTS (called a counterexample) given in input, which falsifies φ, and provides useful information on how to modify the LTS to satisfy the formula φ.
This model checker allows the satisfiability of ACTL formulae on the model of a reactive system to be verified. Requirements can also be maintained and enhanced, on the basis of the results of the verification stage: on the basis of the concrete model of the system and the formalization of requirements (a list of temporal logic formulae), the verification of the latter on the former - by means of the model checker - may provide useful information. Model checking for ACTL can be performed with time complexity $\mathcal{O}((|Q| + |\longrightarrow|) \times |\phi|)$ where ϕ is the ACTL formula to be checked on an LTS that has $|Q|$ states and $|\longrightarrow|$ arcs.
CRLAB **(IEI–CNR):** CRLAB is a system based on rewriting strategies [12, 13]. The input, which is supplied to the system interactively, can be LOTOS [25] or CCS [31] specifications. It is possible to simulate the operational behaviour of a process as well as automatically prove the bisimulation equivalence of two finite processes. The bisimulation equivalences considered are the observational, trace, and branching ones. It is also possible

[1] Intuitively, an abstraction criterion is a collection of abstract actions, which are rational expressions over the concrete set of actions; this is a way to express path properties with all the expressive power of rational expressions.

to define user-driven proof strategies, although no facility for this is explicitly available. However, a strategy to prove the equivalence of two CCS processes by transforming one process term into the other by means of axiomatic transformation is provided.

PISATOOL **(IEI–CNR):** The PISATOOL [22, 23] is a system for specification verification that accepts specifications written in CCS and is parametric with respect to the properties the user wants to study. This means that the user can choose a process observation function from a library of functions.

The PISATOOL represents the processes internally by the so-called *extended transition systems* [24], i.e., transition systems labelled on nodes by regular expressions [8, 37, 36] that encode all the computations leading to the node from the starting state.

After the tool has converted a process into this type of internal representation, the user is able to select an observation function to study interleaving, causality, locality and so on; The process equivalences are checked through the algorithm of [32] and the implemented equivalence observations are the strong, weak, branching and trace ones. A library of observations for studying truly concurrent aspects of distributed systems is provided; moreover, expert users can define their own observations. The tool is equipped with a window-based interface that makes the observation tasks easy.

Companion Tools: FCTOOL/HOGGAR **(INRIA):** These tools offer bisimulation minimisation procedures for systems described as a single transition system (FCTOOL), or networks of transition systems [5, 6]. HOGGAR is actually interfaced with MAUTO which calls it whenever a bisimulation minimisation has to be performed on a single transition system. The interface uses the FC2 format (see below). FCTOOL works with a variety of static networks of parallel and communicating processes using symbolic techniques: it allows global system computation and bisimulation minimisation of such networks. The algorithms are based on a symbolic representation of global transition systems by means of a *Binary Decision Diagram* (BDD), allowing the analysis of "very" large systems with a reasonable cost in terms of time and space. These two tools currently deal with strong, weak and branching bisimulation, but other equivalences and preorders can easily be added.

4.3 The FC2 *format*

Each tool is presumed to have its own input and output format. The FC2 format is a common format adopted by many verification tools to describe input and output data. Its main purpose is to enable communication between tools in a standardized way: for instance when linking specification tools to property checking tools. Historically, the FC2 syntax started with the efforts of some verification tool designers to be able to estabilish links between their tools. This cooperation was based on the simplest exchangeable objects, namely automata. So, the original provided syntax described these objects. It has now been adopted by several tool makers and has been enriched in order to generalise the kind of objects that can be described. The class of objects ranges over networks of *transducers* covering most kinds of input/output objects that can be given or generated by a verification tool in the domain of finite state concurrent systems. The format is organised as follows:

FC2 **Objects and Labels:** The FC2 objects are: vertices, edges and nets. A net is a graph containing a finite number of vertices and edges. Each object has a *label*. Objects are presented in tables. An FC2 file is thus a table of nets; each net has a table of vertice; each vertice has a table of edges.

Each object has a *label*. A label is a record of informations, each being preceded by a field name. The field names are: `struct`, `behav`, `logic`, `hook`. These fields are used to assign semantical information to objects. For instance, the field `behav` of an edge stands for the action label of the underlying transition. Each piece of information is a string or is composed using a set of predefined operators of various arity to express simple set

constructs.
Fc2 **Nets:** Nets in the Fc2 format can be:

1. a single LTS; the net is just a table of vertices representing the set of states of the LTS, and for each state vertex the set of transitions starting from that state form the edge table. The minimum information label is the action name of each transition given through the field `behav`;

2. a synchronised vector of nets; in the field `struct` of the net, the structure of the net is given, listing the set of sub-nets put in parallel and composing the current net. The vertex table is reduced to an unique element (state) from which all synchronisation constraints between the sub-nets are expressed as edges from this state to itself, having as label the synchronisation action set;

3. a transducer; this is a generalisation of the synchronised vector of nets, where the net has several states; from each state, a specific set of synchronisation constraints are given, reaching other states;

The format is more concrete than process algebra. Moreover, it enables the description of different views of parallelism and synchronisation. In fact, all finite state systems that can be represented by process algebrae are also representable by the Fc2 format.

5. The JACK Interface

In order to test our integration ideas, we have developed an user – friendly interface (Figure 0.3) to the different tools composing the JACK system. The interface is designed in an object-oriented style where objects are either term specification files or Fc2 description files.
This interface has been developed using the Tcl/Tk language facilities for the creation and manipulation of graphical widgets linked to a interactive function calls mechanism through mouse and keyboard events.
The interface is basically composed by two objects area, one for term files and one for Fc2 files.

5.1 Term Files Manipulation

Terms are given in a textual form with the syntax adopted by Mauto[2]. In JACK there is a term specification area management (see Figure 0.3), that is a list of files containing term specifications. A set of commands is associated with this area:

- term file management commands for loading/removing a file path name from the path name list, for viewing the contain of a preselected file;

- "shortcut" commands allowing the user to send a specific function of a particular tool. For example, if one wants to get the underlying automaton of a given term, a "shortcut" operation is available in the interface, calling Mauto in a batch mode, so not visible to the user, getting directly the result, without entering in a full session;

- commands starting a tool session initialised with some preselected term file: further work is then available within that session as a normal use of the called tool. This feature is provided for PisaTool, Crlab, Mauto.

