Aboard AUTO

Robert de Simone, Didier Vergamini

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tél (1) 39 63 55 11

Rapports Techniques

N° 111

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

ABOARD AUTO

Robert de SIMONE
Didier VERGAMINI

Octobre 1989

RT_0111

Aboard Auto
Aborder Auto

Robert de Simone
INRIA
Sophia-Antipolis
rs@cma.cma.fr

Didier Vergamini
CERICS
Sophia-Antipolis
dvergami@cma.cma.fr

Abstract

This User Operating Manual describes the practical aspects of AUTO. AUTO is a verification system for process calculi terms. It incorporates increasingly many features, starting with bisimulation equivalence proving.

AUTO deals mostly with finite automata, and pretends to treat partial verification as much as possible as an homogeneous activity: statements themselves consist in reduced automata obtained by "slicing" in an homomorphic fashion the original one. The means to attain this is through the notion of *abstraction criteria*.

The present Manual describes syntax and functionalities, but does not try to instill pragmatics or disciplines in the use of AUTO or process calculi modelization in general. This has been long undertaken in a variety of articles, recalled in the bibliographic last section.

Résumé

Ce Manuel d'Utilisation décrit les aspects pratiques d'AUTO. AUTO est un système de vérification de termes de calculs de processus. Il contient aujourd'hui un nombre grandissant de fonctionalités, à partir de la notion initiale de preuve de bisimulation.

AUTO manipule principalement des automates finis, et tente au maximum de traiter les vérifications partielles de cette manière homogène: les propriétés sont elles-mêmes considérées comme des automates, obtenus par des "coupes" de nature homomorphiques, dans l'automate de départ. Ceci est réalisé par la définition des *critères d'abstraction*.

Le présent Manuel décrit la syntaxe et les fonctionnalités d'AUTO, à l'exclusion des considérations méthodologiques et pragmatiques de bonne utilisation, sur AUTO ou les calculs de processus en général. Une liste d'articles traitant d'exemples sur ces thèmes est contenue dans la section bibliographique.

1 Foreword

This handbook is meant for people who care about Process Calculi. The theory of Process Calculi [Mil 80] aims at providing both a syntax and a sound semantics to model concurrent systems at a certain level of mathematical abstraction, in an algebraic framework. It has now gained wide recognition in this domain, and gave rise to a whole field of study in its own.

And so AUTO is specifically designed at algebraic manipulations of such expressions. It limits itself to the finitary case, and constructs their underlying automaton. It also performs on demand a range of reductions on these automata. Actually since the early days these reductions were asserted as a general philosophy, with reduced (or "abstracted") versions of automata considered as partial statements on them. This leads to a discipline of verification where specification of correctness and realising processes share the same semantic domain of automata. As these are highly graphical objects results may be displayed the very same way terms were provided.

AUTO is fronted by the graphical editor AUTOGRAPH[RS 89], where both automata and process networks may be edited. Reduced automata obtained by AUTO may also be drawn in AUTOGRAPH under human guidance.

AUTO has been endowed with lots of side functions which bridge the gap from theoretical verification theory to everyday life debugging for the best of our intend. Still this is an evergrowing activity which needs more practice on modelling large systems to reach its ends to meet full satisfaction.

AUTO was found a valuable tool not only for process calculi, but also for any language providing automata as reactive systems. It is now widely used in connection with Esterel [BG 87] to observe automata.

Contents

2 The objects

The range of objects types manipulated in AUTO is rather small. Some of these objects are user-provided, and entered as input to the system. Others are produced by AUTO as results of computation. They may themselves be further analysed.

2.1 User-provided Objects

They range upon:

- *actions and signals* (and lists of them),
- *process calculi terms* (for the time being MEIJE terms),
- *abstraction criteria.*

Objects of the first two types can also be produced by AUTO, for instance as *sorts* of processes for signal lists, or as readable format of automata for process terms.

Objects of these classes may be supplied to AUTO by using dedicated parsing commands so that they are syntactically checked. Actually the parser of both actions and process terms is shared, while criteria are recognized with a different parser.

2.1.1 Actions

They represent the basic behaviours of systems, consisting of communication attempts. They shall become the labels of automata transitions. Actions are thus generated by emission/reception signals together with atomic actions, and then built by possibly aggregating them in simultaneous co-ocurrences products. Along with actions come structures like actions lists.

