FAST — Fast Acceleration of Symbolic Transition Systems

Users' Manual

Laboratoire Spécification et Vérification
CNRS UMR 8643
École Normale Supérieure de Cachan
61, avenue du prsident Wilson
F-94235 CACHAN Cedex
FRANCE
http://www.lsv.ens-cachan.fr

Sébastien Bardin
Alain Finkel
Jérôme Leroux
Laure Petrucci
Laurent Worobel

July 18, 2003

LSV Technical Report LSV-10

Introduction

FAST [BFLP03] is a tool dedicated to the analysis of infinite systems. It performs automatic verification of systems augmented with (unbounded) integer variables, namely *extended counter automata*. The main issue addressed by FAST is the computation of the *exact* (infinite) set of configurations reachable from a given set of initial configurations. Although such sets are in general not computable, FAST uses a semi-algorithm which is expected to terminate in most practical cases.

The techniques used in FAST are based on *acceleration* [FL02], which involves computing the effect of iterating a control loop of arbitrary length. The models FAST operates on are composed of a finite control structure, guards are Presburger formulae, and operations affine functions. The user can provide a strategy to direct the reachability set computation. Strategies are written in a high-level language, which supports, for the moment, forward and backward reachability as well as more advanced constructs such as incremental sub-model analysis.

FAST has been applied to a large number of examples, available on the FAST web pages. In about 80% of these case studies the reachability set could effectively be computed with a basic predefined strategy. A more elaborate strategy, using incremental sub-model analysis, can lead to a considerable drop in computation time for a complex protocol.

Contents

Chapter 1

FAST Main Characteristics

FAST is a verification tool based on model-checking. It allows to check safety properties[1] on systems modelled as automata extended with integer variables. To succeed in computing all the reachable states of the systems, FAST uses both symbolic representation and acceleration techniques.

1.1 What is Acceleration?

Acceleration comes down to computing in one step the effect of iterating a control loop of the model. Thus acceleration is similar to *widening*. But acceleration computes the exact reachability set while usually widening only computes an upper-approximation.

Thanks to acceleration, FAST *quickly computes very large reachability sets*. The example shown in figure 1.1 is computed immediatly whereas tools without acceleration have to compute 10^9 iterations of transitions.

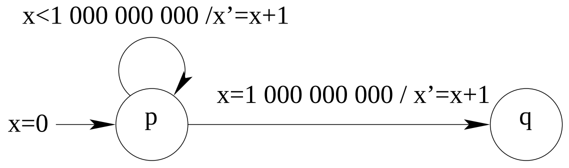


Figure 1.1: A model with a very large reachability set

Moreover FAST *allows to compute efficiently infinite reachability sets*. The computation of the exemple given in figure 1.2 terminates immediatly, although the reachability set is equal to the set of positive integers. In this case, usual methods (firing one by one all the firable transitions) do not terminate.

[1]"Something wrong will never happen."

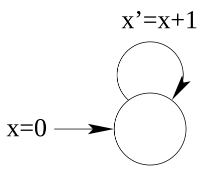


Figure 1.2: A model with an infinite reachability set

1.2 Key Points for FAST

FAST mainly provides:

- a *powerful model*,
- the *automatic computation of the reachability set* in most practical cases, even when the reachability set is very large or infini,
- a *user-friendly* Graphical User Interface (GUI).

Chapter 2

Installation and Global Settings

This chapter explains the few steps required to install the FAST tool. Note that it can be used in two different manners: *standalone*, where only FAST must be installed, and with its dedicated *GUI*.

2.1 Installing FAST

To install FAST, go to the directory where it should be installed (e.g. `/usr/local/bin`), and unpack the package:

```
tar xzvf fast.tgz
```

To start FAST, use:

`fast` *filename*

where *filename* describes the model to analyse and how to analyse it (see chapters 4 and 5).

2.2 Installing FAST GUI

To use the GUI, both a communication server and the GUI itself must be installed. This is due to the tool architecture (see figure 7.1, chapter 7), which allows the user to have a dedicated computation machine running FAST and another one for the GUI. The server must be installed on the machine which will run FAST.

2.2.1 Starting with the Server

To install the server, go to the directory where it should be installed (e.g. `/usr/local/bin`), and unpack the package:

```
tar xzvf serverFAST.tgz
```

The server handles communications between the GUI (which can be launched on a different machine), and the FAST computation engine. To start the server, use:

`serverFAST` *portnumber*

where *portnumber* is the number of the port used for connecting the GUI to the server. The default port is 35000.

2.2.2 GUI Installation

To install the GUI, go to the directory where it should be installed (e.g. `/usr/local/bin`), and unpack the package:

```
tar xzvf interFAST.tgz
```

To start the GUI, use:

```
java -jar InterFAST
```

2.2.3 Dependencies

In order for the GUI to work correctly, FAST, the server and a java virtual machine (version 1.4 or higher) must already be available.