[2]Not all the integrated tools dealing with terms currently accept this syntax. However, we shall provide translation functions from the Mauto syntax to these other syntaxes

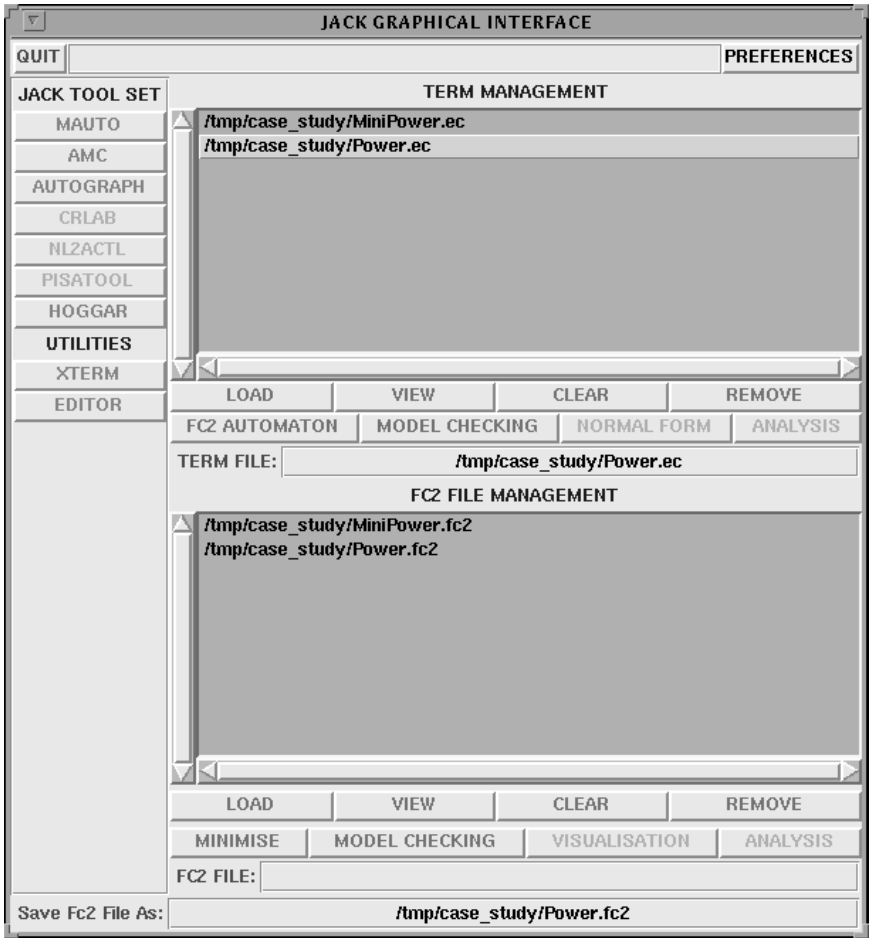

Figure 0.3: The JACK general control panel.

5.2 Fc*2 Files Manipulation*

Fc2 files represent essentially either a single automaton or a network of automata. When automata are translated into the Fc2format, they can be submitted to the various tools of the JACK system. Like for terms, JACK provides a Fc2 file management area (see Figure 0.3), that has associated the following commands:

- Fc2 file management commands to load/remove a file path name from the list and to view the contain of a preselected file;
- shortcut commands, which are basically abstraction/reduction of the (global) automaton: a graphical panel offers to the user different choices regarding bisimulation equivalences and a simplified edition area to set up abstraction criteria. When a choice is made, then either Hoggar machinery is called if no abstraction criteria is given, or else Mauto is called which anyway calls Hoggar to perform efficiently minimisations. Of course tool executions are not visible from the user, who deals just with the output Fc2 files;
- commands to start tool sessions initialised with a preselected Fc2 file. It is the case for AMC, Mauto, Hoggar, Autograph.

5.3 Other Integrated Graphical Interfaces

In JACK, some of the integrated tools have their own graphical interface. Some of them are displayed in section 6.

It is the case for AUTOGRAPH which has a menu for the selection of its functions and manipulates graphical objects through a window hierarchy.
It is also the case for HOGGAR which has a small interface for the selection of FC2 files and options before processing.
We have built, during the development of JACK, also a graphical interface for AMC: this interface allows interactive session of the model checker making ist use easier. For instance, an automatic command of select/load FC2 files is included, avoiding the typing of commands to the user. The same for ACTL formulae, which are saved in a history list after submission. The history list can be displayed and the user can select one ot its element to re-apply it or to slightly modify it and apply the new one. Other graphical supports are available for formulae files and formulae shortcuts management.

5.4 Concluding Remarks
Following [11], JACK offers natural strategies using some of the differents tools it contains. The specification problem is made easier by AUTOGRAPH. Designing graphically networks of communicating processes save effort and is less error-prone than writing terms by hand. Terms are then automatically generated from specifications. AUTOGRAPH provides also the translation of such a graphical design into an FC2 file.
The FC2 file can be submitted to MAUTO, AMC, or FCTOOL/HOGGAR. The way to do is submitting the file for transition system computation, abstraction, minimisation to offer a reduced model for model checking and/or further automata analysis.
When results can be saved as FC2 files, then graphical display can be performed within AUTOGRAPH.

6. A CASE STUDY WITHIN JACK

In this section we will give an example of use of the JACK environment to specify and test the behaviour of a part of a reactive system, involving several functionalities from different tools. The case study we chose is very simple, but is a real one: it is a piece of a hydro power plant control system specification. Now, we will give a brief description of such a plant.