Concrete syntax for actions and related structures is:

labels
: They are the atomic signals basenames. They consist of alphanumerical strings, possibly containing - (dash) and _ (underline) characters, and must start with a letter.

hidden action
: The string `tau` is reserved and denotes the invisible, or "hidden" action corresponding to the theory of `weak bisimulation`.

emission signals
: They are just consisting of *labels*, possibly ending with an optional "!" character. One may also indicate a multiple simultaneous emission of a signal by grouping as many "!" as needed, as in `emit!!!`. Another alternative syntax for this is `emit^3`.

reception signals

They are consisting of *labels* suffixed with one or several "?". Equally, one may type `receive^-3` instead of `receive???`.

compound actions

They are formed with "." (dot) separated lists of emission and reception signals, as for instance in "`receive?.transmit^2.token!`"

lists

Lists of actions are used by several AUTO functions. They are comma separated, braces spanned, as in {`emit!!, receive?.transmit`} for instance.

2.1.2 Process Calculi Terms

They form an algebra based on primitive operators. The operators available in AUTO are those of MEIJE, which closely resemble those of CCS, but for an additional semantic rule for the parallel operator: arbitrary simultaneous co-occurrences of actions are allowed (and not only when they are mutually inverse signals).

In order to obtain the required finitary conditions, since AUTO only manipulates finite automata, the algebraic structure is split in two levels:

Sequential regular terms

At a first level, "non-deterministic sequential" *regular* terms are produced, with a syntax directly representing automata through right-linear grammars. Still there may be free variables in those terms, to be matched against previously computed automata in the global toplevel environment.

In the sequel we shall let R, R_1, ..., R_N range over syntactic terms of this first level of syntax. Concrete syntax for regular processes is:

stop

for the term without behaviours.

a:R

for the process performing an a action before reconfigurating into R.

$R_1 + R_2$

for the process which behaves either as R_1 or R_2 on demand. Sums should imperatively be guarded, that is all variables occurring in either R_1 or R_2 should be prefixed by an action somehow.

variables

They consist of alphanumerical strings, possibly containing - (dash) and _ (underline) characters, and must start with a letter. This was the lexicographic class of actions labels already. Process variables are supposed to be bound,

either statically to a previously defined global variable containing an already compiled automaton, or to their surrounding recursive definition.

let rec $\{ \mathbf{X}_1 = R_1$ **and ... and** $\mathbf{X}_N = R_N \}$ **in** R

for recursive definitions of variables $\mathbf{X}_i$.

Example1: a process representing a boolean variable.

```
let rec
  {False = write_false?:False
         + read_false!:False
         + write_true?:True
and
  True = write_true?:True
       + read_true!:True
       + write_false?:False}
in True
```

Networks

At a second level another set of syntactic constructs is used for building networks setting subcomponents in parallel, with proper communications and scoping of signals visibility. This level of syntax is also known as building *flowgraphs*.

In the sequel we shall let P, P_1, ..., P_N range over all syntactic terms.

let $\{$**X1** $= P_1$ **and ... and XN** $= P_N \}$ **in** P

for local non–recursive definitions of terms variables.

P_1 // P_2

for parallel composition of *P1* and *P2*.

P**\siglab**

for restriction of a process P on a signal label. This limits the scoping of the signal communication by forcing it to happen inside of this term. Restriction on several signals at once is simply `P\siglab1\siglab2 ... \siglabN`.

P**[act1/siglab1,act2/siglab2, ..., actN/siglabN]**

for the term P in which where atomic signals labels `siglab1, siglab2, ..., siglabN` (representing signals with "emission polarity") are substituted by possibly compound `a1, a2, ..., aN` actions.

a$\star P$

for the process P piloted — or driven — by an action a. This means that this action is added as a simultaneous action to whatever action P may perform. In particular this can turn the invisible τ action to become noticeable again!