2.2.4 GUI Configuration File

When first starting the GUI, a default configuration file is created in the user's home directory: `$HOME/.fastrc`. In this file are specified:

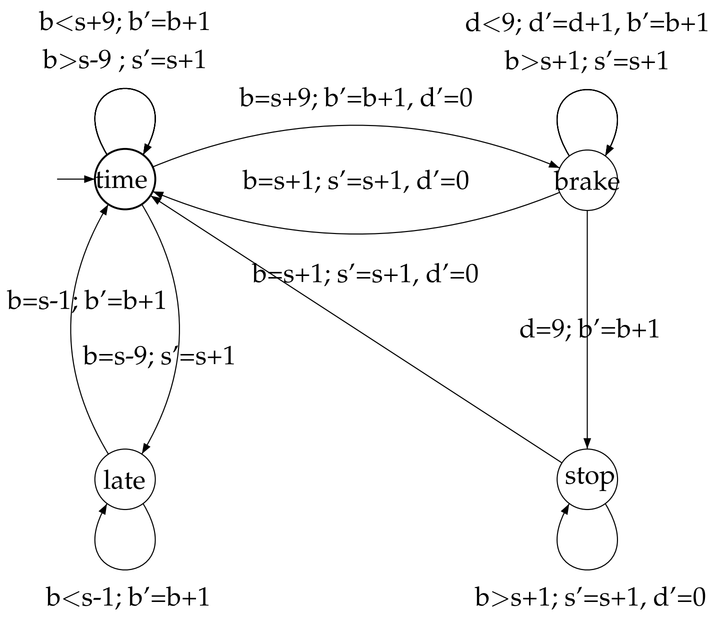
- the directory containing the *default strategies*: `dirStrategy` ;
- the *default model edition mode* `modeModelDefault`: 0 for the textual mode (see section 7.2), 1 for the assistant mode ;
- the *server address* `serverAddress`: the IP address either in numbers format or interpreted by the DNS ;
- the *port* on which the connection to the server is established, `serverPort` ;

- the *command line* used to start FAST on the server's machine, `serverFAST`;
- the *frequency of graphics updates*, `freqGraphics`;
- the *frequency of CPU time* updates, `freqCPUtime`;
- the display of FAST *textual output*, `printOutput` (`true` to display the output, or `false` to hide it) and `freqPrintOutput`.

Chapter 3

Running Example

In this chapter, we present a simple running example that will be used throughout the manual. It models a train speed control system. The train has to be on schedule, in order to avoid any accident with other trains. When the train is late, it speeds up and when the train is too fast, it brakes and stops. The speed control system receives two inputs: the time given by a global clock, and the number of beacons encountered on the line. To be on time, the train has to pass a beacon per second.



Initial configuration:
state=time $\wedge$ b=s $\wedge$ s=d $\wedge$ d=0

Property to check :
$|b - s| \leq 20$ always holds.

Figure 3.1: Train speed control system

In the model described in figure 3.1, b is the number of encountered beacons, s is the global clock, d is an auxiliary counter used during braking. The train can be in 4 different control states: `time`, `brake`, `late` and `stop`. The train is on time (state $time$) when $s - 9 < b < s + 9$. If $b = s - 9$, the train is late (state $late$). It speeds up until $b = s - 1$ and comes back to $time$. If $b = s + 9$, the train is early and has to brake (state $brake$) and then stops (state $stop$). When $b = s + 1$, the train is on time again ($time$).

A safety property of the system is that the train is never late (or early) more than 20 seconds. This can be expressed by the following formula:

$$(P1)\ |b - s| \leq 20$$

Chapter 4

Model Specification

In this chapter, we explain the input format of a FAST *model*. The full grammar for model specification can be found in annex A.

4.1 Model Description

FAST works on automata extended with unbounded counters. Concretly, the input file declares the counters and the control structure of the automaton (nodes and transitions). Counters range over *positive integers*[1]. A transition is defined by its name, a source node, a destination node, a guard and an action.

4.1.1 Guards

Guards are expressed as Presburger formulae. There is only one restriction: there cannot be quantification over variables of the system (counters). However quantification over variables local to the transition is possible.

The operators $<, >, =, <=, >=$, the constants $true, false$ as well as the logical connectives $\&\&(\wedge), ||(\vee), !(\neg), =>\ (\Rightarrow), <=>\ (\Leftrightarrow)$ are available. Existantial and universal quantification are expressed by `exists var.formula` and `forall var.formula`.

4.1.2 Actions

Actions are arbitrary affine functions over the counters [2]. Actions are specified by:

$$x_1' = \sum \lambda_{1j} x_j + c_1, \ldots, x_n' = \sum \lambda_{nj} x_j + c_n$$

where x_i' is the new value of counter x_i. If a primed variable is not written, the transition will not modify its counter value.

[1] This is not an important limitation: this way, computation is faster, and most case studies deal only with positive integers.

[2] But if the associated monoid is not finite, FAST will not terminate.

4.2 Example

First, the model is declared, with a name (`train`). Although for the moment FAST only deals with full models, future extensions will allow to use several communicating models.

Then, all the characteristics of the model are described: variables, states and transitions. A list of variables starts with the keyword `var`. Variable names are strings. The states are declared in a list, identified by the keyword `states`. The states names are also strings.

The declaration of transitions is a bit more complex. A transition is declared as a structure where all fields are mandatory and should be instantiated. After `transition`, the transition name (a string) is provided. This means that the transitions of figure 3.1 should be identified. In the example, we chose to give them numbers. The transition structure describes its characteristics. The `from` (respectively `to`) attribute allows to specify the source (resp. destination) state of the transition. The `guard` of the transition is a Presburger formula. The `action` is a list of affine functions applied to the variables. The new value of the variable, appearing on the left-hand side of =, is identified by the name of the variable followed by a prime. E.g. in `s'=s+1`, the new value of `s` will be `s+1`.

```
model train {
    var b, s, d;
    states time, late, brake, stop;

    transition t1 := {
        from   := time;
        to     := time;
        guard  := b>s-9;
        action := s'=s+1;
    };
    transition t2 := {
        from   := time;
        to     := brake;
        guard  := b=s+9;
        action := b'=b+1, d'=0;
    };
    transition t3 := {
        from   := brake;
        to     := time;
        guard  := b=s+1;
        action := s'=s+1, d'=0;
    };
    transition t4 := {
        from   := late;
        to     := time;
        guard  := b=s-1;
        action := b'=b+1;
    };
```

```
    transition t5 := {
        from   := time;
        to     := late;
        guard  := b=s-9;
        action := s'=s+1;
    };
    transition t6 := {
        from   := late;
        to     := late;
        guard  := b<s-1;
        action := b'=b+1;
    };
    transition t7 := {
        from   := brake;
        to     := stop;
        guard  := d=9;
        action := b'=b+1;
    };
    transition t8 := {
        from   := brake;
        to     := brake;
        guard  := b>s+1;
        action := s'=s+1;
    };
    transition t9 := {
        from   := brake;
        to     := brake;
        guard  := d<9;
        action := d'=d+1, b'=b+1;
    };
    transition t10 := {
        from   := stop;
        to     := stop;
        guard  := b>s+1;
        action := s'=s+1;
    };
    transition t11 := {
        from   := stop;
        to     := time;
        guard  := b=s+1;
        action := s'=s+1, d'=0;
    };
    transition t12 := {
        from   := time;
        to     := time;
        guard  := b<s+9;
        action := b'=b+1;
    };
}
```

Chapter 5

Elements of Strategies

In this chapter, we introduce the specification of *strategies for reachability analysis* with FAST.The full grammar for strategy specification can be found in annex B.

5.1 Specifying a Strategy

Strategies are written in a high-level language, which allows the user to express properties to check as well as to direct the computation. Strategies are easy to use and expressive: forward or backward analysis is straightforward, complex analysis (for example modular analysis) can also be specified.

5.1.1 Data Types

Data types used in strategies are *boolean*, *Transitions*, *Region*.

boolean usual boolean, with `True`, `False`, `!` ($\neg$), `or` ($\vee$), `and` ($\wedge$).

Transitions represents sets of transitions. An object of type Transitions is created with the explicit list of all the transitions it contains.

Region represents (symbolic) sets of configurations. An object of type $Region$ is created from a valid Presburger formula.Available operators are `&&` ($\cap$), `||` ($\cup$), `!` (^{c}). Available functions are:

$$subSet : Region \times Region \to boolean$$

$$eqSet : Region \times Region \to boolean$$

$$isEmpty : Region \to boolean$$

$$post* : Transitions \to Region(\to integer) \to Region$$

$$pre* : Transitions \to Region(\to integer) \to Region$$

For functions $post*$ and $pre*$, $(\rightarrow integer)$ means that the argument is optional. $post*$ $(trans, states, k)$ returns the $Region\ S$ which is the reachability set starting from $states$, using transitions in $trans$ and accelerating cycles of length $\geq k$. The semantics of $pre*$ is similar. Note that $pre*$ and $post*$ use semi-algorithms.

5.1.2 Control Structures

Writing a strategy is structured by the declaration of variables to store intermediate results:

```
Type name := value;
```

and conditional expressions:

`if` *condition* `then` ... `else` ... `endif`.

5.1.3 Predefined Procedures

$setMaxState : integer \rightarrow unit$ sets the maximal size of FAST internal representation for $Region$[1]. When this limit is reached, FAST tries to accelerate cycles of greater length. 0 means infinity (no limit), and is the default value.

$setMaxAcc : integer \rightarrow unit$ sets the maximal number of concatenated accelerations. When this limit is reached, FAST tries to accelerate cycles of greater length. 0 corresponds to infinity (no limit), and is the default value.

$print : string \rightarrow unit$ prints a string on the output device.

5.1.4 Strategies in practice

The strategies specification language is very rich and allows user to express complex analysis requests. However, *in most practical cases, a common strategy is sufficient to analyse the system.* Here are the two basic strategies for reachability analysis, and their translation in FAST.

Typically, a forward analysis consists in checking the following property:

$post^*(trans, Initial) \subseteq GoodProperty$

which, in FAST, comes down to check the value of the following $boolean$

```
subSet(post*(trans,Initial),GoodProperty)
```

Similarly a backward analysis is often expressed by

$pre^*(trans, BadProperty) \cap Initial = \emptyset$

and in FAST

[1] It is an automaton representation called LNDD. The size is the number of nodes.

```
isEmpty(pre*(trans,BadProperty) && Initial)
```

These two strategies are proposed as a standard in INTERFAST, the GUI for FAST (see chapter 7). The use of an elaborate strategy may lead to serious improvements in computation time.

5.1.5 Dealing with the two parameters $MaxAcc$ and $MaxStates$

It may seem complex to set correctly the parameters $MaxAcc$ and $MaxStates$ (see section 5.1.3). However they are not so important in the computation, and beginners can just ignore them while advanced users can use them to improve (a little) reachability set computation.

Here are a few rules which can help you to set them correctly:

- Parameters are here only to give criteria during the computation of $pre*$ or $post*$ for increasing the length of cycles FAST considers. If you are sure of the length, just use $pre*$ or $post*$ and do not set any limit (default behaviour).
- Experimentally, the number of concatenated accelerations is the most appropriate criterion to see if the reachability set computation will finish or not.
- When the computed LNDDs are too large, computation is slower. It can be interesting to restart with longer cycles (and smaller LNDDs). Note that the size of an LNDD is not necessarily monotonous with time, because there is no correlation between the size of an LNDD and the "number"of configurations contained in the `Region` this LNDD encodes.