6.1 Presentation of the Case Study
The plant is composed by a *water basin*, from where the water flows into a number of *energy production engines*, that convert the water mechanical energy to electrical energy; Each engine has an *embedded controller*, that can detect if the engine is working well, or if a failure happens. A general software *control system* monitors all the plant components and sends commands to the engines controllers.
Engine failures are dued to different reasons. For example, suppose that a piece of an engine gets broken; the engine controller is able to detect this event, and it can just signal the failure, while it waits for someone repairs the broken piece. This is an example of a *persistent* failure: the controller cannot repair it, it just waits for the signal the piece is repaired, and then it continues to drive the engine. In another situation it may happen that engine's temperature overcomes its security level; this means that the engine should be put offline, until its returns under that level. In such a case the controller will block the engine, waiting for the decreasing of the temperature: the engine is not broken, but it cannot be driven because it will be off-line for a period of time.
The two examples above let us understand that two kind of failures exist: one kind is related to engine failure, and cannot be automatically repaired; in this case we say the engine has a *lock*. The second one is related to failures the controller can react to, and this means that after a period the engine will return able to be driven; in this case we say that the engine is *off–line*.

The engines can produce a variable quantity of electrical power, in relation with the increasing/decreasing of the water flow, but the flow variation does not automatically set the power output level of the engines: they explicitly need commands from the control system to put themselves in states of greater/less power production. We assume each engine has just three productivity states to avoid a too complex specification. The engine's controller can receive the following commands from the control system:

- `inc/dec`, to increase or decrease the energy production;
- `start/stop`, to start or stop the engine;
- `test`, to send to the control system a signal about the engine *status*. This signal can be one of the following:
  - `pone, ptwo, pthree`: the current power production level of the engine is one of pone, ptwo, pthree;
  - `stopped`: the engine is stopped;
  - `offline`: the engine will be off–line for a period;
  - `locked`: the engine is broken.

When the embedded controller receives an *engine command*, i.e. one of `start`, `stop`, `inc`, `dec`, it must return an answer message to the control system; moreover, the controller must send a `begin` signal *when* it starts to execute an engine command[3].
The generic synchronisation scheme between the control system and an engine controller is:

1. the control system sends an *engine commend* to the engine's controller;
2. when the controller starts the execution of such a command, then it sends `begin` to the control system;
3. when the controller detects the engine command termination, then it sends an *answer* to the control system.

Such an answer can be:

- `ok`: the *engine command* was successfully executed;
- `notdone`: the command cannot be executed; for security reasons there are situations in which the controller cannot obey to `inc/dec` commands, if it wants to let the engine hardware safe.

Moreover, the controller can send asynchronous signals to the control system; these ones are:

- `ko`: there was a failure in the engine. It could have a lock or could be off–line;
- `online`: the engine is again on line;
- `lockfixed`: someone has repaired the engine.

In the following we will specify the general functionalities of an engine's embedded controller in a formal way. What we said above should be sufficient to make clear the understanding of the specification.

[3]Notice that `test` does not change the engine status: it simply reads it, so that the engine controller can immediately answer.

The AUTOGRAPH **Approach:** The first step in the Jack specification development system may be a graphical one: we use AUTOGRAPH to draw the automaton that characterize the desidered behaviour of the controller[4]. The Figure 0.4 shows the output of the AUTOGRAPH session. Now we give some explanation about the states and edges of the pictured automaton.

The edge labels terminated by a "`!`" represent signals that our controller will *output*, while those ones terminated by a "`?`" are controller *inputs*.

Initially, the engine is stopped and its controller is in the initial state S. Notice that if the controller is in the state S and it receives a `test` command, it will respond by sending a `stopped` signal. To represent this synchronisation with the control system a dummy state is needed to input `test` and output `stopped`. We did not want to give a name to dummy states: AUTOGRAPH fills no–named states with auto–generated names.

`G1, G2, and G3` (Generating states) correspond to the three engine power production levels; When the controller is in `G1` and the control system asks for decreasing the power, the controller does nothing but producing synchronisation messages, as in the user requirements; it behaves in the same way when it is in `G3` and receives `inc`.

`H12, H21, H23` and `H32` are intermediate states; the controller is in the state `Hxy` when has received the order to change the power production level from the x level to the y level. Notice that if the increasing/decreasing cannot be made, the controller maintains its current production level and sends a `notdone` to the control system.

It is also possible to switch off an engine when it is in one of the `Gx` states: the controller can receive an `halt` command, and then goes in an intermediate state `gTs` (Generating To Stopped) that represents the period needed to stop the engine (in general, this is not an immediate operation). In the case of a `start` command there is a symmetrical behaviour.

The signals `noise` and `lock` do not come from the control system: they represent events that (respectively) obly the controller to put the engine off-line or to declare the engine locked. When `noise` is received, the controller waits for the engine stops — this is represented by the `gTtb` (Generating To Timed Block) state — and then says the control system that the engine is `ko`; then, it goes in the `Tb` (Timed Block) state, waits for the noise conditions will disappear, and finally it signals the engine is `online` again, returning to the initial state `S`.

The signal `lock` is managed in a similar way (in this case, `gTpb` means "Generating To Persistent Block"); when the controller is in the `Pb` (Persistent Block) state, it waits for someone that repairs the engine and signals to the controller the engine is `OK` again (this event is represented by the `repaired` signal).

We label the states `A, B, C, D, E` because we will refer to them in the following.

Working with MAUTO**:** When we have finished to use AUTOGRAPHto layout graphically the specification, we get three different outputs from the tool: a PostScript one (Figure 0.4), an FC2 description of the automaton and a CCS/MEIJE specification file (suitable as MAUTO input).

Notice that another way we can take to start the formal specification job is directly writing the controller specification in the CCS/MEIJE language and using it as a MAUTO input. Such a textual specification is shown in Figure 0.5. It describes the same automaton drawn by AUTOGRAPH, but it is different from the automato CCS/MEIJE description o produced by such a tool. The difference are owing to AUTOGRAPH uses only single transitions in its CCS/MEIJE output, while the specification in Figure 0.5 sometimes uses sequences of transitions, avoiding to explicitely give a name to dummy states. The CCS/MEIJE

[4]This way of starting the specification process has been found particulary convenient in this case, because the informal specification of the system included already some "state–machine" descriptions.

