Bk 15
(1994)

ISSN 0252-9742

BULLETIN

of the

European Association for Theoretical Computer Science

EATCS

Number 54

October 1994

could open new vistas for the theory as well. Ready for another swim?

References

[1] D. Angluin, Finding patterns common to a set of strings. *JCSS* 21 (1980), 46–62.

[2] J. Dassow, G. Paun and A. Salomaa, Grammars based on patterns. *International Journal of Foundation of Computer Science* 4 (1993), 1–14.

[3] J. Hartmanis, About the nature of computer science. *EATCS Bulletin* 53 (1994), 170–190.

[4] T. Jiang, A. Salomaa, K. Salomaa and Sheng Yu, Inclusion is undecidable for pattern languages. *Springer LNCS* 700 (1993), 301–312.

[5] T. Jiang, A. Salomaa, K. Salomaa and Sheng Yu, Decision problems concerning patterns. *JCSS*, to appear.

[6] L. Kari, A. Mateescu, G. Paun and A. Salomaa, Multi-pattern lnguages. *TCS*, to appear.

[7] A. Mateescu and A. Salomaa, Nondeterminism in patterns. *Springer LNCS* 775 (1994), 661–668.

[8] A. Mateescu and A. Salomaa, Pattern languages with a finite degree of ambiguity. *RAIRO* 28 (1994), 233-253.

[9] V. Mitrana, G. Paun, G. Rozenberg and A. Salomaa, Pattern systems. Submitted for publication.

[10] Y. Osada and S. Ross-Murphy, Intelligent gels. *Scientific American* May 1993, 42–47.

[11] A. Salomaa, Two-way Thue. *EATCS Bulletin* 33 (1987), 42–53.

TECHNICAL CONTRIBUTIONS

JACK: Just Another Concurrency Kit. The integration project.

Amar Bouali[*]
CWI
Kruislaan 413
NL-1098 SJ Amsterdam (NL)
amar@cwi.nl

Stefania Gnesi[†] Salvatore Larosa[†]
IEI-CNR
via Santa Maria, 46
I-56100 Pisa (ITALY)
(gnesi,larosa)@iei.pi.cnr.it

August 5, 1994

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 a 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 of describing 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.

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 (Bolognesi and Caneve, 1989; Cleaveland et al., 1989; Madelaine and Vergamini, 1990a; van Eijk, 1991; Godskesen et al., 1989; Fernandez et al., 1992).
Recently a new verification environment, JACK, (De Nicola et al., 1993) 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.

[*] A part of the work described in the paper was carried out while the author was visiting the *Istituto di Elaborazione dell'Informazione* of CNR in Pisa as part of the fellowship program 1992/93 of the European Research Consortium for Informatics and Mathematics.

[†] The work described was partially performed in the context of the LAMBRUSCO project, supported by the CNR Finalized Project for Information Systems and Parallel Computation, unità operativa IEI - University La Sapienza, Rome.

JACK covers much of the formal software development process, including the formalization of requirements (Fantechi et al., 1994), rewriting techniques (De Nicola et al., 1990), behavioural equivalence proofs (Inverardi et al., 1992; De Nicola and Vaandrager, 1990), graph transformations (Roy and de Simone, 1990), logic verifications (De Nicola et al., 1992). The logical properties are specified by an action based temporal logic, ACTL defined in (De Nicola and Vaandrager, 1990), 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. Finally, in Section 5 the JACK interface is described (with an overview of the components 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*
$\mathcal{A} = (Q, q_0, Act \cup \{r\}, R)$, *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 r is the unobservable action. We let $a, b, \ldots$ range over Act, and $\alpha, \beta, \ldots$ range over $Act \cup \{r\}$;*

- *$R \subseteq Q \times Act \cup \{r\} \times Q$ is the transition relation.* $\Box$