Example2: A process representing a scheduler on 3 cyclers, extracted from [Mil 80].

```
(((cycler [beta3 /beta, alpha3 /alpha,
           sig_1 /myTurn, myTurn /Next])
  //
  (cycler [beta2 /beta, alpha2 /alpha,
           Next /myTurn, sig_1 /Next])
  //
  (cycler [beta1 /beta, alpha1 /alpha])
  //
  ((let rec
      {st_1 = Start!:st_2
    and
      st_2 = stop}
    in st_1) [myTurn /Start]))
    \ sig_1 \ Next \ myTurn)
```

These two levels of terms syntax closely correspond to the two modes of edition in AUTOGRAPH.

Terms may of course be parenthesized to overwhelm the default binding powers. The default binding priority is that unary operators bing stronger than binary ones, and that ***restriction*** and ***renaming*** bind stronger than ***prefixing*** and ***piloting.***

2.1.3 Abstraction Criteria

Abstraction Criteria are finite collections of *abstract actions.* In a sense, they provide for a new alphabet of actions, with less fine grain of atomicity as the concrete former actions, and on which an abstracted version of the automaton will be extracted. In essence this new automaton will be simpler, with many details forgotten. Thus it shall "give an angle" on a specific property to be checked for. Interestingly, unlike for temporal logics formulae one needs not know exactly what the relevant property is, but rather focus on it with more and more precision by playing around composing criteria. Still the theory of criteria is perfectly founded on the semantic side. So one always knows how statements on the reduced automaton should be interpreted going back to the original one, which more directly represents the system to be observed. These observations led us to introduce the type of *pathes*, trying to retrieve original information in a practical format. Pathes will be described later on.

Concrete syntax for abstraction criteria is, for the time being (it is subject to improvement in the next future):

criteria

A criteria is just a comma separated list of abstract actions definitions. It must be bound to a name using the `parse-criterion` command.

abstract actions

An abstract action definition is of the shape "`name = body`". This definition

may not be recursive. That is, if ever an abstract action bears the same name as a concrete one, they shall be considered different. In particular, one may have τ as an hidden abstract action. The body is a regular expression based on basic predicates as generators. Regular expression syntax is simply:

Abstr$_1$ + Abstr$_2$
for set union

Abstr$_1$: Abstr$_2$
for sequential product

Abstr *
for Kleene star.

For instance "`A = tau*:/a:tau*`" is a well formed abstract action.

basic predicates
They range along:

act
for each act "concrete" action names, to stand for identity predicate to this action,

/ siglab
for the "divisibility by a signal **siglab**" predicate,

or, and, not
to form boolean combinations of the precedent two predicates.

Predicates may be parenthesized to overpass usual binding powers.

The following criterion was drawn from an example of ISO transport layer specification.

```
A=(tau *):
  (((not / CONres_A_B) and (not / CONcon_B_A))*):
  (tau *),
B=(tau *):
  (/CONres_A_B + /CONcon_B_A):
  (tau *):
  ((/DATreq_A_B + /DATind_B_A)*):
  (tau *):
  (/DATreq_A_C + /DATind_C_A + /CONind_C_A
     + /CONreq_A_C + /DISind_C_A
     + /DISreq_A_C + /CONres_A_C + /CONcon_C_A):
  (tau *):
  ((/DATreq_A_C + /DATind_C_A + /CONind_C_A
     + /CONreq_A_C + /DISind_C_A
     + /DISreq_A_C + /CONres_A_C + /CONcon_C_A)*):
  (tau *):
```

```
    ((/DATreq_A_B + /DATind_B_A)*):
    (tau *):
    (/DISreq_A_B + /DISind_B_A):
    (tau *),
  C=(tau*):
    (/CONres_A_B + /CONcon_B_A):
    (tau *):
    ((/DATreq_A_B + /DATind_B_A)*):
    (tau *):
    (/DISreq_A_B + /DISind_B_A):
    (tau *);
```

2.2 AUTO generated Objects

The nature of AUTO is to compute functions on these input objects, of types actions, process terms or criteria. It then returns other objects as output, this time with one of the following types:

- *automata*, which as we saw could be viewed to a certain extend as a subtype of this of terms,
- *actions lists*,
- *pathes*, or sequences of transitions, and
- *partitions*, or equivalence classes of states referring to a given automaton.

Out of these, the type of automata is certainly the most important, while others are just means of recollecting informations on these automata for better comprehension and analysis.