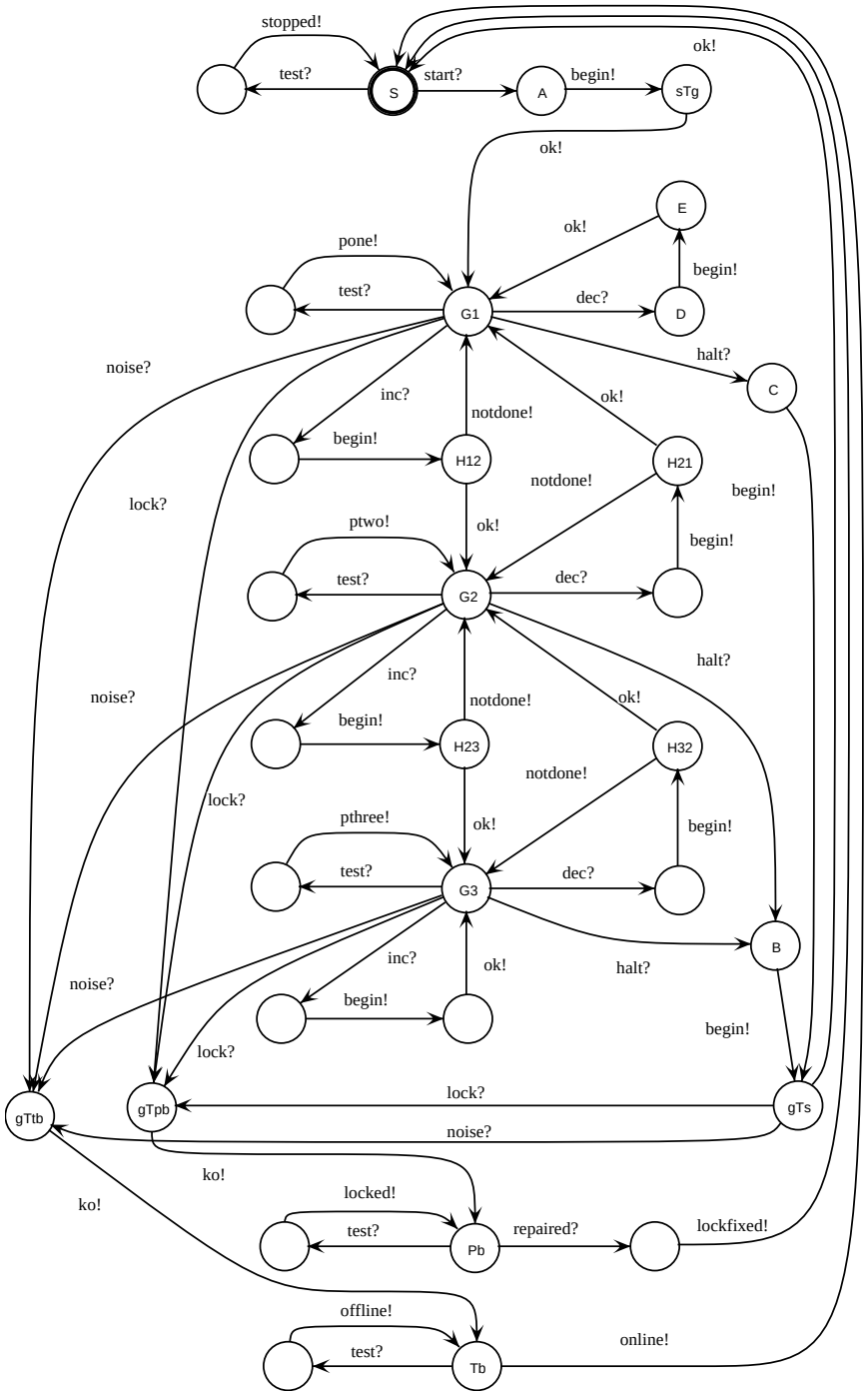

Figure 0.4: The AUTOGRAPH controller specification.

```
parse PowerEngine =
let rec {
                S = start? : (begin! : sTg) +
                    test? : (stopped! : S)
        and
              sTg = ok! : G1
        and
               G1 = noise? : gTtb +
                    lock? : gTpb +
                    dec? : (begin! : (ok! : G1)) +
                    inc? : (begin! : H12) +
                    halt? : (begin! : gTs) +
                    test? : (pone! : G1)
        and
              H12 = notdone! : G1 +
                    ok! : G2
        and
              H21 = ok! : G1 +
                    notdone! : G2
        and
               G2 = noise? : gTtb +
                    lock? : gTpb +
                    inc? : (begin! : H23) +
                    dec? : (begin! : H21) +
                    halt? : (begin! : gTs) +
                    test? : (ptwo! : G2)
        and
              H23 = notdone! : G2 +
                    ok! : G3
        and
              H32 = ok! : G2 +
                    notdone! : G3
        and
               G3 = noise? : gTtb +
                    lock? : gTbp +
                    dec? : (begin! : H32) +
                    inc? : (begin! : (ok! : G3)) +
                    halt? : (begin! : gTs) +
                    test? : (pthree! : G3)
        and
             gTtb = ko! : Tb
        and
             gTpb = ko! : Pb
        and
              gTs = ok! : S +
                    noise? : gTtb +
                    lock? : gTpb
        and
               Tb = online! : S +
                    test? : (offline! : Tb)
        and
               Pb = repaired? : (lockfixed! : S) +
                    test? : (locked! : Pb)
} in S;
```

Figure 0.5: A textual controller specification.

```
audit "Power";

add-search-path "/tmp/case_study";

calculus "meije";

set ccs = true;
set debug-algo = true;
set debug-cycle = true;
set debug-gc = true;

load "Power";

set automaton = mini(PowerEngine);

write "MiniPower" automaton autograph;
write "MiniPower" automaton fc2;

close;

end;
```

Figure 0.6: A MAUTO script created by JACK.