Note 2.2 *For $A \subseteq Act$, we let A_r denote the set $A \cup \{r\}$.*
For $A \subseteq Act_r$, 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 $R_{Act_r}(s)$.
For $A, B \subseteq Act_r$, we let A/B denote the set $A - (A \cap B)$. Often, we simply write $q \xrightarrow{\alpha} q'$ for $(q, \alpha, q') \in R$ $\Box$

Definition 2.3 *Given an LTS $\mathcal{A} = (Q, q_0, Act \cup \{r\}, R)$, 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.* $\Box$

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, (Austry and Boudol, 1984; Boudol, 1985), 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 (Milner, 1989). 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 (de Simone and Vergamini, 1989).

2.2.1 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 "?". r denotes the special action not belonging to Act, symbolising the unobservable action (to model internal process communications); we let $Act_r = Act \cup \{r\}$ 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_r; X is a label denoting a process variable;

$$\begin{array}{lll} R & ::= & \mathtt{stop} \mid a : R \mid R + R \mid \\ & & \mathtt{let\ rec}\ \{X = R\ [\mathtt{and}\ X = R]\ \}\ \mathtt{in}\ X \end{array}$$

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:

$$\begin{array}{lll} P & ::= & R \mid P \parallel P \mid P \backslash a \mid P[a/b] \mid a * P \mid \\ & & \mathtt{let}\ \{X = P\ [\mathtt{and}\ X = R]\ \}\ \mathtt{in}\ X \end{array}$$

- $\parallel$ is the parallel operator;
- $P \backslash a$ 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 1 shows the structural operational semantics of some CCS/MEIJE operators previously described, in terms of labelled transition systems (Plotkin, 1981): 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		
$a : P$	$\dfrac{}{a : P \xrightarrow{a} P}$		
$P + Q$	$\dfrac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$	$\dfrac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$	
$P \parallel Q$	$\dfrac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}$	$\dfrac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}$	$\dfrac{P \xrightarrow{a?} P', Q \xrightarrow{a!} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$

Figure 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, (Park, 1981).

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:*

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

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

Observational bisimulations, first introduced by Milner (Milner, 1989), 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)

$$\overset{\epsilon}{\Longrightarrow} \ \overset{def}{=} \ (\xrightarrow{\tau})^*$$
$$\forall a \in Act, \overset{a}{\Longrightarrow} \ \overset{def}{=} \ \overset{\epsilon}{\Longrightarrow}\xrightarrow{a}\overset{\epsilon}{\Longrightarrow}$$

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 (Van Glabbeek and Weijland, 1984) 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:*

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

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 (De Nicola and Vaandrager, 1990); 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:

$$\begin{array}{lll} a \models tt & & \textit{always;} \\ a \models b & \textit{iff} & a = b; \\ a \models \neg\chi & \textit{iff} & \textit{not } a \models \chi; \\ a \models \chi \vee \chi' & \textit{iff} & a \models \chi \textit{ or } a \models \chi'. \end{array}$$

□

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:*

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

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:*

$$\begin{array}{lll} s \models tt & & \textit{always;} \\ s \models \phi \wedge \phi' & \textit{iff} & s \models \phi \textit{ and } s \models \phi'; \\ s \models \neg\phi & \textit{iff} & \textit{not } s \models \phi; \\ s \models E\pi & \textit{iff} & \textit{there exists a path } \sigma \in \Pi(s) \textit{ such that } \sigma \models \pi; \\ s \models A\pi & \textit{iff} & \textit{for all maximal paths } \sigma \in \Pi(s),\ \sigma \models \pi; \\ \sigma \models X_\chi\phi & \textit{iff} & |\sigma| \geq 1 \textit{ and } \sigma(2) \in R_{\kappa(\chi)}(\sigma(1)) \textit{ and } \sigma(2) \models \phi; \\ \sigma \models X_\tau\phi & \textit{iff} & |\sigma| \geq 1 \textit{ and } \sigma(2) \in R_{\{\tau\}}(\sigma(1)) \textit{ and } \sigma(2) \models \phi; \\ \sigma \models \phi\,{}_\chi U \phi' & \textit{iff} & \textit{there exists } i \geq 1 \textit{ such that } \sigma(i) \models \phi', \\ & & \textit{and for all } 1 \leq j \leq i-1\textit{:} \\ & & \sigma(j) \models \phi \textit{ and } \sigma(j+1) \in R_{\kappa(\chi),}(\sigma(j)); \\ \sigma \models \phi\,{}_\chi U_{\chi'}\phi' & \textit{iff} & \textit{there exists } i \geq 2 \textit{ such that } \sigma(i) \models \phi',\ \sigma(i-1) \models \phi, \\ & & \sigma(i) \in R_{\kappa(\chi')}(\sigma(i-1)) \textit{ and for all } 1 \leq j \leq i-2\textit{:} \\ & & \sigma(j+1) \in R_{\kappa(\chi),}(\sigma(j)). \end{array}$$