None of these types really owes a fancy syntax for reading, as they are not supposed to be provided by the user. There are AUTO functions to print them about nicely, but mostly they are intended in the next future to be graphically displayed in AUTOGRAPH(which may require human help). One may also use further utilitary function to extract even finer information from them (like " which class in a partition does a state belong to" for instance...).

We shall now sketch a number of simple considerations applying to the intuition behind the usefulness of such objects in AUTO. We hope to hint at how they may bridge the gap in between rather theoretical functionalities in parallel systems verification, in one hand, and on the other practical questioning of the results, specially when negative, to build up to *ad-hoc* debugging techniques in the domain of parallelism where a new kind of mistakes are raising up by nature.

2.2.1 Automata

Constructing automata from terms is AUTO's primary function. It does so following the celebrated *Structural Operational Semantics*. Automata have an internal representation to allow fast operations on them as required by the algorithms in AUTO's engine. There is a variety of functions performing these automata constructions, since most of them apply reductions at intermediate levels according to what congruences properties allow. Sizes may then be brought down in the course of construction, which is of great importance for the ulterior algorithms.

Functions range over simple automaton construction, construction of quotients both up to strong and weak bisimulation, construction of quotients parameterised by abstraction criteria, construction of quotients up to trace language semantics, and a couple of others of less importance. The latter are provided to the user so that he may perform step by step some of the inner algorithms involved in the preceding ones, setting several tracing booleans, and learn more about where space and time were spent. Then the succession of performed functions may be tuned so as to improve efficiency. This makes AUTO a (small) programming language.

In AUTO automata may be explored or displayed as terms. They may also be passed to AUTOGRAPH for human-guided graphical displays, or further observed in AUTO to provide perceptive informations. For instance reduced versions may be extracted using abstraction criteria. *Deadlock* states may be listed, as may *sorts* and lists of possible labelling actions.

2.2.2 Pathes

Pathes are sequences of transitions, with states alternated by performed actions leading from one to the next.

They are a means of diagnostic to recover information after playing the *criteria* reduction game: typically, one first reduces down some automaton, abstracting about those behaviours which shall not be relevant to this aspect the user has in mind. As the size becomes smaller, one is perhaps in the end able to put in evidence either some state as a specially important one —a partial deadlock for instance—, or a path connecting two states while it should not. But then one would need to retrieve this information on the former automaton, prior to reduction. Using *pathes* actually achieves this. A further step would be to recollect this path in its distributed form in the starting term itself, with indication of what behaviours components share in the action. This is not done yet.

2.2.3 Partitions

Partitions are made out of an automaton name together with a list of lists of states. Each list of states amounts for an equivalence class. In case the partition is incomplete, all remaining states are supposed to belong to the same additional class.

There are utilitary functions for regaining information about which class a state falls in (so that one may easily check whether two states are equivalent). One may also build partition "by hand", manipulating lists of states and then linking the result to a given automaton. Partitions may also be used as an imperative starting case for the strong bisimulation refinement algorithm.

3 Entering the fair world of Auto

3.1 Variables and Commands.

AUTO consists in a main toplevel loop. The user is prompted for *commands*. Commands typically consist in computing a result, or parsing a term, and then store the result in a global variable, or *identifier*. Variables bound to objects form a global environment (in a ML-like fashion). Bindings are static, so that free variables appearing in a command are automatically bound to their previous value in the then environment. In case the variables weren't found a syntax error is raised. Identifiers are stored in the same environment, no matter the type. Then no two objects bearing the same name but of different types may be present at the same time in the toplevel environment. One cannot name a the process `a:stop`.

Command are –possibly several input lines long– strings terminated with a semicolon. Errors are recovered typing *two* semicolons in a row. To exit AUTO, just type `end;` as a command.

3.2 Binding the variables.

The way to bind variables is two-fold, depending on the object to be bound. Actions, MEIJE/CCS terms and abstraction criteria need to be parsed. Then the command obeys the syntax:

```
parse action_var = <actions>;
parse term_var = <term>;
```

or

```
parse-criterion crit_var = <criterion>;
```

respectively. Actions and terms calls for the same parser. AUTO shiftes its prompt character when a parsing command is not over while typing a `carriage-return`.

In other cases, a global variable may only be bound to the result of a –possibly composed– function computed in AUTO. The list of all AUTO functions is to be found in section 5.