AUTOGRAPH output, saved in the file `Power.ec`, will be used in the rest of this case study. Now we are going to use MAUTO .
We can start now a MAUTO session from JACKto study some bisimulation properties of our automaton, and we select the CCS/MEIJE calculus from JACK's MAUTO configuration panel. A MAUTO script to transform the `Power.ec` specification into an FC2 one is shown in Figure 0.6:
JACK automatically generates such a script, in agreement with the user selections in the MAUTO control panel, shown in Figure 0.7; then, JACK pipes it into MAUTO, and lets the user eventually run MAUTO interactively. Notice that the setting of the flag `ccs` replaces the MEIJE semantic of the || operator with that one of CCS.
Let's explain the script of Figure 0.6. The second step of our specification job is to study our automaton, trying to reduce it in a simpler form, if it is possible. We use the MAUTO function `mini`, that tries to reduce the automaton associated to the PowerEngine process into a newer, simpler automaton that is strongly equivalent (and then observational equivalent, too) to the original one. We need to simplify the automaton because

- we can perform a better visual debugging by a second AUTOGRAPH session if we have less states to handle. To do it, we get a new AUTOGRAPH–loadable automaton using the first `write` command;
- the task to test ACTL formulae over it will be faster. The AMC tool inputs FC2 files, so using the last `write` command in the script we get the file `MiniPower.fc2` that contains the minimized automaton.

The results of the second AUTOGRAPH session are visible in the Figure 0.8.
It is possible to see that the state `D` collapsed with the state `A`, and the state `E` with the state `sTg`; moreover, notice that the state `B` was not necessary: it could be cut, so that

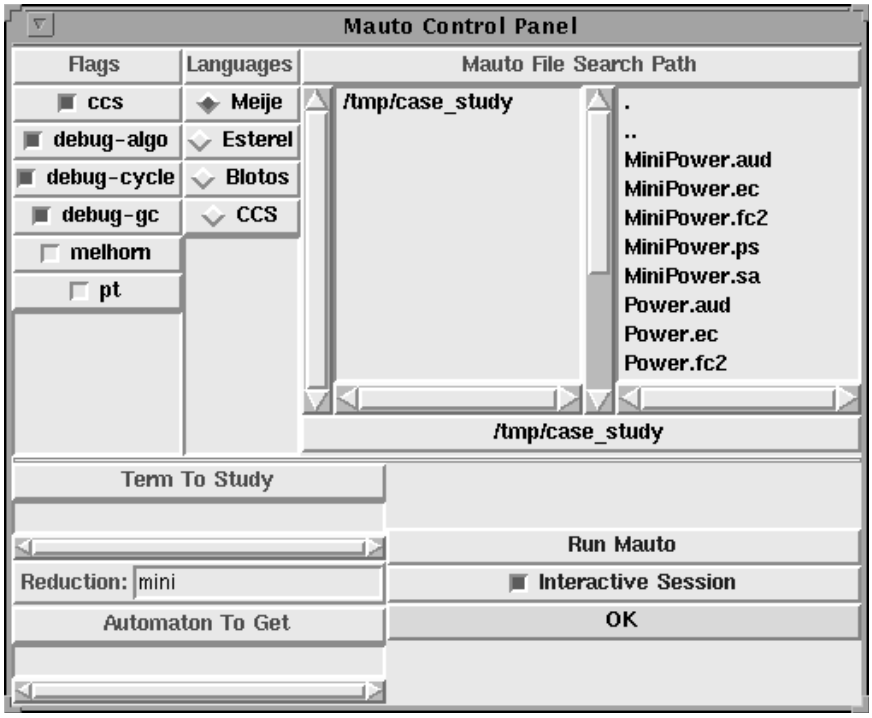

Figure 0.7: The MAUTO Control Panel.

the edge that before was entering it now is entering the state C. You can notice the way used by AUTOGRAPH to label dummy states.
Thus, after having reduced the automaton, we are ready to study it using the tools NL2ACTL and AMC, that let us write and check ACTL formulae, respectively.
Describing properties with NL2ACTL**:** There are two ways to handle formulae. The first one is to use NL2ACTL to write in natural (English) language the property we wish to check; NL2ACTL will give us the related ACTL formula, and we will be able to put it into AMC. The second one is to directly use the AMC formulae editor to write the formulae related to our properties, avoiding the usage of NL2ACTL. A logic–skilled user probably will prefer the latter way, while, if an user is not sure about the way to write the properties he keeps in mind, (s)he could use the interactive and question–based tool NL2ACTL.
In Figure 0.9 there is an example of use of NL2ACTL. Here we wish to verify that if it is possible that our system crashes; so we write that "if start is performed, then lock is performed", to mean that if we switch on the engine, then it could get broken. The above phrase does not express a *possibility*, but a *sure event*. To avoid natural language interpretation mistakes, the tool asks the user to specify if he (her) whishes his (her) phrase refers to *all* the possible ways in which the engine runs, or *at least* one of them. Then (Figure 0.10) the user has to specify if the action in the phrase can be performed after some other visible actions (*eventually*), or just after some *unobservable* actions (*soon*).
the NL2ACTL translation of the above phrase is shown in Figure 0.11.
Using the AMC **model checker:** Now, we are able to run AMC and check our formulae. The Figure 0.12 shows a session with our model checker; we input a formula to be checked and AMC displays if it is true or false. The result of checking the NL2ACTL formula is obviously true; suppose now we want to check if the synchronisation between the control system and the controller is consistent. In particular, we wish to verify that if the controller receives an engine command and starts the command execution, after such execution the

