Reference : Lo/WP1/T1.2/IEI/N0026/V1

ESPRIT Project 2304[1]

Title : **Verifying Concurrent Systems by Talking to them**

Status : Public

Type : Paper, Preliminary

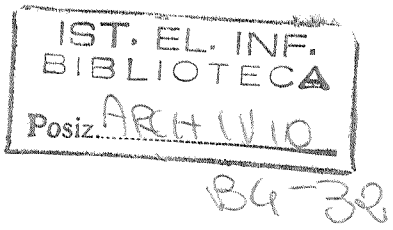
Editor : A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Marino, P. Moreschini

Date : 3/3/1992

Distribution : PM, WP1M, T1.2

Owner :

Note :



[1] The LOTOSPHERE consortium consists of the following companies/institutes/universities:

Alcatel Standard Electrica, Ascom Tech, British Telecommunications, C.N.R.-CNUCE, PTT-DNL, Consorzio Pisa Ricerche, Gesellschaft für Mathematik und Datenverarbeitung, C.N.R.- Istituto di Elaborazione della Informazione, Institut National de Recherche en Informatique et en Automatique, Laboratoire d'Automatique et d'Analyse des Systemes du CNRS, Océ Nederland, SYSECA Logiciel, TECSIEL, Technische Universität Berlin, Universidad Politécnica de Madrid, University of Stirling, University of Twente.

Verifying Concurrent Systems by Talking to them

A. Fantechi♪, S. Gnesi♪, G. Ristori♆
M. Carenini∾, M. Marino∾, P. Moreschini∾

♪ *I.E.I. - C.N.R. Pisa, Italy*
∾ *AITech s.n.c. Pisa, Italy*
♆ *Dip. di Informatica, Univ. di Pisa, Italy*

Abstract

In this paper we present a prototype translator from Natural Language expressions into Temporal Logic formulae. The translator is realized using a general development environment for Natural Language processing and it has been interfaced with a verification environment of behavioural properties on concurrent systems specified by process algebras. This tool can be viewed as an aid to override the many imprecisions that frequently occur in the passage from informal requirements expressed by means of Natural Language expressions to Temporal Logic formulae.

1 Introduction

A critical phase in the classical software life cycle is the capturing of informally stated requirements into a formalized design or specification of a system.
This step is particularly critical in the case formal verification is used, which requires rigorous formal description of the specification, against which the later products of the development process will be formally verified to prove their correctness. A failure in embedding the informally stated requirement into the formal specification will invalidate the later verification efforts.
The problem is therefore to concentrate on the ability to extract the intended meaning of informally stated requirements and to express it formally.
Temporal Logic, which has been recognized suitable for the abstract specification of properties and requirements on concurrent systems, happens to be a good candidate as a formalism to aid this step. In our experience on the specification and verification of properties by means of Temporal Logic, we have verified that many imprecisions frequently occur in the passage from informal requirements expressed by means of Natural Language expressions to Temporal Logic formulae, due to the inherent ambiguities of Natural Language, and sometimes also to the difficulties in understanding the deep meaning of the logic operators. We have hence looked for an answer to these problems in the current state of the art in Natural Language Understanding.
Development environments are available which support the construction of Natural Language Processing applications. Among these, PGDE is a general development environment for Natural Language Processing [12], aimed at the construction, debugging and testing of Natural Language

grammars and dictionaries. Using PGDE, it has been possible to realize a prototype translator, NL2ACTL, from Natural Language expressions into Temporal Logic formulae.
The context in which we want to verify the preservation of formal requirements between a specification and its implementation is that of concurrent systems defined by process algebras, like CCS [14], and interpreted on Labelled Transition Systems. In the last years some logics interpreted on Labelled Transition Systems have been proposed [1, 4, 6, 8, 10, 16]; among these we choose ACTL [5], a logic simple enough to deal with, but at the same time powerful enough to cover interesting properties of concurrent systems.
Therefore NL2ACTL has been tailored to ACTL, on a limited domain of application, in order to be able to experience the problems which arise in this field and to elaborate a strategy for developing a more general tool with this functionality.
The translator has been integrated in a verification environment based on the ACTL logic; this environment includes a model checker which permits to verify ACTL formulae on process algebra terms [3]. The environment so obtained is able to cover a larger extent of the formal software development process, giving the opportunity to check the validity of informal requirements on process algebra specifications.
The paper is organized as follows: we introduce the problem of informal requirements capturing by an example specification (Section 2); the requirements are given an ACTL description by hand (Section 3), showing some of the problems which arise in this translation. The translator is then described (Section 4), both by outlining its internal organization and by showing its use.