Some global variables names are reserved, and correspond to *Control flags*, of boolean nature (see their list in section 6. They may be assigned to `true` or `false` by a command. We see this as applying a nullary function.

When no specific parser is to be called, the command shapes as:

```
set var = ...;
```

As a result to performing the command, AUTO shall print the name of the variable together with the type that was inferred to it. The actual value, or content, that was

computed is itself not printed, as it may outstretch what is decently readable on the screen. In case of automata there are functions which let the user figure out their sizes without printing them.

3.3 Showing the result

In any case you may visualise the content of a variable using a command of the form:

```
show <var>;
```

Actually you may also visualise the result of applying a function without binding it to a variable by typing

```
show function(arguments);
```

For structured objects, like automata and terms, the use of `show` is not advised. The user should rather use the `display` command (see section 5).

In case one simply wants to see the result of a function without actually binding a variable to it, the command `show <functions>(<argumentlists>)` is perfectly alright. One may even recover the content of the last unassigned computation result, as it was stored in the dummy variable `it`. Just beware that `it` is **not** a synonimous to the last assigned variable, as in ML for instance.

3.4 Auditing the session

AUTO sessions may be recorded, with sequences of instructions interleaved with system's answers. This is done through the `audit` command, which requires a file name and adds an `.aud` suffix. This audit is ended by typing `close;` as a command.

3.5 Need Help?

There are two commands for recalling informations upon AUTO's syntactic features to the forgetful user.

The `help` command is the more complete of both. It may only be invoked on any keyword in AUTO(hereafter called "topics"), either *objects*, *command* or *function* name, *control flag* or optional format keyword. The lay-out of the `help` command is inspired from the Unix "man" command, even though much more primitive.

Informations upon concrete syntax commitments together with short indications about funtionalities are provided. The command `help;` on its own, without arguments, explains how to run it (with arguments). One may get a full list of available help topics by typing `help topics;` .

The `apropos` command gives a shorter presentation of these topics, usually just one line. It may be invoked with other, broader keywords to encompass a subset of

topics. This feature is of interest when the name of something in AUTO is *indeed* what is to be remembered. Typing apropos; without arguments provides a first range of fields to start sorting out the search.

The way apropos is implemented is also rather bare. It shall print all lines from a small database which somehow correspond to a given key. So the lay-out may seems odd at times, with unconnected lines in succession. But still the information is present...

3.6 Regarding Unix

To exit AUTO back to Unix, one types:

```
end;
```

Still, a few commands allow the user to execute very simple Unix commands without exiting AUTO. They will certainly become obsolete some day because of multiwindowing and other escaping orders that would pause AUTO from the Unix system itself. They are still worth mentioning for the time being.

One may execute any simple unix command by typing:

```
comline "command-text";
```

A set of commands deal with the call to a text editor from inside AUTO. This is done through the command:

```
edit "filename";
```

Without further notice, the called editor is *Emacs*. To change this, one updates a specific internal string by typing:

```
default-editor "editor-name";
```

Now to how the files are spotted in the Unix directory systems: not all files need to be present in the directory AUTO was called from, be it for editing or actually loading into our system. Absolute pathes are alright, but also relative pathes to a certain list of ordered path prefixes that AUTO knows of. There is a default basic such prefixes initially in the system. It is printed by typing

```
search-path;
```

This command in general shall provide the search-path list in use presently. This list is changed with the two commands:

```
add-search-path "directory-prefix";
```

and

```
del-search-path "directory-prefix";
```

respectively. Notice the order is important. There are no commands for changing it. Each `directory-prefix` name should imperatively end with a "/" *(slash)* symbol, as the residual of the file name will simply be appended.

The user may reinsure himself about which actual file was taken into account by the preceding mechanism by typing

```
search-in-path "relative-file-name";
```

which shall produce the absolute path of the corresponding file (where it was found).

4 The input/output commands

There are two kinds of files which may be read, from files or screen: those which contain *objects*, and those which contain *sequences of instructions.* Only objects files may be printed, even though there is a way to record sessions with the `audit` mechanism.