□

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

- $E\tilde{X}_\chi\phi$ for $\neg AX_\chi\neg\phi$, and $A\tilde{X}_\chi\phi$ for $\neg EX_\chi\neg\phi$; these are called the *weak next* operators.
- $EF\phi$ for $E(tt\,{}_\tau U\,\phi)$, and $AF\phi$ for $A(tt\,{}_\tau U\,\phi)$; 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 $\chi \not\to ff$;
- $<> \phi$, 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) (Emerson, 1990; Hennessy and Milner, 1985; Ben-Ari et al., 1983; Kozen, 1983). 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 (De Nicola et al., 1992). Following this first attempt, we have developed an environment based on the links proposed in (De Nicola et al., 1992), and on new links that exploit the Fc2 format (Madelaine and de Simone, 1993) (see Figure 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

4.1.1 AUTOGRAPH (INRIA)

This is a graphic specification tool for the design of parallel and communicating processes (Roy and de Simone, 1989) that provides functionalities for a compositional development of

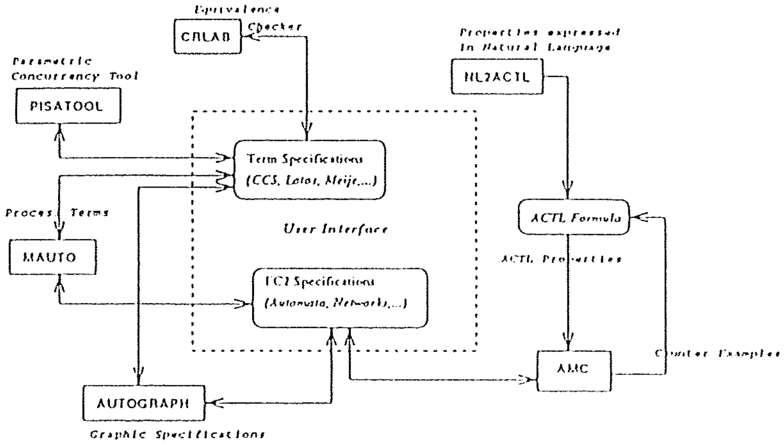


Figure 2: The integration project

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 several output formats in which a graphical specification can be translated, including Fc2, the MAUTO syntax for terms and Postscript.

4.1.2 AUTO/MAUTO (INRIA)

MAUTO (de Simone and Vergamini, 1989) (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 (Austry and Boudol, 1984), CCS (Milner, 1989), LOTOS (ISO, 1987) and ESTEREL (Berry and Cosserat, 1984) 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 (Madelaine and Vergamini, 1990b). 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.

4.1.3 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 (Marino, 1989).

4.2 Verification Tools

4.2.1 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.

[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 expresive power of rational expressions.

4.2.2 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.

4.2.3 CRLAB (IEI-CNR)

CRLAB is a system based on rewriting strategies (De Nicola et al., 1990; De Nicola et al., 1991). The input, which is supplied to the system interactively, can be LOTOS (ISO, 1987) or CCS (Milner, 1989) 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 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.