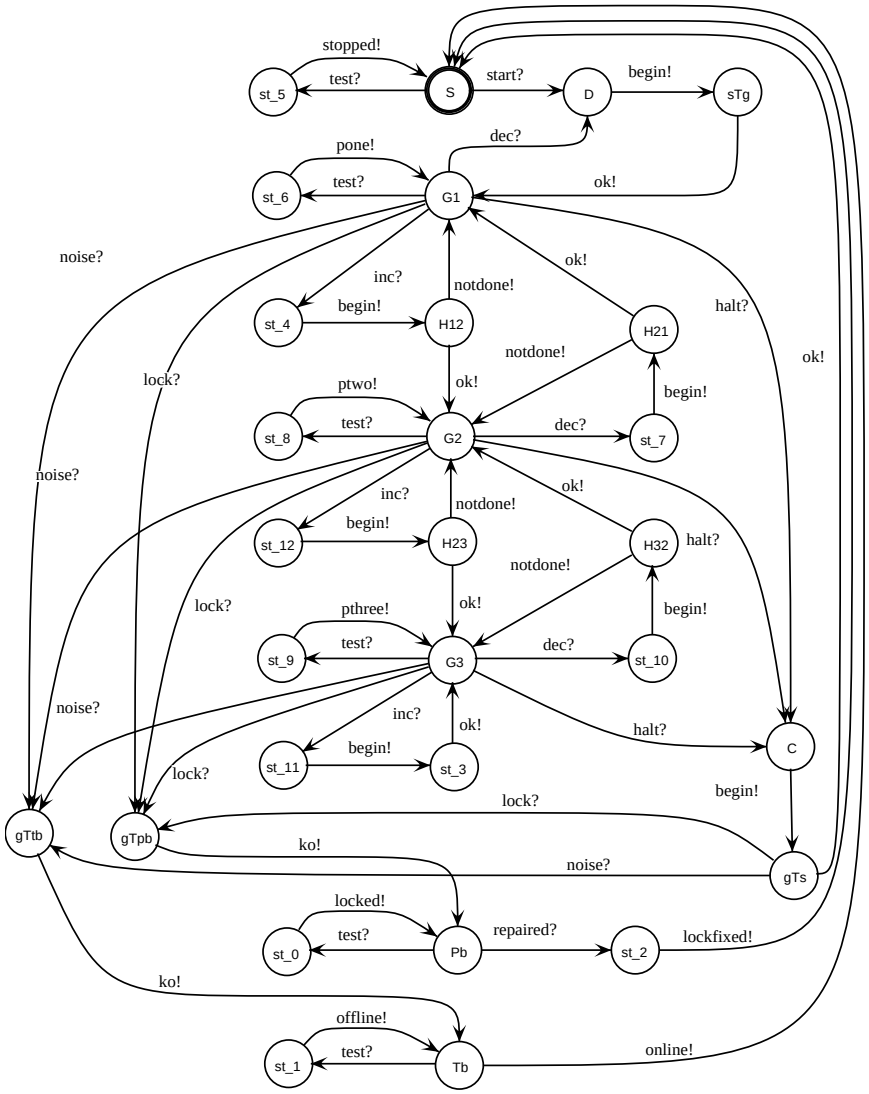

Figure 0.8: The reduced automaton.

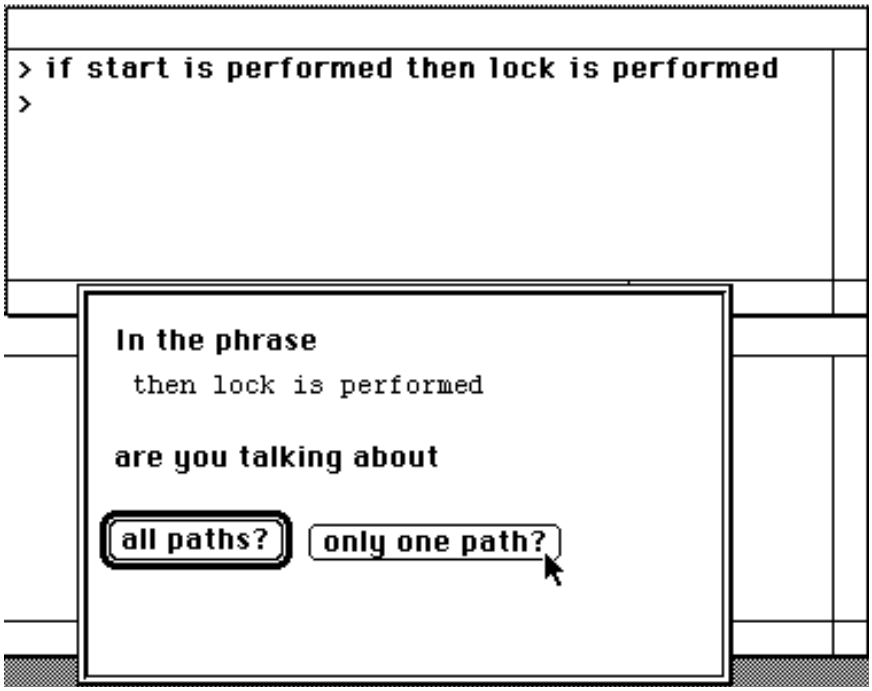

Figure 0.9: A NL2ACTL session.

controller could be able to send a signal about the termination of such a command. In ACTL language, we can express it by the following formula:

$$AG(\ EX_{!begin}true \quad \Rightarrow \quad EX_{!begin}(EF\ EX_{!ok \vee !notdone}true)\ )$$

The results of the model checking of the formula generated by NL2ACTL and of that one above are in Figure 0.12; both formulae are signaled to be true, and they are automatically recorded in a scroll-panel.
Using AMC, it is possible to create and update files of formulae and to create formulae macros to make easier the verification job. It is also possible to understand *why* a formula is true or false on a model: clicking on the button `why`, AMC will show the path that make the formula true or false, letting the user navigate in the automaton paths.

We have shown a relevant part of the JACK methodology; if we check that our specification has not a certain property, using AMC we can discover which are the paths that make such a property false and after we restart the specification cycle shown in this section, by reloading AUTOGRAPH from JACK and navigating in the model graph to reach the "bad" paths and correct them.