5.2 Example

```
strategy train {
    print("Initialization");
    Region init := {state=time && b=s && s=d && d=0};

    print("Transitions");
    Transitions t := {t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12};

    print("Computation starts");
    Region good := {s-b<=20 && b-s<=20};

    Region reach := post*(init,t,2);

    if (subSet(reach,good))
        then print("OK : reached states are SAFE");
        else print("WARNING : BAD states are reached");
    endif
}
```

Chapter 6

FAST Output

Once launched (as a standalone program, i.e. without the GUI), FAST provides the user with textual information on the computation in progress. This chapter describes briefly these textual outputs which can be useful for the analysis of complex systems, when basic strategies do not succeed. We focus here on outputs which occur when invoking a pre^* or a $post^*$ computation.

6.1 FAST Computation Phases

FAST computation is divided in different phases:

1. computation of cycles of length $\leq$ k
   (a) computation of cycles of length $\leq$ k
   (b) simplification
   (c) cycle acceleration computation (or meta-transitions computation)
2. computation of the reachability set
   (a) try to iteratively reach the fixpoint
   (b) and then?
      - if a fix point is reached $\rightarrow$ stop the computation
      - if a stop criterion is violated $\rightarrow$ goto 1 with $k' := k + 1$

Each of these phases has its own messages.

6.2 Computation of Cycles of Length $\leq$ k

First, FAST indicates the maximum number of cycles of length $\leq$ k, and for each of the possible cycles, try to effectively build it by composing the elementary transitions of the cycle. If the transition cannot be composed, it is not a cycle and it is discarded.

```
(1/169)->
(2/169)->
(3/169)->
(4/169)->
(5/169)->
(6/169)->
(7/169)->
  ...
(169/169)->
```

When it is finished, FAST has computed all the cycles, but some of them can be simplified as explained in [FL02]. The next output indicates when cycles are simplified and the final number of cycles considered is written.

```
Simplification of the linear system
4 functions have the same action, only 1 remains
4 functions have the same action, only 1 remains
2 functions have the same action, only 1 remains
4 functions have the same action, only 1 remains
2 functions have the same action, only 1 remains
1 functions have the same action, only 1 remains
 ...
2 functions have the same action, only 1 remains
Number of actions before simplification : 51
Number of actions after simplification  : 23
```

Finally FAST computes the accelerations of these cycles. They are internally represented as LNDDs, and the sizes of these LNDDs are indicated. The larger the LNDD, the slower the computation. Names of functions are such that: `a.b` means composition of `a` and `b`, `a+b` is the simplification of functions `a` and `b` (see [FL02]).

```
cycles of length 2
(1/23), name = (+t3+t11+t3+t11), size = 10
(2/23), name = (+t1.t3+t1.t11+t3.t8+t11.t10), size = 14
(3/23), name = (+t2+t2), size = 13
(4/23), name = (+t3.t9+t11.t7+t12.t3+t12.t11), size = 20
 ...
(22/23), name = +t7.t9, size = 11
(23/23), name = (+Id+Id), size = 3
```

6.3 Computation of the Reachability Set

Then FAST tries to iteratively compute the fixpoint, using transitions and meta-transitions found earlier. Each time FAST tries a function σ, the name of σ is written, as well as the effect of the computation. There are three possible effects:

`No new state computed` No new reachable state was computed using σ.

I.N.C.R.E.A.S.E New states have been computed, but the size of the LNDD is larger. This computation is forgotten.

OK ! New states have been computed and the size of the LNDD is smaller (or remains the same). This computation is kept. The message:

`Now we have an automaton with p states(q bdd nodes)` indicates the size of the new LNDD. The message `Minimization in Progress` indicates that the computation is going on.

If no transition is found such that new states are computed and the size of the LNDD decreases, a random choice is made. The following message shows a random choice.

```
increase = 1
**********************
Random choice
```

Finally when the fixpoint is reached, the following message appears:

```
increase = 0
nb accelerations 13
Fix point FOUND
```

Here is part of the output for FAST reachability set computation, in the case of the train example:

```
**********************
Random choice
Name = (+t8.t9+t9.t8)  No new state computed
Name = (+t5+t5)  No new state computed
Name = (+t1.t4+t1.t12+t12.t1)  OK !
Now we have an automaton with 4 states(11 bdd nodes)
Minimization in Progress
Name = (+t5.t12+t6.t5) No new state computed
Name = (+t8.t9+t9.t8) No new state computed
Name = (+t3.t9+t11.t7+t12.t3+t12.t11) No new state computed
Name = +t9.t2 No new state computed
Name = (+t6+t6) No new state computed
Name = (+t2.t1+t8.t2) No new state computed
Name = (+t5+t5) No new state computed
Name = (+t1+t1)  I.N.C.R.E.A.S.E
 ...
increase = 1
**********************
Random choice
Name = (+t4+t12+t4+t12)  OK !
Now we have an automaton with 11 states(13 bdd nodes)
Minimization in Progress
Name = (+t5+t5) No new state computed
Name = (+t2+t2)  I.N.C.R.E.A.S.E
Name = (+t4.t6+t12.t4) No new state computed
```

```
 ...
Name = +t5.t1 No new state computed
Name = (+t1.t4+t1.t12+t12.t1) No new state computed
Name = +t9.t2 No new state computed
increase = 0
nb accelerations 13
Fix point FOUND
```

Chapter 7

INTERFAST, a GUI for FAST

INTERFAST is a Graphical User Interface (GUI) for FAST. Before using it, one should be aware of the tool architecture, described in figure 7.1. The interface itself can be started on a first machine, e.g. M1. Then, if the user wants the computation to be performed by a machine M2 (eventually different from M1, e.g. for memory consumption and performance issues), the server should be started on machine M2.

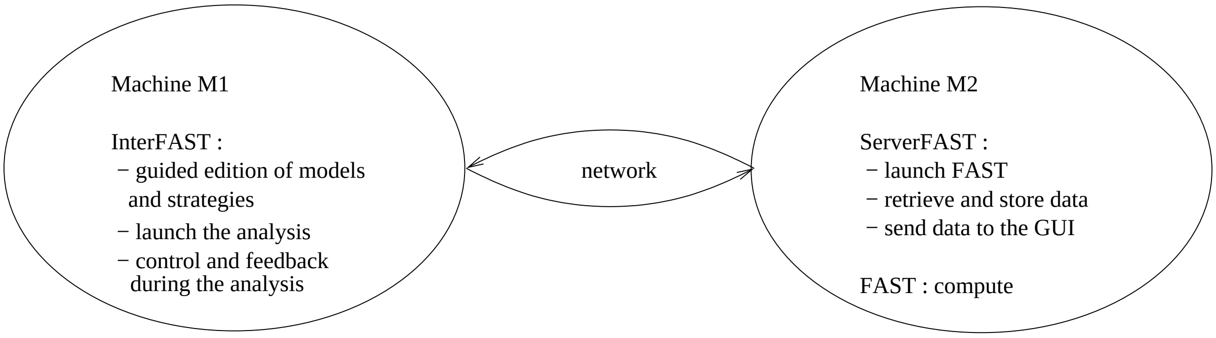


Figure 7.1: InterFAST architecture

When the user requests a computation from the interface, a connection is established between the GUI and a FAST process, via the server process. Note that a single server can handle simultaneously several connections with multiple FAST computation engines. Hence, several users can simultaneously connect to the same server, on the same port, at the same time.

7.1 Starting the GUI

To start the GUI, type:

```
java -jar InterFAST
```

When starting the interface for the first time, a message indicates that the configuration file cannot be found. Then a default configuration file, named `.fastrc`, is created in the user's home directory.

One of the two windows in figures 7.2 and 7.3 is then displayed:

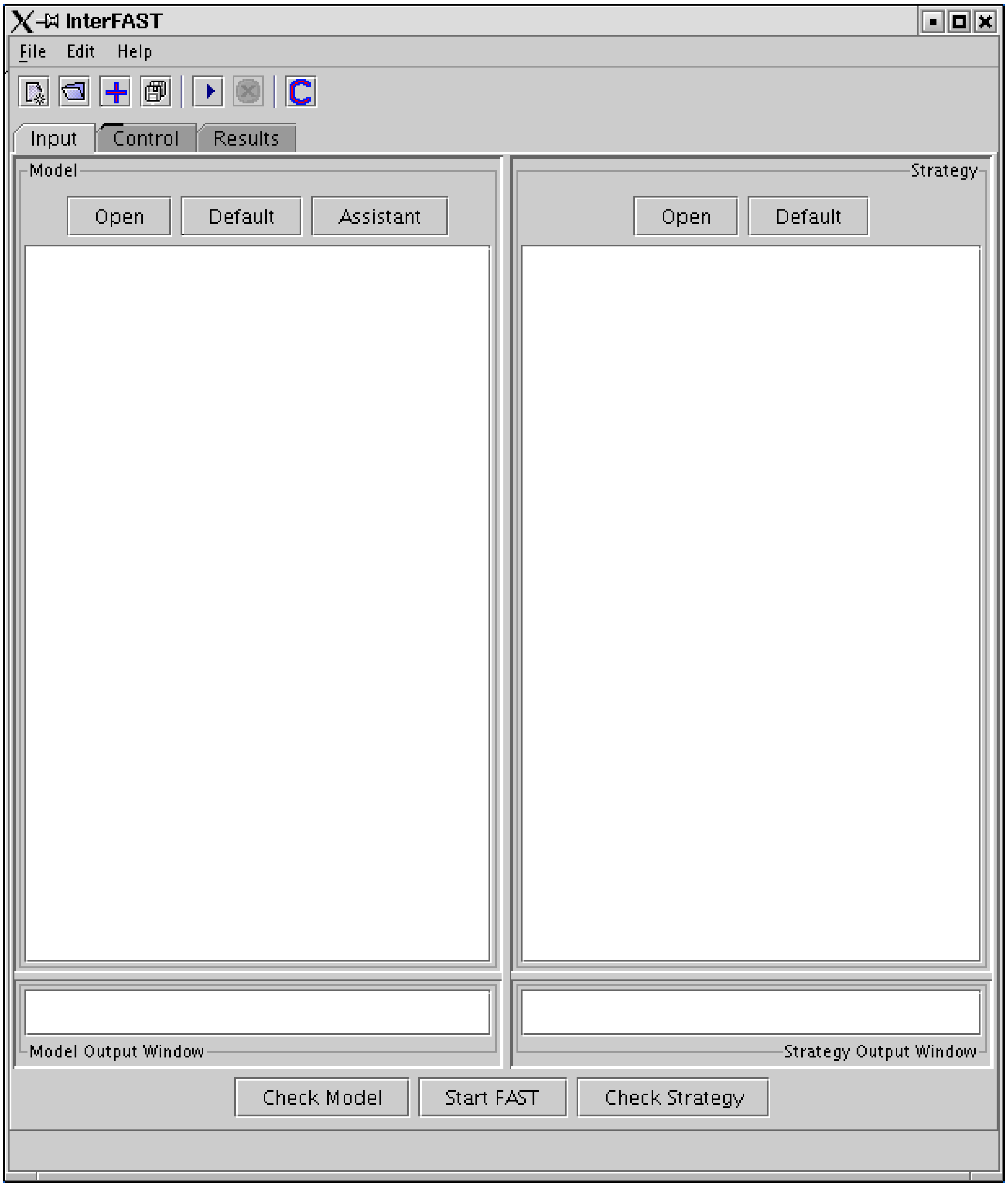


Figure 7.2: One of the starting windows

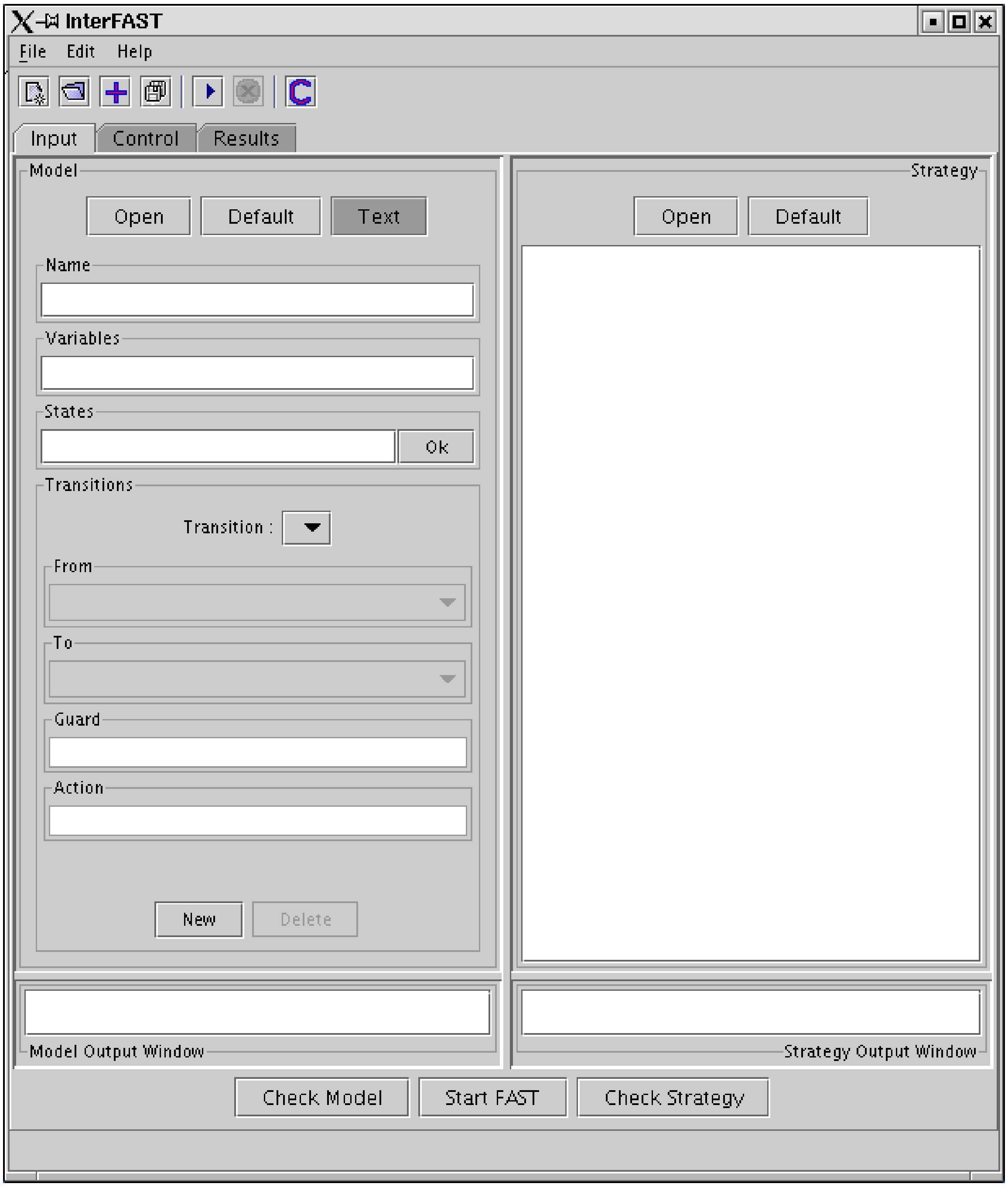


Figure 7.3: The other starting window

In both windows, the left-hand side is used for models while the right-hand side deals with strategies.

7.2 Editing a Model

There are two ways to edit a model. One is *purely textual* while the second features an *assisted mode* (with fill-in zones), meant for beginners. Figure 7.2 presents the inital window for textual input and figure 7.3, the one for the assisted mode.

We will now detail the functions performed by all the buttons on these pages. In the *textual mode*, the model is entered as text with the syntax described in chapter 4.

Assistant (textual mode only) toggles between the textual input mode and the assisted input mode. It is only permitted when the model syntax is valid. However, note that when doing this operation, all comments are lost.

Text (assisted mode only) toggles between the assisted input mode and the textual input mode.

Open allows the user to open an existing model.

Default is a shortcut to obtain a template of a model with n transitions, where n is a number of transitions supplied by the user through a dialog box.

Check Model performs a syntax check on the current model. The results of this syntax check are displayed in the bottom window (under the one where the model is entered). The model syntax is explained in chapter 4 and its grammar detailed in annex A.

The *assisted mode* is intended for users who are not yet familiar with FAST syntax. The corresponding window offers several fields: the model name, the list of variables it uses, the list of control states in the automaton, and a zone for transitions.

The *model name* is a string, starting with a character, composed only of strings and numbers. The *list of variables* is a comma-separated list of identificators. The same holds for the *list of states*. However, the list of states is taken into account only once the `OK` button next to it is pressed. If the list of states thus entered is syntactically correct, but some states previously used in transitions have disappeared, the user receives a warning. If he/she confirms the new state list, all those that have thus disappeared are replaced with `undefined`.
The zone describing *transitions* is more elaborate. An unfolding list allows the user to choose a transition from a list. The zone to handle transitions has 4 fields and 4 buttons:

From proposes a list to choose the *source state* of the transition ;

To proposes a list to choose the *destination state* ;

Guard is a text zone which represents the *guard* of the transition, with the same syntax as described in chapter 4 and annex A ;

Action is a text zone indicating the operations on variables, result of the transition occurrence ;

Delete button allows the user to delete an existing transition ;

New button allows the user to create a new transition.

7.3 Editing a Strategy

The syntax of strategies is detailed in chapter 5 and annex B.

Strategies are entered in the right-hand side window. The `Open` button opens an existing strategy.

When clicking the `Default` button, a default strategy is imported, giving access to all main strategies (`post*`, `pre*`, ...) available in FAST . These strategies should be completed by the user with a name, an initial state, the list of transitions that FAST should use for the computation.

Default strategies are located in the file *dirStrategy* specified in the configuration file (see sections 2.2.4 and 7.5). Other default strategies can be added by users, just storing the corresponding files in *dirStrategy*. These files do not have to be syntactically correct.

7.4 Starting Computations

To start a computation, the user must either click on the `start FAST` button or the corresponding icon in the icon bar (see section 7.5). If the syntax of the model and the strategy are both correct, and a connection to the *server* is possible, the interface is switched to its `Control` panel and the computation is started (see figure 7.4).

In the control panel, three graphics are displayed: they show (from left to right), the *memory usage*, the *number of accelerations* computed and the *number of states* in the automaton under construction, as functions of time.

Beneath the three graphics are two panels: `Information` and `Control`.

The `Information` panel displays instantaneous information from FAST: the current *cycle length*, *CPU time (processor time and user time)*, the *number of functions* used in the computation and FAST *status*. Basically, during a computation, FAST can be in three different states: *composing* basic functions in order to increase the cycle length, *computing accelerations* of the basic functions, *computing* the reachability set.

The `Control` panel provides two buttons: a *Pause/Resume* permitting to temporarily stop the computation and pursue later, and one to *stop* the computation.

Finally, the bottom window displays (when `print output` is ticked), FAST textual output.

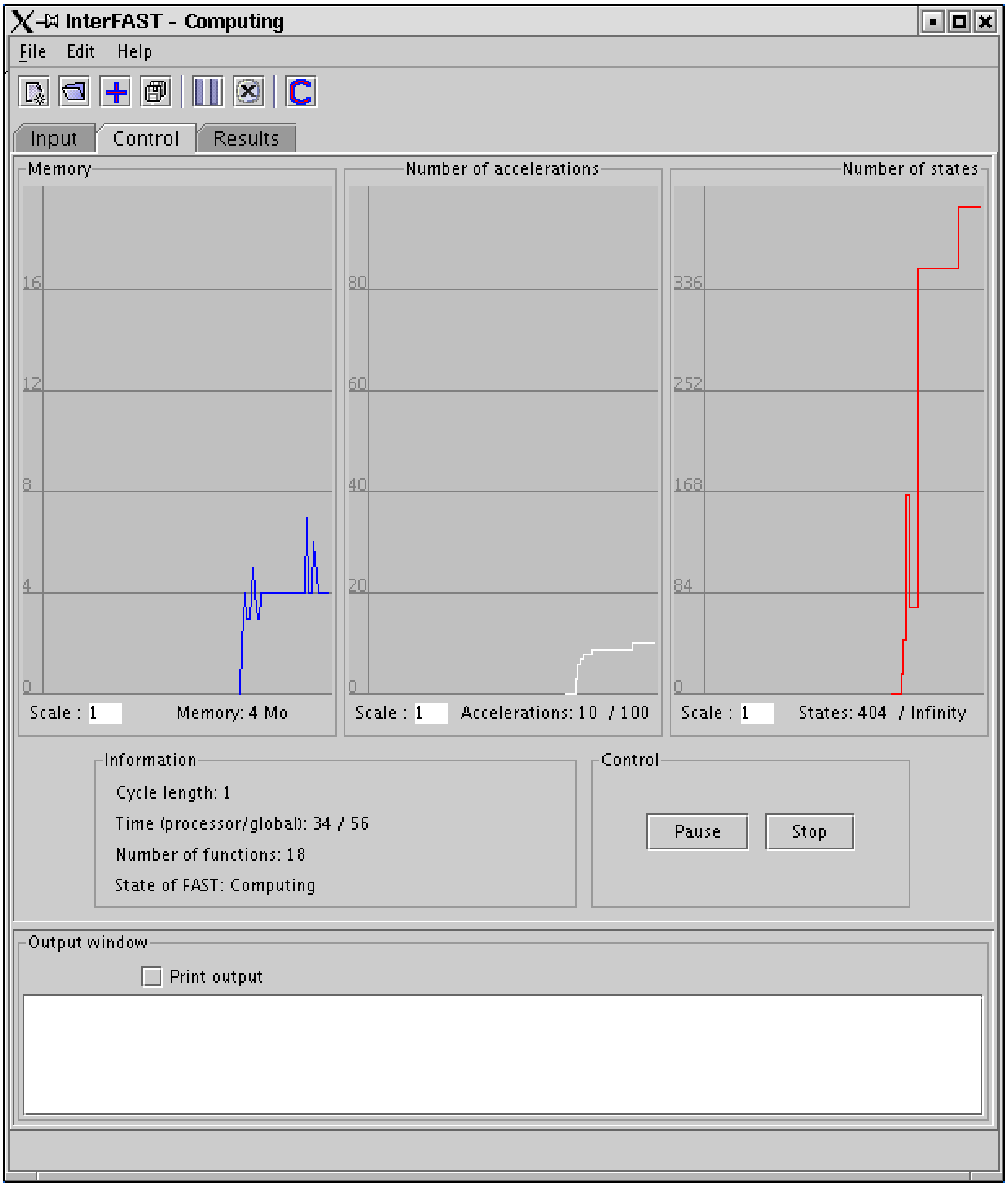


Figure 7.4: The Control panel

7.5 Icon Bar: Shortcuts

The icon bar on top of the window provides shortcuts for different functions. We explain these from left to right:

- *close* both models and strategies ;
- *open* an existing model ;
- *open* an existing strategy ;
- *save* the model and strategies ;
- *(re)start* a computation or *pause* if a computation is currently being operated ;
- *stop* a computation ;
- display the current *configuration*.

Bibliography

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002), Kanpur, India, Dec. 2002*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

Appendix A

Grammar of Model Specifications

```
MODEL == model IDENT {
        VARIABLES
        STATES
        TRANSITIONS
        }

IDENT == string

VARIABLES == var VARLIST ;

VARLIST == IDENT | IDENT , VARLIST

STATES == states STATELIST ;

STATELIST == STATE | STATE , STATELIST

STATE == IDENT

TRANSITIONS == TRANSITION | TRANSITION TRANSITIONS

TRANSITION == transition IDENT := {
        from := IDENT ;
        to :=  IDENT ;
        guard := NONPRIMEFORMULA ;
        action := AFFECTATIONLIST ;
        };

NONPRIMEFORMULA == NONPRIMEFORMULA && NONPRIMEFORMULA
        | NONPRIMEFORMULA || NONPRIMEFORMULA
        | NONPRIMEFORMULA <=> NONPRIMEFORMULA
        | NONPRIMEFORMULA => NONPRIMEFORMULA
        | exists IDENT . NONPRIMEFORMULA
        | forall IDENT . NONPRIMEFORMULA
        | ! NONPRIMEFORMULA
        | ( NONPRIMEFORMULA )
        | TERM
```

```
        | STATE = IDENT
        | true
        | false

TERM == NONPRIMELINEARSUM = NONPRIMELINEARSUM
        | NONPRIMELINEARSUM < NONPRIMELINEARSUM
        | NONPRIMELINEARSUM > NONPRIMELINEARSUM
        | NONPRIMELINEARSUM <= NONPRIMELINEARSUM
        | NONPRIMELINEARSUM >= NONPRIMELINEARSUM

AFFECTATIONLIST == AFFECTATION | AFFECTATION , AFFECTATIONLIST

AFFECTATION == IDENTPRIME = NONPRIMELINEARSUM

IDENTPRIME == IDENT'

NONPRIMELINEARSUM == NONPRIMELINEARSUM + NONPRIMELINEARSUM
        | NONPRIMELINEARSUM - NONPRIMELINEARSUM
        | NUMBER * NONPRIMELINEARSUM
        | - NONPRIMELINEARSUM
        | ( NONPRIMELINEARSUM )
        | NONPRIMEVALUE

NONPRIMEVALUE == IDENT | NUMBER

NUMBER == positive_integers

COMMENTS == /* COMMENT */ | // COMMENT

COMMENT == string
```

Appendix B

Grammar of Strategies Specifications

```
STRATEGY == strategy  IDENT { INSTRUCTIONLIST }

INSTRUCTIONLIST == INSTRUCTION | INSTRUCTION  INSTRUCTIONLIST

INSTRUCTION == DECLARATION ;
        | IFTHENELSE
        | PROCEDURE ;

DECLARATION == Region IDENT := REGION ;
        | Transitions IDENT := TRANSITIONS ;
        | boolean IDENT := BOOLEAN ;

REGION == REGION && REGION
        | REGION || REGION
        | ! REGION
        | ( REGION )
        | REGIONTERM
        | pre* ( REGION , TRANSITIONS )
        | pre* ( REGION , TRANSITIONS , NUMBER )
        | post* ( REGION , TRANSITIONS )
        | post* ( REGION , TRANSITIONS , NUMBER )

REGIONTERM ==  IDENT | { NONPRIMEFORMULA }

TRANSITIONS == IDENT | { TRANSITIONLIST }

TRANSITIONLIST == TRANSITION | TRANSITION , TRANSITIONLIST

TRANSITION == IDENT

IFTHENELSE == if BOOLEAN then INSTRUCTIONLIST  endif
        | if BOOLEAN then  INSTRUCTIONLIST  else  INSTRUCTIONLIST endif

BOOLEAN == ( BOOLEAN )
        | ! BOOLEAN
        | BOOLEAN = BOOLEAN
```

```
         | BOOLTERM
         | BOOLEAN and BOOLEAN
         | BOOLEAN or BOOLEAN

BOOLTERM == IDENT
         | True
         | False
         | eqSet ( REGION , REGION )
         | subSet ( REGION , REGION )
         | isEmpty ( REGION )

PROCEDURE == setMaxState ( NUMBER )
         | setMaxAcc ( NUMBER )
         | print ( STRING )
```