4.2.4 PISATOOL (IEI-CNR)

The PISATOOL (Inverardi et al., 1992; Inverardi et al., 1993a) 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* (Inverardi et al., 1993b), i.e., transition systems labelled on nodes by regular expressions (Boudol and Castellani, 1988; Tarjan, 1981b; Tarjan, 1981a) 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 (Paige and Tarjan, 1987) 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.

4.2.5 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 (Bouali, 1991; Bouali and de Simone, 1992). 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:

4.3.1 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.

4.3.2 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;

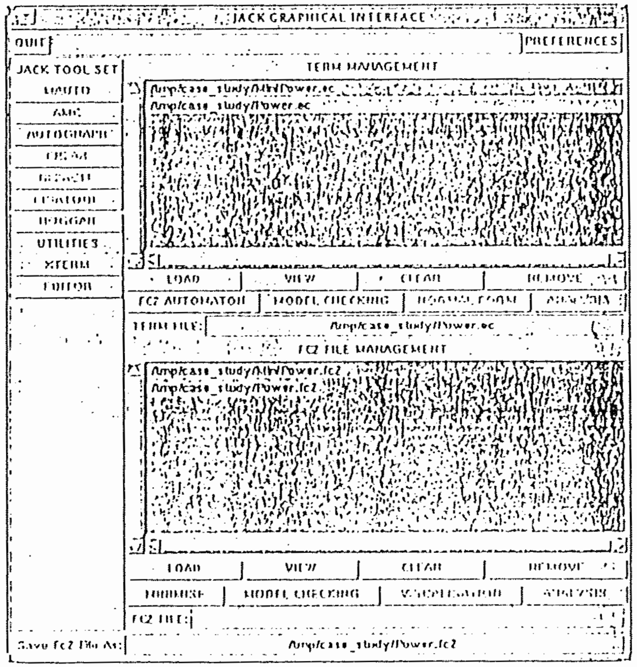
The format is more concrete than process algebra. Moreover, it enables the description of different views of parallelism and synchronisation. In fact, all finite state systems that can be represented by process algebrae are also representable by the Fc2 format.

5 The JACK Interface

In order to test our integration ideas, we have developed an user - friendly interface (Figure 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.



TERM FILE: /tmp/case_study//Tower.ec

Save Fc2 File As: /tmp/case_study//Tower.fc2

Figure 3: The JACK general control panel.

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 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.

5.2 Fc2 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 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.
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 (De Nicola et al., 1992), 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 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.
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.

[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

References

Austry, D. and Boudol, G. Algèbre de processus et synchronisation, *Theorical Computer Sciences*, 1(30), (1984).

Ben-Ari, M., Pnueli, A., and Manna, Z. The temporal logic of branching time, *Acta Informatica*, 20, 207 – 226, (1983).

Berry, G. and Cosserat, L. The synchronous programming language ESTEREL and its mathematical semantics, *Lecture Notes in Computer Science*, 197, (1984).

Bolognesi, T. and Caneve, M. Squiggles: a tool for the analysis of LOTOS specifications, in *Formal Description Techiques*, FORTE'88, 201–216, 1989.

Bouali, A. Weak and Branching Bisimulation in Fctool, Technical Report 1575, INRIA, 1991.

Bouali, A. and de Simone, R. Symbolic bisimulation minimisation, in *Fourth Workshop on Computer-Aided Verification*, Montreal, 1992.

Boudol, G. Notes on algebraic calculi of processes, in *Logic and Models of Concurrent Systems*, NATO ASI Series F13, 1985.

Boudol, G. and Castellani, I. A non-interleaving semantics for CCS based on proved transitions, *Fundamenta Informaticae*, 11(4), 433–452, (1988).

Cleaveland, R., Parrow, J., and Steffen, B. The concurrency workbench, in *Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science*, 1989, 24–37.