7. CONCLUSIONS AND FURTHER WORK

We have presented the integration project for the JACK environment for the specification and verification of concurrent and communicating systems specified by process algebra terms and modelled by finite Labelled transition Systems.
JACK is a set of integrated tools coming from the research teams at IEI–CNR and INRIA. The integration is realised with a graphical interface and communication medium between the tools; this medium is based on text files describing semantical objects (automata, networks of automata), as input/output of the tools, using the FC2 syntax. We have displayed the configuration of the different levels of the graphical interface and dedicated

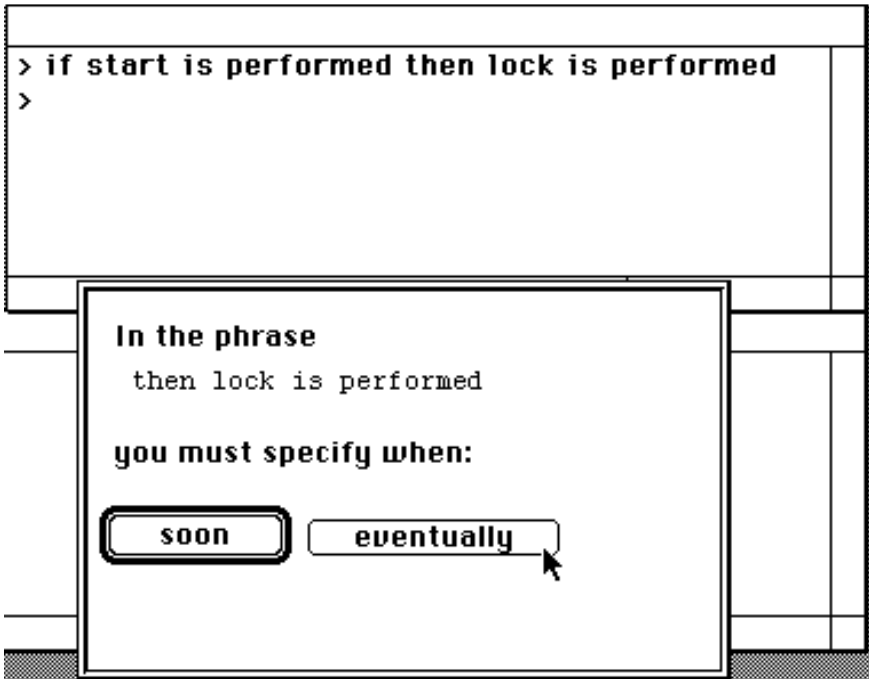

Figure 0.10: A NL2ACTL session (continued).

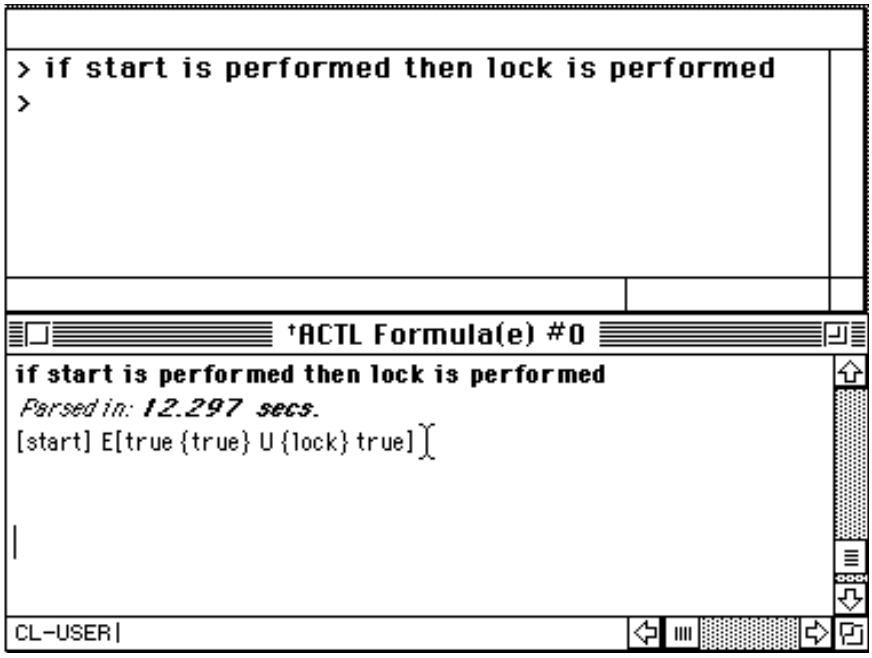

Figure 0.11: A NL2ACTL session (continued).

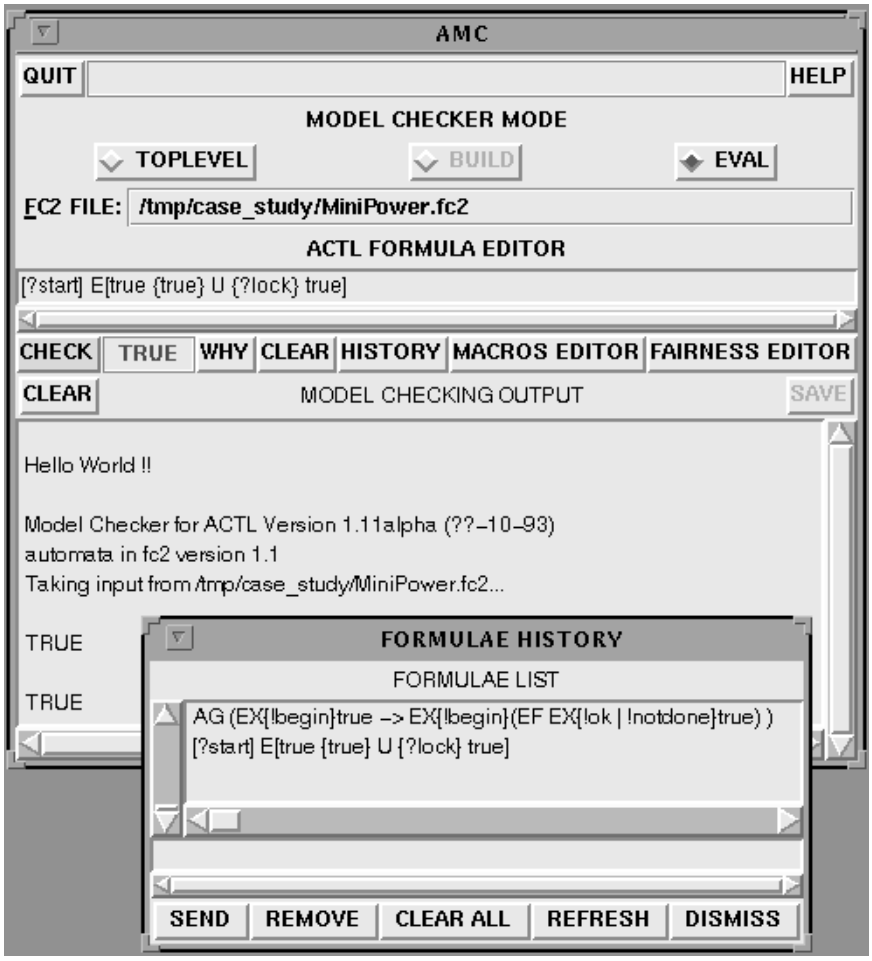

Figure 0.12: An AMC session.