4.1 Feeding AUTO

All three following commands shall make implicit use of the directory searching for files as described in 3.6 whenever needed.

load "InstructionsFileName"

Instructions are kept in Unix files suffixed with "`.ec`". There are read and executed in AUTO by typing the command `load "<filename>" ;` , without the suffix. This file may itself recursively calls for loading of other files –or objects even–. There is provision for inserting comments in the file text, while the reading is temporarily stopped. It is useful for automatic demos: at each line starting with a ">" symbol the system will be made to halt, scanning a portion of uninterpreted text down to a "<" symbol starting another line below. Loading is resumed by typing `<Return>`.

include "MeijeTermFilename"

MEIJE terms are kept in files suffixed with `.m0`. There is an AUTO function, named `include`, which requires a string representing a file name, without its `.m0` suffix, and then reads from this file instead of the usual input.
This function must of course imperatively be used inside of a `parse` command, so that a proper terms parser is turned on.

Abstraction Criteria may also be read from files using this function. The distinction is made to AUTO as then the function is used while in a `parse-criterion` command. In this case the `include` function will be turned to look for a file whose name is suffixed by `.crit` instead.

include-auto "AutomatonFilename"

Automata may be read in their internal form, after they were written down this way ("dumped"). In this case they are stored in files suffixed with "`.au`". This does not require to parse them back, and so proves much more efficient. This function is used while using a `set` or `show` command.

Files of that format are presently produced also by the Esterelv3 system, to which AUTO is therefore fully interfaced.

4.2 Getting it back

display identifier [OptionalFormat]

write "Filename" identifier [OptionalFormat]

The `display` command prints the content of the identifier on the screen, while printing on file calls for the `write` command. Then a file basename is required, to which a suffix is appended in case an optional format is prescribed by the user. We shall now detail these formats, pending on objects. The default format is a plain one which may not be reentered into any system. `display` may also be used in front of the keyword `globals`, which as a result prints the names of all existing variables at this time in the environment, together with their types. Similarly `display flags` prints the states of all boolean control flags.

meije

For terms only. Then a MEIJE process calculus term is generated, using a clever paragrapher for pretty-printing. Files produced with this format may be read back using `include`.

auto

For automata only. The automaton is then dumped all at once in its internal Lisp representation, fit being reentered with a call to `include-auto`.

autograph

When applied to an automaton, produces a format fit for exploration in the graphical editor AUTOGRAPH. When applied to a list of signals labels, produces an AUTOGRAPH net consisting of a single box with ports on its border, named with these labels. When applied to a path, produces a specific format to be visualised in AUTOGRAPH on the preexisting windows, with states jumps pictured on the component individual processes and global actions highlighted on the global network. This output format is still subject to change.

short

for automata only, provides only their size in number of states, transitions and labels.

5 The functions

The substance of AUTO is contained in the list of applicable functions. As they tend to become numerous we split them in two for description: the most evocative ones first, and the utilitary ones later on.

Arguments to functions should be put in between parenthesis. The single exception to this is the case of one argument consisting of a single string of characters (thus a constant or a variable in the environment), in which case parenthesis are not mandatory.

In the sequel, an argument put in between brackets in the type description of a function shall be meant as optional. Wherever we use the type of **Terms** an internal representation automaton could be provided instead.

5.1 Main functions

5.1.1 From Terms to Automata

These contain both construction and reduction functions. The merging of both activities in the same function allows to take benefit of congruences properties and distribute reductions on subterms.

All these functions allow for an optional second argument. It should be a list of labels, and represents the set of labels which are visible externally, any other being then implicitely renamed into τ.

tta(MeijeTerm [,ActionsList]): Automaton
: short for "term to automata", performs the full expansion of terms into automata, using the semantic rules, without performing any sort of reduction whatsoever. One thus gets a global system which in the worst case is in size the product of component sizes, but most often is not, if there is to be synchronisation involved.

mini(MeijeTerm [,ActionsList]): Automaton
: same as before, but then performs a strong bisimulation reduction to minimal representative.

obs(MeijeTerm [,ActionsList]): Automaton
: performs the expansion of terms according to classical weak bisimulation congruence. A reduction to minimal form along this semantics is performed at every stage of construction after setting two subcomponents in parallel.