2 A specification example

In the system analysis phase of the classical software development cycle, requirements are defined; although this process can be *systematic* in some sense, their capture is usually intuitive and informal. This process aims to obtain a general description of functional requirements (what the program should do) and non-functional requirements (as resources, performance etc.). Since requirements should be satisfied by the system, functional requirements must be checked with respect to the program; to do this a number of testing techniques have been developed.
In a formal software development environment in which a system is specified by means of a process algebra term, several notions of equivalence and preorder are available to compare a specification with respect to a more refined one, in order to show that the refined specification is a correct implementation of the more abstract one [17]. However, equivalences provide methods to check the initial set of functional requirements on a given specification only if the key behaviours corresponding to informal requirements are specified using the same formalism; a failure in embedding the informally stated requirements into the a formal specification will invalidate the later verification efforts. In this respect, a logic approach has been developed; in particular, Temporal Logics have been recognized as a useful means to express behavioural properties of concurrent systems, and model checkers have been realized to check properties, written as Temporal Logic formulae, on the semantic model of a process algebra term, i. e. a Labelled Transition System (shortly, LTS). Model checking requires however a translation of the set of the informal functional

requirements established in the system analysis phase into a Temporal Logic formalism; our work is situated in this sphere and it aims to provide an aid in this translation, in the context of the checking problem in a formal software development environment, as represented by the *waterfall model* in Fig. 1.

Also maintenance and enhancement of requirements can be achieved, due to the benefits obtained from the verification phase about the requirement refinement cycle; starting from the concrete model (say, a formal specification) of the system and from the formalization of abstract requirements (a list of Temporal Logic formulae), the verification of the latter on the former - by means of a model checker - may provide useful information to adjust both the specification and the expression (and then formalization) of the requirements.

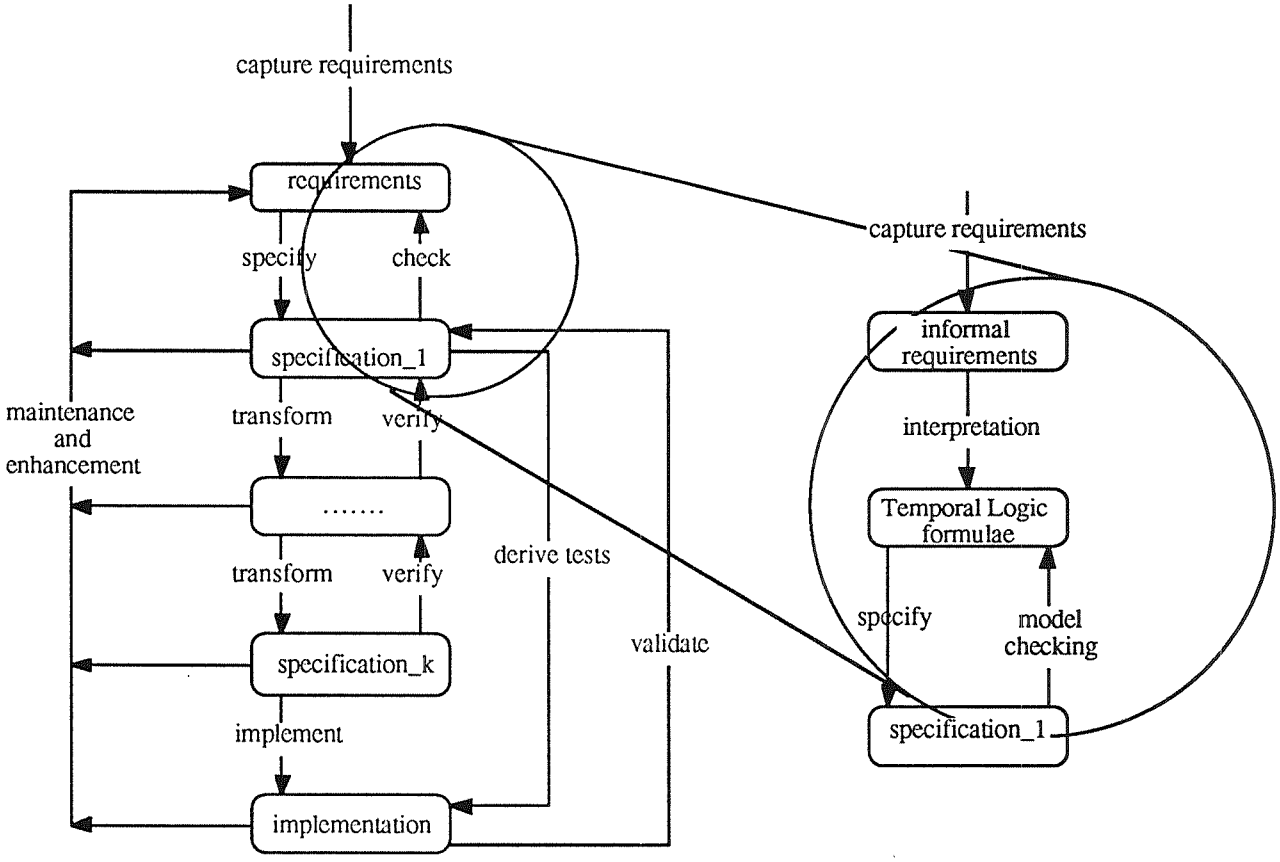


Fig 1: The waterfall model of software development

The prototype we present is a first step in this direction; to simplify the development of the tool we have chosen to take into account a particular model checker [3] (whose input models are LTSs and the underlying Temporal Logic is ACTL) and a precise application domain, in order to have a small set of verbs to manipulate. The application domain that has been chosen is a *vending machine*. As an example, consider a machine able to give a coffee or a tea if someone inserts a coin and a soup if two coins are inserted (this example has already been used in [9]).

Some informal requirements (as given by an anonymous purchaser) that the machine must satisfy may be:

- after a coin has been inserted it is possible only to have tea or coffee or recollect the coin
- after a coin has been inserted it is not possible to have a soup until another coin is inserted
- if a coin is inserted then after another coin is inserted it is possible to have soup
- it is always possible to insert a coin
- it is always possible to recollect a coin
- if it is possible to have a tea then it is possible to have a coffee
- after a coin has been inserted eventually it will be possible to have a tea

The vending machine specification in CCS (see the Appendix for the definition of its syntax and semantics) is the following:

rec Vending_Machine =
coin. (tea. Vending_Machine + recollect. Vending_Machine + τ. coffee. Vending_Machine) +
coin. (coffee. Vending_Machine + recollect. Vending_Machine + τ. tea. Vending_Machine) +
coin. (recollect. Vending_Machine +
coin. (recollect. Vending_Machine + soup. Vending_Machine) +
τ. (tea. Vending_Machine + recollect. Vending_Machine + τ. coffee. Vending_Machine) +
τ. (coffee. Vending_Machine + recollect. Vending_Machine + τ. tea. Vending_Machine))

CCS terms are usually interpreted over Labelled Transition Systems (for the definition, see the Appendix); the LTS corresponding to the CCS specification of the Vending_Machine is shown in Fig. 2.

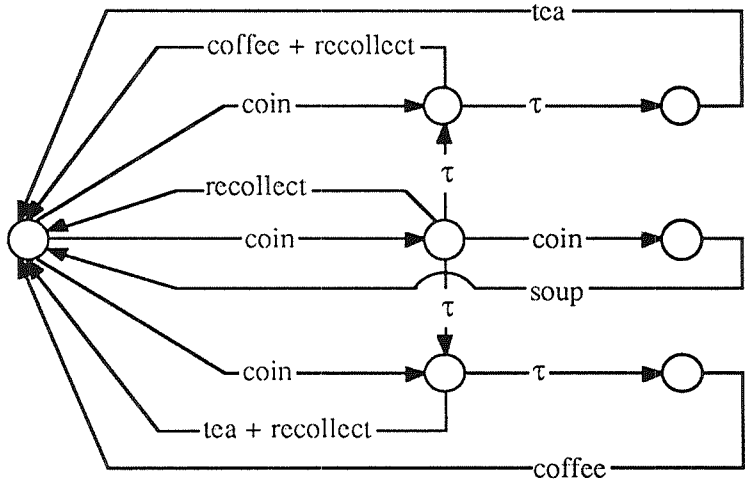


Fig. 2: The LTS of the Vending_Machine.

3 The Branching Time Temporal Logic ACTL

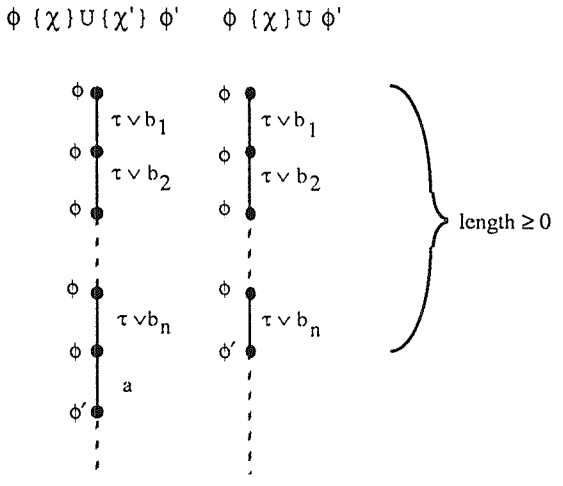
The action based logic ACTL [5] is a branching time Temporal Logic whose models are Labelled Transition Systems. The choice of ACTL to express formally the requirements of concurrent systems, described by means of process algebras was due to the following reasons:

- ACTL is an action based logic whose models are Labelled Transition Systems, therefore it is more suitable to speak about properties of systems interpreted on Labelled Transition Systems than logics as CTL , whose models are Kripke structures [2].
- In the space of action logics, that ranges from HML [6] to the μ-calculus [8], ACTL seems to be the most interesting. In fact it is more expressive than HML, and it is sufficiently powerful to express interesting properties of concurrent systems without introducing the overhead of formulae with fixed points, hard to understand, as in the μ-calculus.
- A verification environment is available, which uses ACTL for the verification of logical properties of concurrent systems described by means of process algebras.

The syntax of ACTL, a subset of ACTL* [5], is defined by the state formulae generated by the following grammar, where φ, φ', ... range over state-formulae, γ over path formulae and χ and χ' are action formulae, that is, formulae which describe the occurrence of actions (for a detailed presentation of action formulae see the Appendix):

$$\varphi ::= \mathtt{true} \mid \sim\varphi \mid \varphi \,\&\, \varphi' \mid \mathtt{E}\gamma \mid \mathtt{A}\gamma$$
$$\gamma ::= \mathtt{X}\{\chi\}\varphi \mid \mathtt{T}\ \varphi \mid [\varphi\ \{\chi\}\mathtt{U}\{\chi'\}\ \varphi'] \mid [\varphi\ \ \{\chi\}\mathtt{U}\ \varphi']$$

The meaning of the boolean connectives and of the existential and universal quantifier are the usual ones. The indexed next modalities $\mathtt{X}\{\chi\}\varphi$, $\mathtt{T}\varphi$ say that in the next state of the path, reached respectively with an action in χ or with a τ, the formula φ holds; the indexed until modalities introduced above are better clarified in Fig. 3:



where $b_1 \models \chi$, ..., $b_n \models \chi$ and $a \models \chi'$

Fig. 3: The meaning of the indexed until modalities.

The satisfaction relation for ACTL formulae is detailed in the Appendix.

As usual, a set of logical operators and modalities derived from the basic ones is introduced: `false`, φ | φ' (or), φ -> φ' (implies), Fφ (eventually), Gφ (always). Other derived modalities [15] similar to those of Hennessy-Milner logic with until in [4] are:

- φ <a> φ' for E[φ {false}U{a} φ'],
- φ <>φ' for E[φ {false}U φ'],
- <k> φ for true <k> φ,
- [k] φ for ~ <k> ~φ.

We can now express in ACTL the informal requirements that the Vending Machine should satisfy:

- after a coin has been inserted it is possible only to have tea or coffee or recollect the coin
  `[coin] A [true {false} U {tea | coffee | recollect} true]`
- after a coin has been inserted it is not possible to have a soup until another coin is inserted
  `[coin] A [true {~soup} U {coin} true]`
- if a coin is inserted then after another coin is inserted it is possible to have soup
  `[coin] [coin] <soup> true`
- it is always possible to insert a coin
  `AG<coin> true`
- it is always possible to recollect a coin
  `AG<recollect> true`
- if it is possible to have a tea then it is possible to have a coffee
  `<tea>true -> <coffee>true`
- after a coin has been inserted eventually it will be possible to have a tea
  `[coin]AFEX{tea}true`

This list suggests the following key observations:

a) sentences with similar structure lead to similar formulae, and, more in general, there are structures which evidently correspond to particular Temporal Logic operators;

b) action designators have been extracted from specific segments of the sentences;

c) the translation has been guided from the knowledge of the semantics of the example; this is apparent in the different translation of the "if <A> then <B>" structure: it is sometimes understood as a succession of events ("if <A> then, at the next step, <B>"), and sometimes is understood as an implication ("<A> implies <B>"). This is just an example of the ambiguities that can be found in Natural Language expressions. In this context, ambiguities of this kind may be due to inaccuracies in the writing of requirements, or to the relying on information derivable only from the context, like in the case above;

d) all the formulae above give a "weak" interpretation of the requirements, in the sense that they allow the vending machine to perform internal actions between, say, the insertion of a coin and the delivery of a coffee. A "strong" interpretation of the requirements could be possible, in which case corresponding strong formulae should be generated (e.g. `AGEX{coin}true` in place of `AG<coin>true`).
All these observations need to be taken into account when attempting the mechanization of the work of translating informal requirements into ACTL formulae.

4 An automatic translator from Natural Language

The first step in the construction of NL2ACTL has been the collecting of a *corpus*, i.e. a set of Natural Language expressions representative of the language to be characterized, with the related Temporal Logic expressions. For this purpose, we have generated a number of sentences expressing different temporal properties of systems of the "vending machine" class, trying on one hand to be as much as possible exhaustive, and on the other hand to extract as much information as possible in the form of the observations made before.
The corpus analysis allowed for the individuation of some regularities in the sentences. This lead to a distinction between:

a) structures strictly related to a possible temporal logic expression of the property (e.g., the structure "it is possible <action_phrase>", where <action_phrase> stands for a sentence fragment that individuates an action like "insert coin", corresponding to the formula `EX{coin}true`; another kind of structure is "if <action_phrase> then <sentence>", where <sentence> individuates a complete ACTL formula; so "if a coin is inserted then it is possible to recollect" corresponds to the formula: `[coin] EX {recollect} true`);

b) structures of segments of the sentences which express the occurrence of an action; these can be:
   b1) simple verb phrases, or fixed locutions of the type "action is performed", "action occurs", "action is possible", which can be considered general enough for every application domain;
   b2) typical complex sentences of an application domain, like "coin is inserted", "recollecting the coin", "get a tea" etc., in the vending-machine domain.

According to a general Natural Language Processing approach, the construction of the grammar had two main tasks: to recognize as many sentences as possible, and to define a semantics for each sentence. The former task has been pursued by the identification of as many structures as possible of the sentences related to ACTL formulae; the latter has been pursued through the identification of the semantics of each fragment of the sentences with the translation into the corresponding ACTL formula. The semantics of a whole sentence is therefore compositionally constructed from the semantics of every single fragment.

A main novelty of this tool is that it differentiates from a pattern-matching approach, where pre-defined structures are selected to represent the structure of input sentences (see, for example, [13], where this approach is used for an automatic derivation of Abstract Data Types from informal specifications). In such an approach the possible alternative is between recognized patterns and

unknown patterns. In NL2ACTL input sentences are analysed word-by-word, and it is possible to individuate and recognize meaningful portions of them.

4.1 The development environment

NL2ACTL has been developed by using a general development environment, PGDE, aimed at the construction, debugging and testing of Natural Language grammars and dictionaries and which permits, basing on the given grammar, to build an application recognizing input Natural Language sentences and producing their semantics [12]. PGDE allows to choose, without restrictions of any kind, the type of semantics one prefers. Furthermore in this environment it is possible to choose (and implement) any grammar formalism, being the parser totally independent from the chosen syntactic representation; for instance a great amount of work has been carried out in the implementation of Lexical-functional Grammars [7].

One of the main characteristics of PGDE is the possibility to write and develop a particular kind of augmented phrase-structure grammar called Process Grammar, PG. A Process Grammar is constituted by rules which are managed, in the generated application, by the Process Grammar Processor, PGP, as processes to be scheduled, depending on certain events which take place during the parsing. A module of the parser, the process scheduler, is in charge of scheduling under certain conditions, like the state of the processes and their execution priorities with respect to others (together with some efficiency factors that will not be described here). Thanks to this interpretation of rules as processes, it is possible, in a Process Grammar, to realize various strategies for the activation of the rules, depending much on the state of the rules and on the priorities assigned to the various processes. These mechanisms can be realized through the use of a set of functions which can be invoked by the rules/processes, called Kernel Functions. The PG Processor uses a PG which, on its turn, can use two sets of functions: the functions for the Feature Structures which manage the syntactic and semantic attributes of the nodes of the parsing structure; and the Kernel Functions, through which it can interact with the PGP's control structures. The output of the PGP is not defined a priori, and the grammar designer can choose the kind of structure returned by the parsing process. The PGP builds a parsing structure named Parse Graph Structure, PGS, which is basically a classic parsing tree structure; starting from the input sentence, the trees are built bottom-up.

A further characteristic of PGDE is the possibility to choose the level of interaction which must be allowed by the application: a naive user can be interested only in the recognition of his sentences as belonging to the language generated by his grammar, while a more expert one, especially in case the sentence parsing fails, can be interested in analysing the derivation tree to find where the parser has failed, to have partial analyses etc.

The final application, in our case NL2ACTL, is totally independent from the development environment itself, and can run therefore in isolation.

4.2 Use of the tool

After the NL2ACTL application has been started, the user can choose to operate interactively or to

give in input a file containing a set of sentences, the requirement document. To each sentence, be it typed separately or stored with other sentences in a file, corresponds, if it is recognized, an immediate ACTL translation. The output is presented to the user, which is supposed to understand its formal meaning and to check whether this corresponds to what was the intended meaning of the informal requirement; if it is not, it is likely that either the requirements was not precisely stated, or its intended meaning was not understood by the user. In both cases, a rewriting of the informal requirement is advisable. If no analysis can be derived, the writing "Parses not found" appears under the sentence, warning the user that the sentence does not belong to the language which can be translated into ACTL formulae. In the case the sentence allows for more than one ACTL formulae as its translation, all of them will be returned. The input sentences and the output formulae appear on distinct windows (see Fig. 4).

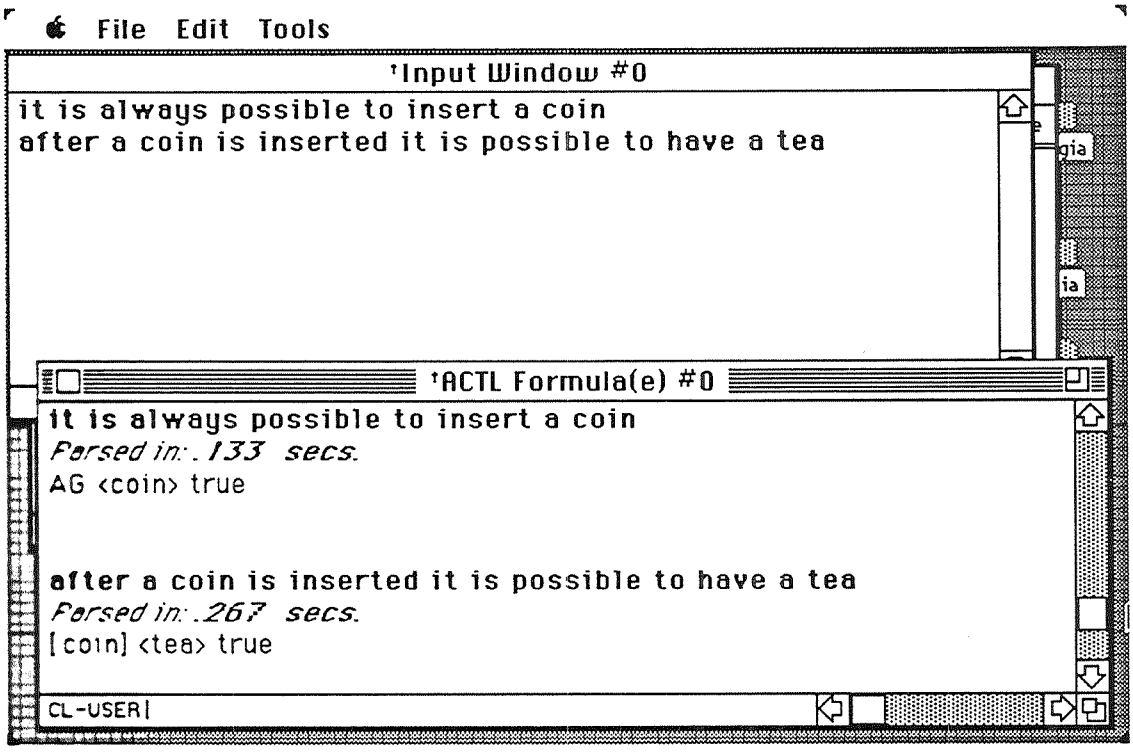


Fig. 4 Input-Output Windows of NL2ACTL

It is obvious that no notion of correctness of the formal description of a requirement with respect to its informal description can be defined; only the experience of the user can tell whether they are not conflicting; anyway, we believe that such a tool has at least the effect to force the specifier to a very accurate consideration of the informal requirements.
The NL2ACTL output for the requirements listed in section 2 is the following:

```
after a coin has been inserted it is possible only to have tea or coffee or recollect the coin
Parsed in: .767  secs.
[coin] A [true {false} U {((tea | coffee) | recollect)} true]
```

```
after a coin has been inserted it is not possible to have a soup until another coin is inserted
Parsed in: .500  secs.
[coin] A [true {~soup} U {coin} true]

if a coin is inserted then after another coin is inserted it is possible to have soup
Parsed in: .317  secs.
AX {coin} true -> [coin] <soup> true
[coin] [coin] <soup> true

it is always possible to insert a coin
Parsed in: .133  secs.
AG <coin> true

it is always possible to recollect a coin
Parsed in: .200  secs.
AG <recollect> true

if it is possible to have a tea then it is possible to have a coffee
Parsed in: .700  secs.
<tea>true -> <coffee>true

after a coin has been inserted eventually it will be possible to have a tea
Parsed in: .300  secs.
[coin] AF (EX {tea} true)
```

The NL2ACTL output reflects exactly what we expected, but the translation of the third requirement. In fact, in this case, the translation gives back two non equivalent formulae. This is due to the inherent ambiguities of the used sentence; it can be interpreted both as a logical consequence or as succession of events. It will be the user responsibility to choose the right one. Weak interpretation of the requirements is provided; there is however the possibility to use NL2ACTL on a different grammar, which provides the corresponding strong interpretation.

5 Conclusions and future works

The presented translation tool is the outcome of the application of the current state of the art in research in Natural Language Processing to the case of expressions of Temporal Logic properties.

NL2ACTL represents one of the very few examples of application to a "real" domain of a development environment which was designed and developed primarily to deal with research matters on Natural Language Processing. The good level reached by the application in the amount of recognized sentences, the correctness of the translation into ACTL formulae, the friendliness of the whole tool emphasizes PGDE's double nature: it is an environment where it is possible to carry out research in the Natural Language Processing field and to build applications usable in several possible domains. This suggests the opportunity to continue in both these directions (research and development). An application which can be compared to NL2ACTL is the one developed for the ESPRIT Project 527 "CFID" (Communication Failures In Dialogues) [11, 15], used to interface an SQL data base with Natural Language queries, and treated complex linguistic phenomena as misconceptions, different kinds of ellipsis etc.

The current version of NL2ACTL runs on a Macintosh, and the integration with the ACTL model checker described in [3] is via cut and paste of formulae among the NL2ACTL windows and windows connected to the Unix-running model checker: a closer integration is planned in the next future.

A deep analysis of the interaction between ACTL and Natural Language has been necessary for the construction of the tool NL2ACTL; this puts in evidence the possibility to extend the set of sentence structures related to ACTL expressions, and particularly the structures of segments of the sentences which express the occurrence of an action.

Another possible extension is to give by the possibility to the user to define his own set of actions and action phrases, and hence his particular domain. This is possible making the grammar parametric, and providing a preliminary dialogue with the user to establish his set of action (phrases).

In order to realize these extensions more experimentation is needed, especially on different and wider domains of application, so to achieve a sufficiently general corpus to be encoded in the grammars and dictionary of the tool.

References

[1] A. Bouajjani, S. Graf, J. Sifakis: A Logic for the Description of Behaviours and Properties of Concurrent Systems. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, (de Bakker, J., de Roever, P. and Rozenberg, G. eds.) Lecture Notes in Computer Science 354, Springer-Verlag, 1989, pp. 398-410.