a part of this work to a small case study entirely treated from specification to verification in the JACK environment.
Several directions for future works:

- Improve the graphical interface in order to enrich the set of functionalities a user can have as shortcuts, saving from tool session callings and command typings.
- Enrich tool links: tool results like AMC failure path diagnostics should be displayable within AUTOGRAPH. This can be achieved by describing the path using the FC2 syntax. More generally, increasing as much as possible tool cooperation by letting a tool interpreting results and diagnostics when possible.
- Experiment on several case studies (toy or real life examples) and improve the interface on user demand.
- Compare our environment with other verification environments.

ACKNOWLEDGEMENTS
The authors would like to thank Giovanni Ferro for his helpful comments and his work on interfacing AMC with the FC2 format, Rosario Pugliese for allowing us to use his specification as a case study, R. De Nicola, A. Fantechi, P. Inverardi, G. Ristori, R. De Simone and E. Madelaine for their contributions on the topics of this paper.

References

1. D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theorical Computer Sciences*, 1(30), 1984.
2. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207 – 226, 1983.
3. G. Berry and L. Cosserat. The synchronous programming language ESTEREL and its mathematical semantics. *LNCS*, 197, 1984.
4. T Bolognesi and M. Caneve. Squiggles: a tool for the analysis of LOTOS specifications. In *Formal Description Techiques*, pages 201–216. FORTE'88, 1989.
5. A. Bouali. Weak and branching bisimulation in fctool. Technical Report 1575, INRIA, 1991.
6. A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Fourth Workshop on Computer-Aided Verification*, Montreal, 1992.
7. G Boudol. Notes on algebraic calculi of processes. In K. Apt, editor, *Logic and Models of Concurrent Systems*. NATO ASI Series F13, 1985.
8. G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, 11(4):433–452, 1988.
9. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Automatic Verification Methods for Finite State Systems*, pages 24–37. LNCS, 1989.
10. R. De Nicola, A. Fantechi, S. Gnesi, and P. Inverardi. The JACK verification environment – internal note. Technical report, I.E.I. – C.N.R., 1993.
11. R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. In K.G. Larsen and A. Skou, editors, *Proceedings of the 3rd International Workshop on Computer Aided Verification,* Aalborg, Denmark, volume 575 of *Lecture Notes in Computer Science*, pages 37–47. Springer-Verlag, 1992.
12. R. De Nicola, P. Inverardi, and M. Nesi. Using axiomatic presentation of behavioural equivalences for manipulating CCS specifications. In automatic verification methods for finite state systems. *LNCS*, 407, 1990.
13. 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*, pages 54–67, 1991. To appear in IEEE.
14. 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,* La Roche Posay, France, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.
15. R. de Simone and D. Vergamini. Aboard auto. Rapports Techniques 111, INRIA Sophia Antipolis, 1989.
16. E. A. Emerson. Temporal and modal logic. in handbook of theoretical computer science. *J. van Leeuwen Editor – Elsevier Science*, 1990.
17. 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):243 – 263, 1994.
18. J. C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis.

14th ICSE. In *A Toolbox for the Verification of LOTOS Programs*, pages 246–261, Melbourne, 1992.

19. H. Garavel and J. Sifakis. Compilation and verification of LOTOS specifications. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification, X*, pages 379–394. North Holland, 1990.
20. J. C. Godskesen, K. G. Larsen, and M. Zeeberg. Tav user manual. Internal Report 112, Aalborg University Center, Denmark, 1989.
21. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM*, 32(1):137 – 161, 1985.
22. P. Inverardi, C. Priami, and D. Yankelevich. Verifing concurrent systems in SML. In *SIGPLAN ML Workshop*, San Francisco, June 1992.
23. P. Inverardi, C. Priami, and D. Yankelevich. Automatizing parametric reasoning on distributed concurrent systems. To appear in Formal Aspects of Computing, 1993.
24. P. Inverardi, C. Priami, and D. Yankelevich. Extended transition systems for parametric bisimulation. In *ICALP'93*, volume 700 of *LNCS*, 1993.
25. ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.
26. D. Kozen. Results on the propositional μ–calculus. *Theoretical Computer Science*, 27(2):333 – 354, 1983.
27. E. Madelaine and R. De Simone. The FC2 Reference Manual. Technical report, INRIA, 1993.
28. E. Madelaine and D. Vergamini. AUTO: A verification tool for distributed systems using reduction of finite automata networks. In S.T Vuong, editor, *Formal Description Techniques, II*, pages 61–66. North Holland, 1990.
29. E. Madelaine and D. Vergamini. Finiteness conditions and structural construction of automata for all process algebras. In R. Kurshan, editor, *Proceedings of Workshop on Computer Aided Verification*, New Brunswick, June 1990. AMS-DIMACS.
30. M. Marino. A framework for the development of natural language grammars. In *International Workshop on Parsing Technologies*, pages 350–360, Pittsburgh, 1989.
31. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
32. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
33. D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.
34. G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
35. V. Roy and R. de Simone. An autograph primer. Rapports Techniques 112, INRIA, 1989.
36. R. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.
37. R. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593,

1981.

38. P. van Eijk. Proceedings of the 4th international conference on formal description techniques (FORTE '91). In *The Lotosphere Integrated Tool Environment*, pages 473–476, New Brunswick, 1991. North Holland.

39. R.J. Van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). *Information processing '89, G.X. Ritter, ed., Elsevier Science*, pages 613–618, 1984.