  The underlying algorithm runs in 3 phases: first all states belonging to the same strongly connected component for the τ-transition relation are merged into one; then a transitive closure of the remaining transitions providing the relevant $\tau^* : a : \tau^*$ behaviours is computed; last

reduction w.r.t. strong bisimulation is performed on the automaton based on these new relations.

tau-simpl(MeijeTerm [,ActionsList]): Automaton

similar to obs, but only the first step of the algorithm is performed, which shrinks states of the same τ-cycle together. The procedure is then guaranteed not to be too costly since no intermediate, eventually space consuming, structure is ever generated.

trace(MeijeTerm [,ActionsList]): Automaton

This function computes the *trace semantics* minimal automaton representative of a term, that is the minimal deterministic automaton recognizing the same language.

dterm(MeijeTerm [,ActionsList]): Automaton

This function computes a standard deterministic version of the automaton, without minimization.

exclusion(MeijeTerm [,SignalsListList]): Automaton

strips from the automaton all transitions whose labelling actions do not obey the following constraint, drawn from the provided list of signals list: the considered action may not contain two signals from the same list!

tau-sature(Automaton): Automaton

Performs the transitive closure of the τ transitions, furthermore computing the $\tau^\star : a : \tau^\star$ transition completion for all a action. The function takes an automaton as argument and provides a completed automaton back, with transitions added. It does not perform any reduction whatsoever.

refined-mini(Partition): Automaton

practices reduction along strong bisimulation semantics. It starts the partitioning algorithm involved from the given partition, working on the automaton contained in the partition.

5.1.2 From Automata and Criteria to Automata

abstract(Automaton, Criterion): Automaton

Takes an automaton and an regular abstraction criterion. Syntax for criteria was provided in section 2. The function constructs the new automaton on abstract actions as labels, without further equivalent states reduction.

5.1.3 From Terms Couples to Truth Values

eq(MeijeTerm, MeijeTerm): boolean

Directly provides a *true/false* answer on the bisimulation question, when applied to a couple of terms. Internally realizes both their minimizations, and compare the results. A foreseen improvement to this function is to provide with an explanatory temporal logic function –on request only !– in case of nonbisimulating terms.

obseq(MeijeTerm, MeijeTerm): boolean

Same as before, but with the weak bisimulation equivalence instead.

5.1.4 From Terms to Actions Lists

Providing with *sort* computing seemed just fair as a protection against mispelling of signals and misplacing of restrictions as scoping operators. It is safe to run it first on a term to check that unwanted signals do not surface to the outside.

sort(MeijeTerm) :LabelsList

Returns the usual sort of a term as a list of signals' and atomic actions' labels.

signed-sort(MeijeTerm) :SignalsList

Same as **sort**, but specifies the polarity of signals as they may get out of the term (only *received* or *emitted*, or both).

actions(Automaton) :ActionsList

Only works on compiled automata. Provides the full list of compound actions which may be performed, that is which are labels of transitions in the global system.

5.1.5 From Automata to Partitions

These functions provide the partitions internally computed by various algorithms described above. Partitions may further be queried using some utilitary functions.

strong-partition(Automaton, Criterion) :Partition

Takes an automaton and computes the partition of its equivalence classes of states *w.r.t.* strong bisimulation.

weak-partition(Automaton) :Partition

Same as before, but with the weak congruence.

crit-partition(Automaton) :Partition

Same as before, but with the congruence induced by a given criterion, to be handed to the function.

To go the other way around:

quotient(Partition) :Automaton

Constructs a new automaton from a given partition on states of a previous automaton. The set of transitions deduded is such that a transition in between two classes is exactly the union of all transitions from any state in the source class to any state of the target.

5.1.6 From Automata to States Lists

refusals(Automaton,[LabelList]) :StatesList

Is applied on an automaton with a possible list of signals. It forms the list of all states whose immediate possible behaviours all contain at least a signal from the optional list (and all deadlock states if no such list is present). For the time being it is not possible to set tau as part of the list.

initial(Automaton) :State

Provides the initial state of a given automaton.

5.1.7 From Automata to Pathes

path(Automaton [, StartState], TargetState) :Path

This function will return a loop–free path of shortest length leading from start state to target. If no start state is present, then the initial state is used instead.

5.2 Auxiliary functions

5.2.1 Lists manipulations

These functions owe their concrete syntax to Lisp.

car(List) :ListElement

returns the first element of a list.

cdr(List) :List

returns the list without its first element.

nth(List) :ListElement

returns the n^{th} element of the list.

append(List, List) :List

concatenates the first and second list together.