De Nicola, R., Fantechi, A., Gnesi, S., and Inverardi, P. The JACK verification environment, Internal Report, I.E.I. – C.N.R., 1993.

De Nicola, R., Fantechi, A., Gnesi, S., and Ristori, G. An action based framework for verifying logical and behavioural properties of concurrent systems, in *Computer Networks and ISDN systems*, 25(7), 761–778, (1993).

De Nicola, R., Inverardi, P., and Nesi, M. Using axiomatic presentation of behavioural equivalences for manipulating CCS specifications, in *Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science*, No. 407 (J. Sifakis, ed.), 1990.

De Nicola, R., Inverardi, P., and Nesi, M. Equational reasoning about LOTOS specifications: A rewriting approach, in *Sixth International Workshop on Software Specification and Design*, 54–67, (1991). To appear in IEEE.

De Nicola, R. and Vaandrager, F. Action versus state based logics for transition systems, in *Semantics of Systems of Concurrent Processes, Lecture Notes in Computer Science*, No. 469 (I. Guessarian, ed.), Springer-Verlag La Roche Posay, France, 1990, 407–419.

de Simone, R. and Vergamini, D. Aboard AUTO, Rapports Techniques 111, INRIA Sophia Antipolis, 1989.

Emerson, E. A. Temporal and modal logic, in *Handbook of Theoretical Computer Science, (J. van Leeuwen, ed.), Elsevier Science*, 1990.

Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., and Moreschini, P. Assisting requirement formalization by means of natural language translation, *Formal Methods in Systems Design*, 4(2), 243 – 263, (1994).

Fernandez, J. C., Garavel, H., Mounier, L., Rasse, A., Rodriguez, C., and Sifakis, J. A Toolbox for the Verification of LOTOS Programs, 14th ICSE, Melbourne, 1992, 246–261.

Godskesen, J. C., Larsen, K. G., and Zeeberg, M. TAV User Manual, Internal Report 112, Aalborg University Center, Denmark, 1989.

Hennessy, M. and Milner, R. Algebraic laws for nondeterminism and concurrency, *Journal of ACM*, 32(1), 137 – 161, (1985).

Inverardi, P., Priami, C., and Yankelevich, D. Verifing concurrent systems in SML, in *SIGPLAN ML Workshop*, San Francisco, 1992.

Inverardi, P., Priami, C., and Yankelevich, D. Automatizing parametric reasoning on distributed concurrent systems, (1993). To appear in Formal Aspects of Computing.

Inverardi, P., Priami, C., and Yankelevich, D. Extended transition systems for parametric bisimulation, in *ICALP'93, Lecture Notes in Computer Science* No. 700, 1993.

ISO. Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, Technical report ISO/TC97/SC21/N DIS8807, 1987.

Kozen, D. Results on the propositional μ-calculus, *Theoretical Computer Science*, 27(2), 333 – 354, (1983).

Madelaine, E. and de Simone, R. The FC2 Reference Manual, Technical report, INRIA, 1993.

Madelaine, E. and Vergamini, D. AUTO: A verification tool for distributed systems using reduction of finite automata networks, in *Formal Description Techniques, II* (S.T Vuong, ed.), North Holland, 1990a, 61–66.

Madelaine, E. and Vergamini, D. Finiteness conditions and structural construction of automata for all process algebras, in *Proceedings of Workshop on Computer Aided Verification* (R. Kurshan, ed.), AMS-DIMACS, New Brunswick, 1990b.

Marino, M. A framework for the development of natural language grammars, in *International Workshop on Parsing Technologies*, Pittsburgh, 1989, 350–360.

Milner, R. *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.

Paige, R. and Tarjan, R. Three partition refinement algorithms, *SIAM Journal on Computing*, 16(6), 973–989, (1987).

Park, D. Concurrency and automata on infinite sequences, in 5th *GI Conference, Lecture Notes in Computer Science*, No. 104 (P. Deussen, ed.), Springer-Verlag, 1981, 167–183.

Plotkin, G. A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

Roy, V. and de Simone, R. An Autograph Primer, Rapports Techniques 112, INRIA, 1989.

Roy, V. and de Simone, R. AUTO and AUTOGRAPH, in *Proceedings of workshop on computer aided verification*, *AMS-DIMACS*, June 1990.

Tarjan, R. Fast algorithms for solving path problems, *Journal of the ACM*, 28(3), 594-614, (1981a).

Tarjan, R. A unified approach to path problems, *Journal of the ACM*, 28(3), 577-593, (1981b).

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

Van Glabbeek, R. and Weijland, W. Branching time and abstraction in bisimulation semantics (extended abstract), *Information processing '89* (G.X. Ritter, ed.), 613-618, (1984).

INFINITE (ALMOST PERIODIC) WORDS, FORMAL LANGUAGES AND DYNAMICAL SYSTEMS

Solomon MARCUS and Gheorghe PĂUN

Institute of Mathematics of the Romanian Academy of Sciences
PO Box 1 - 764, 70700 București, ROMANIA

Abstract. Motivated by recent topics in the study of dynamical systems (as well as by topics in mathematical analysis), we investigate various modes of associating a language of finite words to an infinite word.

Considering the finite subwords of a given infinite word and the infinite words that can be associated to a formal language seems to be a very natural topic. Besides the intrinsic interest, this topic is stimulated by some recent developments in the study of dynamical systems, mainly of the dynamical system (X, T) called *shift*, where X is the set of states, T is a mapping from X into X, the states in X being infinite words on the alphabet $\{0, 1\}$, while the mapping T associates to each infinite word $\omega = a_1a_2a_3\ldots, a_i \in \{0,1\}, i \geq 1$, the infinite word $T(\omega) = a_2a_3a_4\ldots$ (i.e., a_1 is shifted into a_2, a_2 is shifted into a_3, etc.). A particular importance have the infinite words that are almost periodic: ω is almost periodic if for any finite subword x of it we can portion ω into subwords $x_1, x_2, x_3, \ldots$ (i.e., $\omega = x_1x_2x_3\ldots$) of equal length such that every one of the subwords x_i includes a copy of the subword x.

Let us remind that the concept of almost periodicity was introduced, in the field of mathematical analysis, by Harald Bohr, in a series of papers published in *Acta Mathematica* (vol. 45 - 47, 1925 - 1927), under the following form: a continuous function $f : \mathrm{R} \longrightarrow \mathrm{R}$ (where R is the set of real numbers) is almost periodic if for any $\varepsilon > 0$ there exists a positive number $\delta(\varepsilon)$ such that any real compact interval of length $\delta(\varepsilon)$ contains a number $\tau(\varepsilon)$ with the property $|f(x + \tau(\varepsilon)) - f(x)| < \varepsilon$ for any $x \in \mathrm{R}$. A systematic presentation of almost periodic functions is due to J. Favard [1]. The concept of almost periodicity proved its importance in celestial mechanics. An important contribution in this field is due to O. Toeplitz [14]. For further aspects, more related to our concern here, see K. Jacobs, [6] (p. 106 and p. 213 - 216). Particularly, let us call attention on the fact that the famous infinite word Thue-Morse ([13, [9]) $\omega_{TM} = 01101001\ldots$ (we start from 0 and we iteratively apply the morphism $h(0) = 01, h(1) = 10$) and the Mephisto-Waltz word [6] 001001110... (in the first step we write 0; in the second step we write 01; in the third step we repeat what we

wrote at the preceding two steps and then we repeat again the same expression by replacing 0 with 1 and 1 with 0; at the n-th step we repeat what we wrote at the preceding $n-1$ steps, then we write the same expression by replacing 0 with 1 and 1 with 0) are almost periodic. For the sake of completeness and because we shall use this fact below, we prove this assertion for ω_{TM} (in a simple way, different from that in [6]).

Lemma 1. *The Thue-Morse infinite word is almost periodic.*

Proof. Let x be a finite subword of ω_{TM}. Let n be the smallest integer with the property that x is a subword of $h^n(0)$. (It exists, because $h^i(0)$ is a prefix of $h^{i+1}(0)$, hence of ω_{TM}, for all $i \geq 1$.) Write ω_{TM} as a sequence of words of length two, $\omega_{TM} = x_1x_2x_3\ldots$. We have $x_i \in \{01, 10\}$ for all $i \geq 1$ (hence each x_i contains one occurrence of 0). Because $\omega_{TM} = h^n(\omega_{TM}) = h^n(x_1)h^n(x_2)h^n(x_3)\ldots$, the partition of ω_{TM} into the words $h^n(x_1), h^n(x_2), \ldots$ proves the almost periodicity of the Thue-Morse word (note that the length of each $h^n(x_i)$ is 2^{n+1}). ◊

Basically, starting from an infinite word $\omega = a_1a_2a_3\ldots$ over some given alphabet A ($a_i \in A$ for each $i = 1, 2, \ldots$) we can construct a language (of finite words) by "cutting" parts of ω *in a systematic way*. The most natural such ways are (1) to take the prefixes of ω, (2) to take all (finite) subwords of ω and (3) to consider a partition of ω. The first two cases lead to a unique language associated with the infinite word, in the third one we can obtain infinitely many languages, corresponding to the infinitely many possible partitions of an infinite word (more precisely, we can obtain a non-denumerable set of languages, including finite languages and languages which are not recursively enumerable); moreover, every language can lead to an infinite word (by concatenating its words) in infinitely many ways.

An interesting variant of (2) is to consider those subwords of ω with a given property, for instance, to appear infinitely many times in ω.

It is worth mentioning here the natural analogy between the sum of a series, in mathematical analysis, and the concatenation of the words of a language. In both cases we are faced with the property of commutativity. Numerical series are commutative when the terms are of the same sign, but semiconvergent series may have any real number ($+\infty$ and $-\infty$ too) as their sum (following a theorem by Riemann). We may ask under what conditions on L all infinite words obtained by concatenation of all words in L have a similar structure. "Similar structure" could mean, for example, periodicity or almost periodicity, or the property that each finite subword occurs infinitely many times, etc. At the limit, we may ask under what conditions different ordering in concatenation may keep invariant the infinite word obtained. We may also ask what could be here the corresponding theorem of that due to Riemann for semiconvergent numerical series. We do not consider here such questions, but we investigate the first two ways mentioned above of associating a language to an infinite word.

First, some notions and notation. As usual, A^* is the free monoid generated by A, $|x|$ is the length of x, λ is the empty word and $A^+ = A^* - \{\lambda\}$. A language $L \subseteq A^*$ is called *thin* ([10]) if for each $n \geq 0$ we have $\mathrm{card}\{x \in L \mid |x| = n\} \leq 1$ (L contains at most one word of each length). The set of prefixes (subwords, respectively, suffixes) of a word x is denoted by $Pref(x)$ ($Sub(x)$, $Suf(x)$, respectively); the same notations $Pref$ and Sub are used for finite prefixes and finite subwords of infinite words and are extended in the obvious way to languages.

Given an alphabet A, we denote by A^∞ the set of all infinite words over A. Then, for $\omega \in A^\infty$, $\omega = a_1a_2a_3\ldots$, $a_i \in A$, $i = 1, 2, \ldots$, we denote

$$s_i(\omega) = a_{i+1}a_{i+2}\ldots, \quad i \geq 0$$

(the *shift* of lenght i in ω).

We denote by FIL, FSL the families of languages of the forms $Pref(\omega)$, $Sub(\omega)$, respectively, for ω an infinite word.