[2] E.M. Clarke, E.A. Emerson, A.P. Sistla: Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Toplas*, **8** (2), 1986, pp. 244-263.

[3] R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori: An action-based framework for verifying logical and behavioural properties of concurrent systems. *Proc. of 3rd Workshop on Computer Aided Verification*, July 1-4, 1991, Aalborg, Denmark.

[4] R. De Nicola, F. W. Vaandrager: Three Logics for Branching Bisimulations (Extended Abstract) in *LICS '90*, Philadelphia, USA, June 1990, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 118-129.

[5] R. De Nicola, F. W. Vaandrager: Action versus State based Logics for Transition Systems. *In Proceedings Ecole de Printemps on Semantics of Concurrency*, April 1990, (I. Guessarian ed.), Lecture Notes in Computer Science 469, 1990, pp. 407-419.

[6] M. Hennessy, R. Milner: Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, **32** (1), 1985, pp. 137-161.

[7] R. M. Kaplan, J. Bresnan: Lexical-functional grammar: A formal system for grammatical representation. In J.Bresnan (ed.), *The Mental Representation of Grammatical Relations*, Cambridge, MA: MIT Press, 1982.

[8] D. Kozen: Results on the Propositional μ-calculus, *Theoretical Computer Science*, **27** (2),1983, pp. 333-354.

[9] R. Langerak: Decomposition of functionality: a Correctness Preserving LOTOS Transformation. *Protocol*

Specification, Testing and Verifications, X, L. Logrippo, R.L. Probert, H. Ural, eds., North-Holland, 1990, pp. 229-242.

[10] K.G. Larsen, "Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion ", *Theoretical Computer Science*, 72, 1990, pp. 265-288.

[11] M. Marino: A framework for the Development of Natural Language Grammars. *International Workshop on Parsing Technologies*, Carnegie Mellon 1989, pp.350-360.

[12] M. Marino: PGDE: Process Grammar Development Environment. User Manual. AITech TR#1/92 - PGDEUM. Pisa, 1992.

[13] K. Miriyala, M.T. Harandi: Automatic Derivation of Formal Software Specifications from Informal Descriptions. *IEEE Transactions on Software Engineering*, vol. 17, n. 10, Oct. 1991, pp. 1126-1142.

[14] R. Milner: "A Calculus of Communicating Systems",.*Lecture Notes in Computer Science* vol. 92, 1980.

[15] R. Reilly, I. Prodanof, G. Ferrari, M. Marino, A. Saffiotti, E. MacAogain, N. Sheehy: CFID: a robust man-machine interface system. *Atti del Primo Congresso dell'Associazione Italiana per l'Intelligenza Artificiale*, Trento, 1989

[16] C. Stirling: Temporal Logics for CCS, in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, (de Bakker, de Roever and Rozenberg, eds.) Lecture Notes in Computer Science 354, Springer-Verlag, 1989, pp. 660-672.

[17] R. J. van Glabbeek, W. P. Weijland: Branching Time and Abstraction in Bisimulation Semantics (extended abstract). In Information Processing '89 (G.X. Ritter, ed.), Elsevier Science Publishers B.V. (North Holland), 1989, pp. 613-618.

Appendix

CCS Syntax and Semantics

CCS (Calculus for Communicating Systems) [14] describes the behaviour of processes and their parallel composition, employing a finite alphabet of observable actions.
The syntax and the semantics of CCS are sketched in Fig. 5; there 'μ' ranges on a finite non empty set of actions, 'x' ranges on a finite set of variables; moreover we will use p, q (with indexes) as ranging over processes, to be understood as the behaviour expression representing a CCS process. The semantics of CCS is based on the concept of Labelled Transition Systems.

A *Labelled Transition System* (or *LTS*) is a 4-tuple $\mathcal{A} = (Q, Act \cup \{\tau\}, \rightarrow, 0_Q)$ where:

- Q is a set of *states*;
- Act is a finite, non-empty set of *visible actions*; the *silent action* τ is not in Act;
- $\rightarrow \subseteq Q \times (Act \cup \{\tau\}) \times Q$ is the *transition relation*; an element $(r,\alpha,q) \in \rightarrow$ is called a *transition*, and is written as $r - \alpha \rightarrow q$;
- 0_Q is the initial state.

We let $A_\tau = Act \cup \{\tau\}$; $A_\varepsilon = Act \cup \{\varepsilon\}$, $\varepsilon \notin A_\tau$. Moreover, we let r, q, s ... range over states; a, b, ... over Act; α, β, ... over A_τ and k, ... over A_ε.

CCS Syntax

p::= **nil** | μ. p | p\\α | p[ϕ] | p|p | p+p | x | **rec** x. p

where $\mu \in Act \cup \{\tau\}$

CCS Semantics

act) $\mu.p \vdash \mu \rightarrow p$

res) $p1 \vdash \mu \rightarrow p2$ and $\mu \notin \{\alpha, -\alpha\}$ imply $p1\backslash\alpha \vdash \mu \rightarrow p2\backslash\alpha$

rel) $p1 \vdash \mu \rightarrow p2$ implies $p1[\phi] \vdash \phi(\mu) \rightarrow p2[\phi]$

sum) $p1 \vdash \mu \rightarrow p2$ implies $p1+p \vdash \mu \rightarrow p2$ and $p+p1 \vdash \mu \rightarrow p2$

com) $p1 \vdash \mu \rightarrow p2$ implies $p1|p \vdash \mu \rightarrow p2|p$ and $p|p1 \vdash \mu \rightarrow p|p2$

$p1 \vdash l \rightarrow p2$ and $p'1 \vdash \neg\lambda \rightarrow p'2$ imply $p1|p'1 \vdash \tau \rightarrow p2|p'2$

rec) $p1[\mathbf{rec}\ x.p1/x] \vdash \mu \rightarrow p2$ implies **rec** $x.p1 \vdash \mu \rightarrow p2$.

Fig. 5: CCS Syntax and Semantics

The Satisfaction Relation for ACTL

Paths over a LTS, $\mathcal{A} = (Q, Act, \rightarrow, 0_Q)$, are now introduced:

- A sequence $(q_0,\alpha_0,q_1)\ (q_1,\alpha_1,q_2)\ldots \in \rightarrow^\infty$ is called a *path* from q_0; a path that cannot be extended, i.e. is infinite or ends in a state without outgoing transitions, is called a *fullpath*; the *empty path* consists of a single state $q \in Q$ and it is denoted by q;
- if $\pi = (q_0,\alpha_0,q_1)\ (q_1,\alpha_1,q_2)\ldots$ we denote the starting state, q_0, of the sequence by *first(π)* and the last state in the sequence (if the sequence is finite) by *last(π)*;
- concatenation of paths is denoted by juxtaposition: $\pi = \rho\theta$; it is only defined if ρ is finite and last(ρ)=first(θ).

When $\pi = \rho\theta$ we say that θ is a *suffix* of π and that it is a *proper suffix* if $\rho \neq q$, $q \in Q$. We write path(q) for the set of fullpaths from q and let π, ρ, σ, η range over paths.

The actual models that we use are assumed to have only infinite fullpaths; it is not a strong constraint since it is always possible to extend finite fullpaths with τ-loops. This choice is dictated by a requirement of the verification environment where we are working.

The collection *Afor* of *action formulae* over Act is defined by the following grammar where χ, χ', range over action formulae, and $a \in Act$:

$$\chi ::= a \mid \sim\chi \mid \chi \mid \chi'$$

We write `true` for a_0 | ~a_0, where a_0 is some arbitrarily chosen action, and `false` for ~`true`. An action formula permits expressing constraints on the actions that can be observed (along a path or after next step); for instance, a | b says that the only possible observations are a and b, while true stands for "all actions are allowed" and false for "no actions can be observed", that is only silent actions can be performed.

The *satisfaction* of an action formula χ by an action a, notation $a \models \chi$, is defined inductively by:

- $a \models b$ iff $a = b$;
- $a \models \sim\chi$ iff $a \not\models \chi$;
- $a \models \chi \mid \chi'$ iff $a \models \chi$ or $a \models \chi'$.

The satisfaction relation for ACTL formulae is given below.

Let $\mathcal{A} = (Q, Act, \rightarrow, 0_Q)$ be a LTS. *Satisfaction* of a state formula φ (path formula γ) by a state q (path ρ), notation $q \models_{\mathcal{A}} \varphi$ or just $q \models \varphi$ ($\rho \models_{\mathcal{A}} \gamma$, or $\rho \models \gamma$), is given inductively by:

- $q \models$ `true` always;
- $q \models \sim\varphi$ iff $q \not\models \varphi$;
- $q \models \varphi$ `&` φ' iff $q \models \varphi$ and $q \models \varphi'$;
- $q \models$ `E`γ iff there exists a path $\theta \in$ path(q) such that $\theta \models \gamma$;
- $q \models$ `A`γ iff for all paths $\theta \in$ path(q) $\theta \models \gamma$;
- $\rho \models$ `[`φ `{`χ`}U{`χ'`}` φ'`]` iff there exists $\theta = (q,a,q')\theta'$, suffix of ρ, s.t. $q' \models \varphi'$, $a \models \chi'$, $q \models \varphi$ and for all $\eta = (r,\beta,r')\eta'$, suffixes of ρ, of which θ is a proper suffix, we have $r \models \varphi$ and ($\beta \models \chi$ or $\beta = \tau$);
- $\rho \models$ `[`φ `{`χ`}U` φ'`]` iff there exists a suffix θ of ρ s. t. first(θ) $\models \varphi'$ and for all $\eta=(r,\beta,r')\eta'$ of which θ is a proper suffix we have $r \models \varphi$ and ($\beta \models \chi$ or $\beta = \tau$);
- $\rho \models$ `X{`χ`}`φ iff $\rho = (q,a,q')\theta$ and $q' \models \varphi$ and $a \models \chi$;
- $\rho \models$ `T`φ iff $\rho = (q,\tau,q')\theta$ and $q' \models \varphi$.