5.2.2 States manipulations

number(Automaton, StateName) :StateReference
returns an integer corresponding to the internal numbering of the state, when applied on its external name.

structure(Automaton, StateReference) :StateName
returns the external character string of a state name when provided its internal numbering.

The role of these functions is to allow the user to go back and forth in between automata. In a quotiented automaton, each class bears as its external naming the name of a representative state of this class, in the previous automaton before the reducing function took place.

row(Partition, State) : ClassRef
takes a partition and a state identification (be it internal reference or external name) and produces an integer indicating in which class of the partition the state falls.

class(Partition, ClassRef) :StatesList
takes an integer indicating a class order in the partition and provides the class itself.

partition(Automaton, StatesListList) :Partition
takes a list of lists of states, whose names refer to this automaton, and turns it into the proper internal representation of a partition.

6 Controls

In addition to its collection of various functions, AUTO makes use of several global boolean controls setting permanent options. These flags may also select the printing of size information so that the user may further evaluate where most time was spent, where to focus on improving matters, and other potential problems.

All the following flags are set to `false` initially when AUTO is started.

Algorithms flags

ccs
controls the interpretation of the parallel operator as this of the CCS process calculus, thereby discarding the use of simultaneity product. When on, it drastically reduces the combinatorial explosion. On the other hand, in many cases the product was used to cut down the states number in the early modelization of the problem.

pt

causes the `mini` and other similar functions performing a state partitioning along the bisimulation lines to use the so-called Paije-Tarjan algorithm instead of the naive one.

melhorn

causes the `tau-sature` and other similar functions performing a transitive closure algorithm on τ-transitions to use the so-called Melhorn algorithm instead of the naive one.

Sizes flags

timer

causes the printing of an (approximative) time measure for any command executed.

debug-algo

causes the printing of size triplets: *(states number, transitions number, actions number)* each time two automata representing subsystems are put in parallel, and each time the weak bisimulation `obs` function is invoked by the system.

debug-cycle

completes information provided by `debug-algo` in adding size triplets reached after intermediate reductions of the two kinds: collapsing of states on the same τ-relation strongly connected components (or "cycles"); and collapsing states with only one leaving transition, the label being τ, on the end point of the transition.

debug-gc

causes the printing of informations upon calls to Lisp garbage collector.

debug-state

causes the printing of new state numbers as they are created during execution of a parallel construction.

References

[Bou 85] G. Boudol *"Notes on algebraic calculi of processes", Logics and Models of Concurrent Systems, NATO ASI Series F13,K. Apt, Ed.(1985)*

[BSV] G. Boudol, R. de Simone, D. Vergamini, *" Experiment with Auto and Autograph on a simple case of Sliding Window Protocol", INRIA Research Report No. 870, (1988)*

[BG 87] G. Berry, G. Gonthier, *"The Esterel Synchronous Programming Language: Design, Semantics, Implementation", to appear in Comp. Sci. Prog. (1989)*

[CPS 89] R. Cleaveland, J. Parrow, B. Streffen, "A semantics Based Verification Tool for Finite State Systems", in *Proceedings of the Ninth International Symposium on Protocol Specification, Testing, and Verification*, North-Holland.

[Lec 89] V. Lecompte, *"Vérification Automatique de Programmes Esterel", Thèse de l'Université Jussieu Paris 7,* (1989)

[Mil 80] R. Milner *"A Calculus of Communicating Systems", LNCS 92, Springer-Verlag (1980)*

[RS 89] V. Roy, R. de Simone, *"An* AUTOGRAPH *Primer", I.N.R.I.A. Technical Report, to be published (1989)*

[Ver 86] D. Vergamini, *" Verification by means of observational equivalence on automata" INRIA Research Report No. 501, (1986)*

[LMV 87a] V. Lecompte, E. Madelaine, D. Vergamini, *"Auto Un système de vérification de processus parallèles et communicants" INRIA Technical Report No. 83, (1987)*

[Ver 87b] D. Vergamini, *"Vérification de réseaux d'automates finis par équivalence observationnelles: le système Auto", Doctorate Thesis (1987)*

[Ver 88] D. Vergamini, *"Verification of Distributed Systems: an Experiment", INRIA Research Report 934, (1988)*

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique