Technical Report

Automatic instantiation of Generic Statechart

Index

Index of figures

1 References

[1] M. Banci, *"SNCF Model Reduction – Identification of a subset of SNCF statecharts and rules"*, FMT-ISTI-CNR internal report, 2004

[2] M. Banci, *"STATEMATE XMI EXPORTER – Esportazione di modelli SNCF da Statemate a un formato XMI"*, FMT-ISTI-CNR internal report, 2005

[3] M. Banci, "inSTanCe Reference Manual 0.1", FMT-ISTI-CNR internal report, 2005

[4] SNCF-RFF, "SNCF Generic Model" ,SNCF-RFF internal report, 2004

[5] I-Logix, *"DATAPORT Library"*, Statemate, 2004

[6] M. Banci, "Manuale utente INSTANCE", FMT-ISTI-CNR manuale utente , 2005

[7] D. Harel,. Politi: Modeling reactive systems with statecharts: the Statemate approach, 1999.

[8] A. Torchi:
SMARTLOCK Rappresentazione di specifiche funzionali in forma di diagrammi di stato
Alstom Ferroviaria S.p.A, (2003)

[9] P. E. Debarbieri, F. Valdambrini, E. Antonelli (1987). A.C.E.I. Telecomandati per linee a semplice binario, schemi I0/19. CIFI Collana di testi per la preparazione agli esami di abilitazione, Quaderno 12, 1987.

[10] Free Software Foundation: GNU Autoconf, v. 2.58,
http://www.gnu.org/software/autoconf/autoconf.html

[11] XSLT Transformations Version 1.0, W3C Recommendation, 16/11/1999,
http://www.w3.org/TR/xslt

[12] Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation, 14/01/2003 http://www.w3.org/TR/SVG11/

[13] AT&T Research and Lucent Bell Labs, Graphviz, Graph Drawing Programs, v. 1.10,
http://www.research.att.com/sw/tools/graphviz/

[14] E. Gansner, E. Koutsofios, S. North: Drawing graphs with dot, 4/02/2002,
research.att.com, dist/drawdag/dotguide.ps.Z

[15] Pavel Hlavnicka: Sablotron XSLT processor, v. 1.0, http://www.gingerall.org

[16] Adobe Systems Incorporated: Adobe Scalable Vector Graphics plugin, v. 3.01 beta 3, http://www.adobe.com/svg/main.html

2 Introduction

This report describes the results of the collaboration between ISTI-CNR and ALSTOM performed from November 1st, 2004 to April 30th, 2005, finalized to study the realization of a tool able to automatically instantiate generic statecharts supplied by SNCF. The tool follows the idea of a lifecycle starting from a Statemate generic model and producing an instantiated Statemate model.

The results of the collaboration are reported starting form section 3, in which a reduced SNCF model specification of an interlocking system for a sample station is shown. This model has been developed starting from the whole SNCF specification, and has been reduced focusing on the main interesting features of the model. This reduced set of generic statecharts is still able to be simulated and maintains a proper semantic meaning. The generic reduced model has also been introduced in the Statemate tool.

Section 4 shows the filtering tool (XMI Exporter) which is needed to export information by the Statemate tool, where the SNCF generic model has been defined.

Section 5 shows the instantiator tool, which is the core of the development activity.

Section 6 shows the several possibilities of representation and post-processing of the instantiated statecharts.

The final chapter reports a list of open topics that need to be further analysed and studied in a future collaboration between ALSTOM and ISTI-CNR.

3 The reduced SNCF model

The development of the instantiator required a well defined subset of statecharts extracted from the global model supplied by SNCF. Particularly, this work has been required in order to have some generic charts which could be used to generate a consistent and semantically correct instantiated model.

This section of the report describes the reduction applied to the SNCF specification, in order to identify a closed set of statecharts able to describe a complete subset of interlocking functionalities.

The work was finalized performing some assumptions. These assumptions have allowed to minimize the variables used into a statechart, but other statecharts are then required in order to elaborate their values. In other words, when a statechart includes a variable there is the necessity to calculate its value, and to perform this another statechart has to be evaluated: this of feature has been named “nesting”.

An exhaustive description of the methodology adopted to reduce the model and which statecharts have been selected is included in reference 1.

3.1 Results of the reduction activity

After the reduction a system which is able to perform route locking and route destruction functions has been obtained.

In the Figure 1 it is shown the final nested statecharts. This minimal set of statecharts has been obtained recursively tracing the variables that, starting from the statecharts GRAPHE_ETAT_IT, have brought to the expansion of another statechart, and so on.

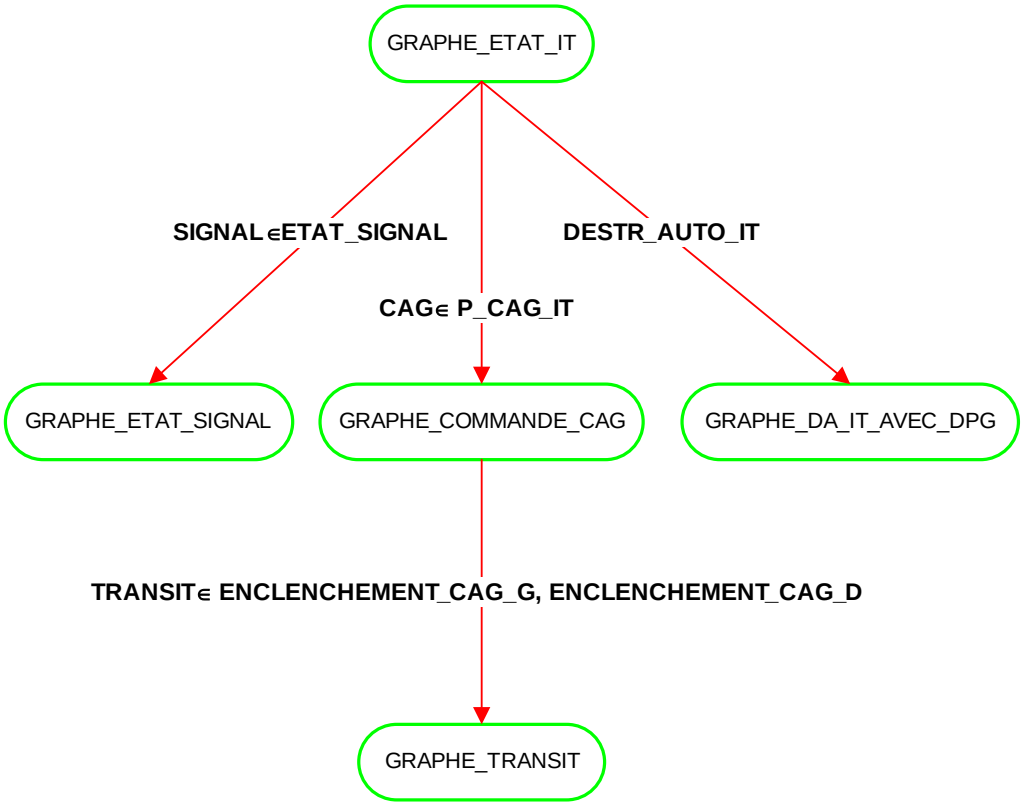


Figure 1 – Structure of graph nesting leading the graphs exploration

The Generic Model of interlocking system requirements supplied from SNCF are defined in an estensive document [1]: the system has been modelled using a high level language named Statecharts and, in particular, the Statemate Statecharts seems to be the preferred one. I-Logix Statemate is a comprehensive graphical modelling and simulation tool for the development of complex embedded systems.

Statemate Statecharts allow to model reactive systems. The Statemate language includes a lot of different statements and allows the construction of very complex applications.

However, a railway interlocking system is a simpler sort of application, which requires only a small part of the language power to be used. In fact, the generic model supplied from SNCF uses only a very restricted part of the language.

The final reduced model has been developed in the Statemate environment, and has been used as a database. The main feature of Statemate that has been used is the following one:

- access to the Statemate model internal information using a particular API (DataPort);

The other features of Statemate have not been used at this time, because the model is considered to be validated by the customer.

The reduction work has been done starting from one statechart named GRAPHE_ETAT_IT which describes the route setting (it is the first graph into the chain of route locking function).

3.2 Description of generic model structure

The first analyzed graph is GRAPHE_ETAT_IT, it is the first graph studied, because it is the starting point to the route setting; following the terms of this graph the model has been identified.
In Figure 2 the GRAPHE_ETAT_IT is shown. For more information about description of the graphs, refer to the documents *"SNCF Model Reduction – Identification of a subset of SNCF statecharts and rules"* [1] and *"SNCF generic model"* [4]. The first document describes the reduction activity to generate an executable model, the second describes the whole generic model.

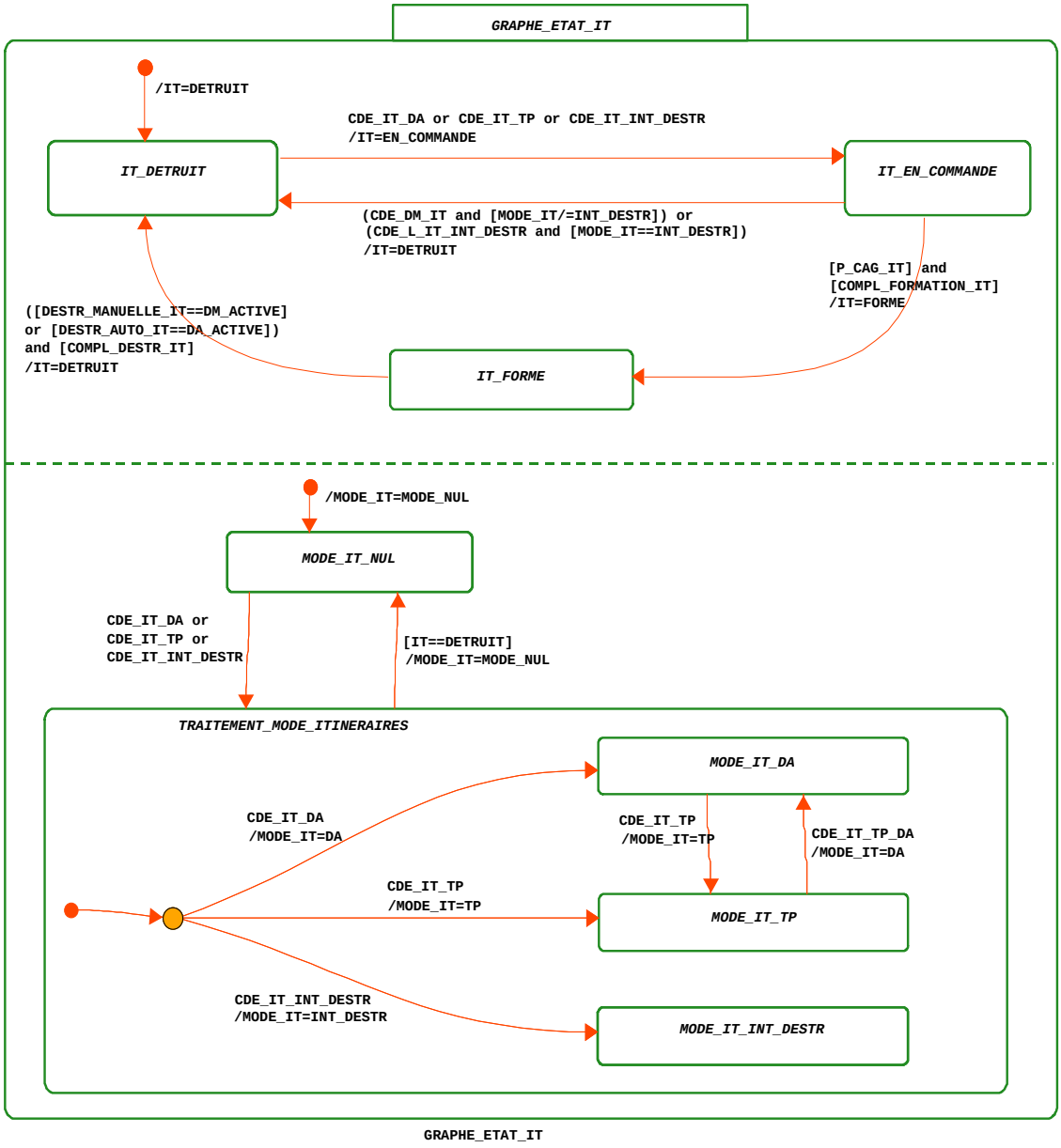


Figure 2 – Graphe_etat_it

3.2.1 Brief graph behaviour description

The graph is split into two “AND” subgraphs, which are independent each other. The graph below allows to choose the kind of route that is allowed: DA=Automatic Destruction, TP=Permanent route (tracé permanent), INT_DESTR=route with destruction not allowed (itinéraires formés avec interdiction de destruction).

Consequently to the reduction the behaviour of the system, and obviously of this graph, has been reduced to only one mode of route setting, the DA mode (automatic destruction).

The reduced SNCF model

For this reason the graph is not reduced modifying its terms, but simply not considering the external commands CMD_IT_TP and CMD_IT_INT_DESTR. In other words, only the command CMD_IT_DA is considered.

Consequently, it is evident that some parts of the graph might not be used.

This methodology of reducing the graph has been used for all the other graphs as well. Consequently, many functionalities of the model are not considered, and for this reason only some parts of it will be stimulated.

Figure 3 shows one AND-part of the first graph.

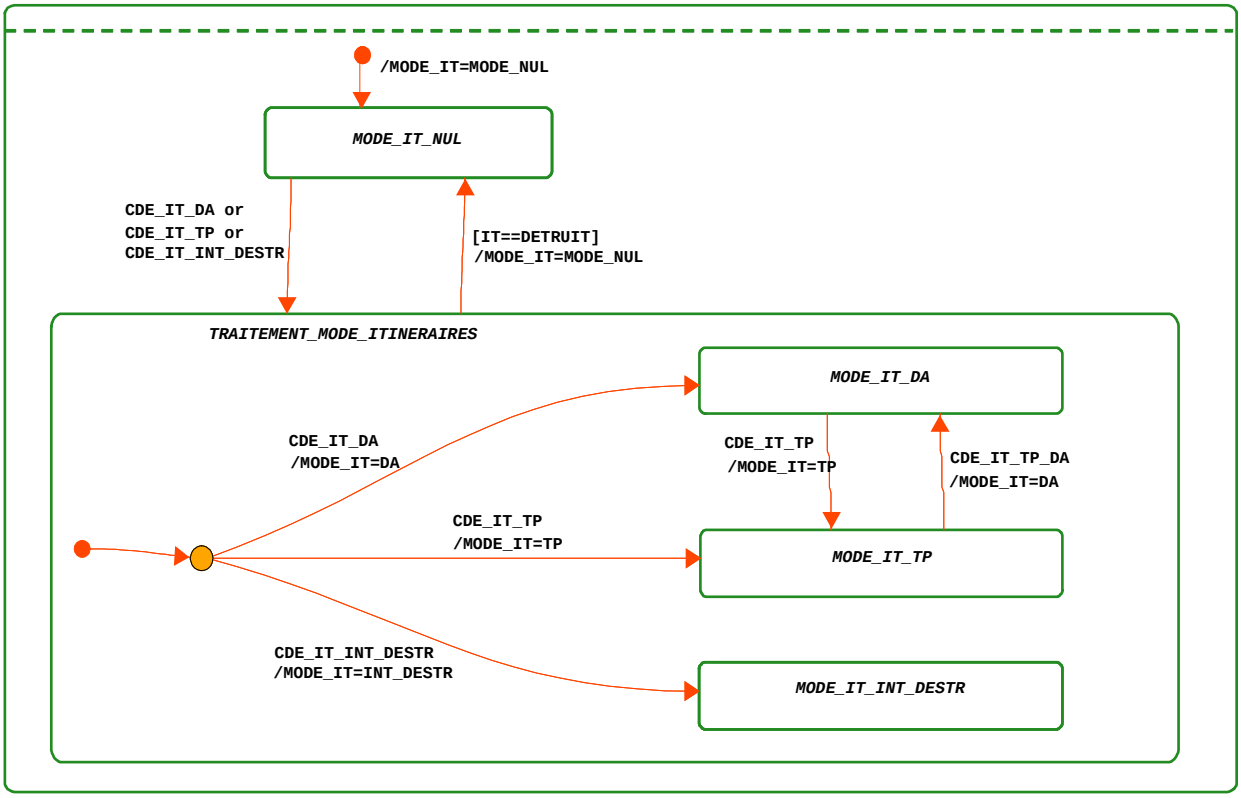


Figure 3 – choice of the route type

This part of the graph contains the following elements:

Command inputs	
CMD_IT_DA	used

The reduced SNCF model

CMD_IT_TP	not used
CMD_IT_INT_DESTR	not used
Variables	
IT	used

And referring to the second AND-part of the graph:

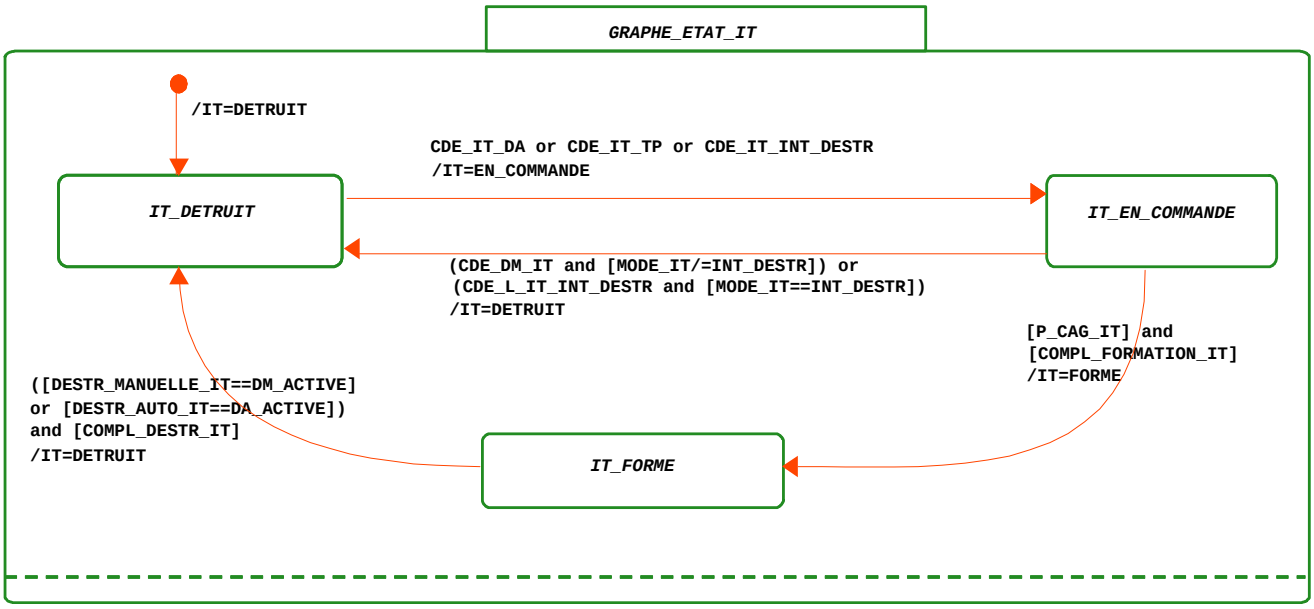


Figure 4 – route formation and destruction

Figure 4 shows the main part of the graph:

Receiving a route setting command (DA type only) the graph goes to IT_EN_COMMANDE. To go from this state to IT_FORME it is required to evaluate two terms that have to be expanded according to the Expansion Rules described below.

3.2.2 Expansion Rules

3.2.2.1 COMPL_DESTR_IT

Term always considered “true”.

These assumptions are useful to reduce the functionalities of the system and obviously reduce the relationships between different statecharts.

3.2.2.2 P_CAG_IT

It represents a Boolean equation that is true if all the devices used by this route are in the correct position.

The rule has been reduced eliminating some terms.

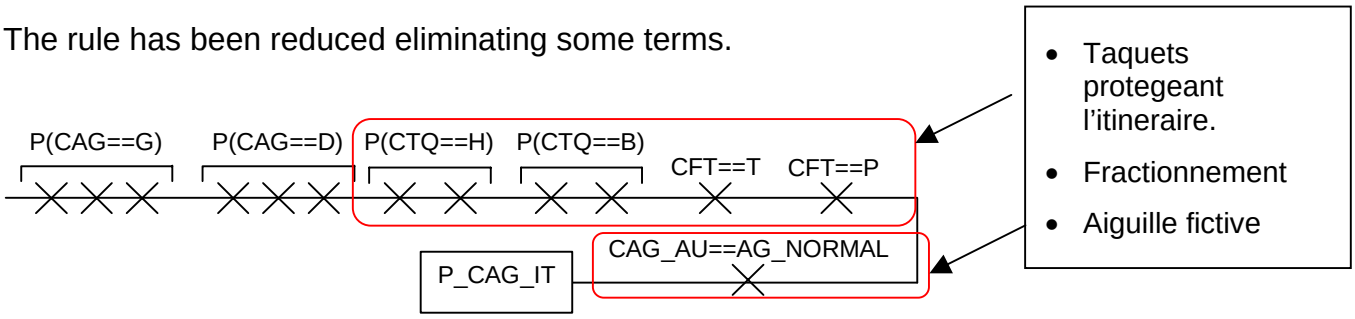


Figure 5 – Original rule

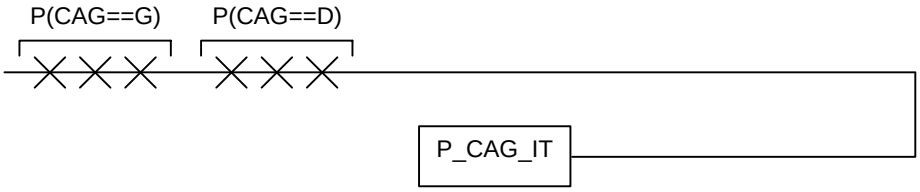


Figure 6 – Reduced rule

In other words, only the variables related to switch points have been considered; the other devices are managed in the same way, so they have been eliminated without reducing the validity of the model.

P(CAG==G): it is the logical product (AND) of switch points in normal position (Left)

P(CAG==D): it is the logical product (AND) of switch points in reverse position (Right)

These variables are related to other graphs (first level of nesting).

3.2.2.3 COMPL_FORMATION_IT (Complementary conditions for route setting)

It represents a Boolean equation that is true if all the signals over this route have a correct aspect.

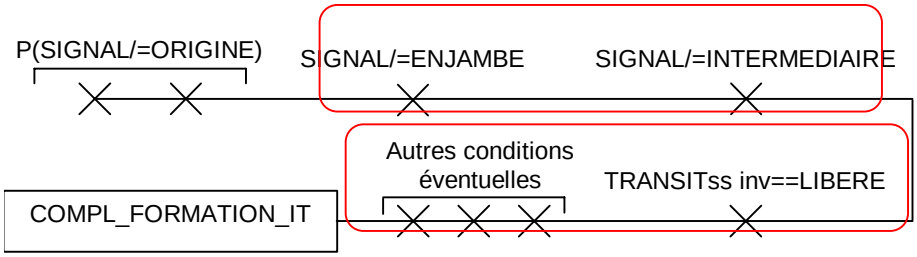


Figure 7 – original rule

This rule has been reduced deleting some variables because they seem to be not clearly explained and also because they do not supply any interesting features to the expansion methodology.

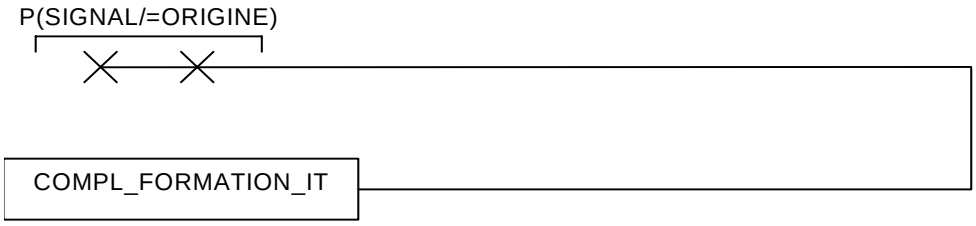
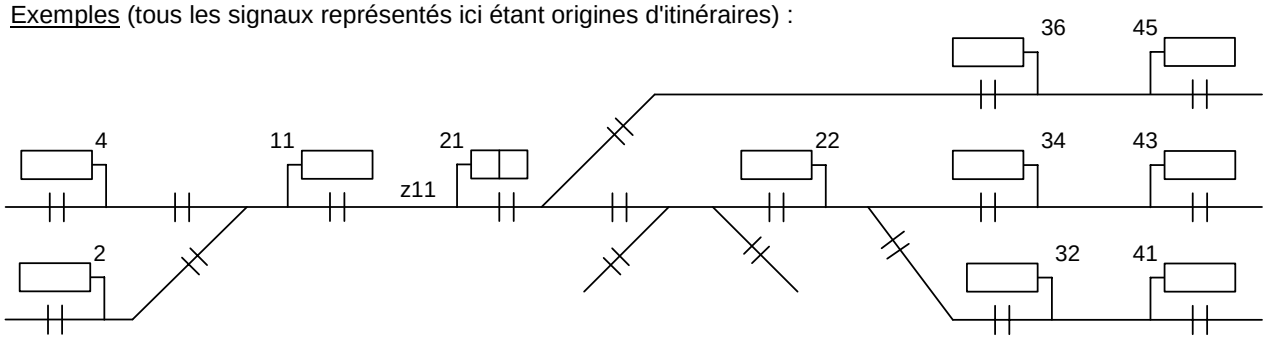


Figure 8 – reduced rule

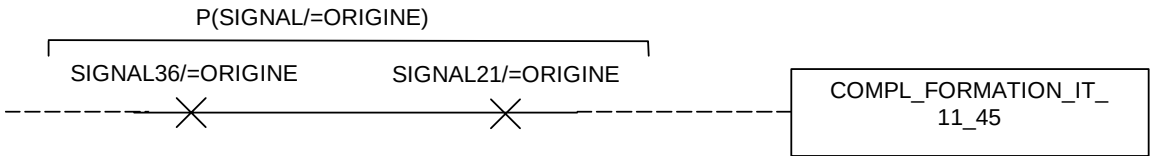
The SIGNAL variables are related to a graph named GRAPHE_ETAT_SIGNAL (first level of nesting).

Examples:

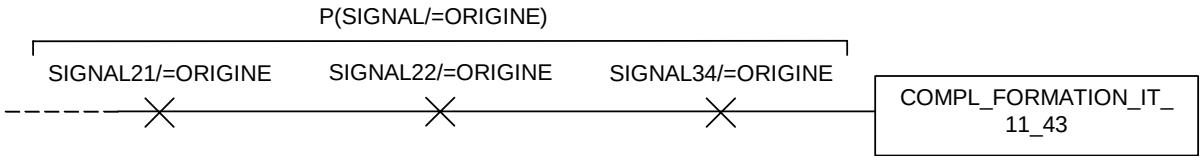
Exemples (tous les signaux représentés ici étant origines d'itinéraires) :



Condition complémentaire de formation de l'itinéraire 11-45 :



Condition complémentaire de formation de l'itinéraire 11-43 :



Condition complémentaire de formation de l'itinéraire 36-4 :

In order to set the route 11_45 the state of the signals 36 and 21 has to be evaluated.

The reduced SNCF model

If the state of one of these signals is ORIGINE it means that there is another already route set.

Output to the environment:

IT

MODE_IT

Variables stimulating statecharts nesting:

Variable name	***Term name***	***Needed Statechart***
CAG	P_CAG_IT	COMMANDE_AG
SIGNAL	COMPL_FORMATION_IT	ETAT_SIGNAL
DESTR_AUTO_IT		DA_IT_AVEC_DPG

3.3 Description of nesting and iteration

The previously explained methodology has been applied to five generic statecharts, in order to reduce their functionalities. The iteration has been driven by variables contained into conditions of the first statechart, requiring another statechart in order to be evaluated. Following this trace the new statechart has been analyzed and so on. The nesting activity is terminated when there are no more variables needed to be evaluated using other statecharts.

4 The XMI exporter

This is the filter tool that was developed to permit the use of generic model. In fact the generic model has been designed using the Statemate tool. This tool exports the model in a XMI-like language (XML Metadata Interchange), and this XMI model can then be used for instantiation.

4.1 Dataport

The Statemate Dataport Library contains functions that allow to extract information from the Statemate database using an external program.

- It allows to communicate with the Statemate tool
- It allows to get any information from the internal database of Statemate tool
- Dataport functions can be used within external programs

More specifically, the Dataport library includes:

- Functions to perform a wide variety of database extraction operations
- Using the library's function, the information pertaining to an element can be extracted from the specification database

Using these informations a model can be translated to an XMI specification.

4.2 Projects database access

The Statemate Magnum tool allows the user to model new projects using a well defined formalism. Projects are saved using an internal format, more exactly there is a structure dedicated to it (Data Dictionary). Statemate Magnum allows to export most of the information of projects: with the Statemate tool there is also a library (DataPort) (with C code interface) which allows data export. The functions to call to access the data dictionary are described in the library documentation.

These functions are classified in three categories:

- ***Single Element Functions***: functions acting on a single element of database at a time;
- ***Query Functions***: functions extracting lists of elements of a specified type;
- ***Utility Functions***: functions that elaborate elements of lists.

By using this kind of functions it has been possible to implement the tool to extract the project from Statemate, converting it in XMI format.

4.3 XMI Exporter tool implementation

Initially the XMI standard has been used (for UML, proposed by OMG) in order to export the Harel Diagram from Statemate, but Harel Statecharts are not the same than UML Statecharts. For instance the connectors are not implementable in UML. Furthermore, the event management is associated, in UML, to the model and, in Statemate, to Statecharts. In other words, Harel Statecharts and UML Statecharts are not equivalent.

Consequently, using the XMI specifications an XML model has been created in order to describe Statemate statecharts.

In other words, the statechars exportation to XMI requires a new DTD respecting the Harel rules.

4.3.1 Harel DTD

The structure of the defined DTD is similar to the one used by OMG (XMI 1.2), using only the needed elements for the analysed model. Some elements have also been renamed, and the structure has been changed to adapt it to Harel statecharts. In the following table an example of the ***state*** element is shown.

***State* (by OMG)**

`<!ELEMENT UML:State (UML:ModelElement.name | UML:ModelElement.visibility | XMI.extension |`
 	
`...`
 	
`* >`
 	
`<!ATTLIST UML:State`
`name CDATA #IMPLIED`

The XMI exporter

```
visibility (public | protected | private) #IMPLIED

...

%XMI.element.att;
%XMI.link.att;
>
```

***State* (by us) (Harel.dtd)**

```
<!ELEMENT HAREL:State (XMI.extension |      HAREL:StateVertex.outgoing |
 HAREL:StateVertex.incoming | HAREL:StateVertex.parent |
 HAREL:State.stateChart | HAREL:State.Reaction)*
>

<!ATTLIST HAREL:State
 name CDATA #IMPLIED
 %XMI.element.att;
 %XMI.link.att;
>
```

For more information related to description of XML Metamodel Interchange (XMI), MOF and DTD, refer to the documents *"STATEMATE XMI EXPORTER – Esportazione di modelli SNCF da Statemate a un formato XMI"* [2]. More specifically, the document details the XMI exporter tool and the following concepts:

- XML METAMODEL INTERCHANGE (XMI)
  - XML ed XMI
  - Meta Object Facility (MOF)
  - XMI e MOF
  - DTD structure of an XMI document
    - Metaclasses representation
    - Initial declarations
    - Needed attributes
    - Common elements to DTDs
  - Medamodel specification
  - XMI and UML
    - DTD structure of the UML

4.3.2 Tool architecture

The tool includes two separated elements: data acquisition and XMI code generation. Both these phases are independent each other; there is only one contact point between them (shared data structure).

The XMI exporter

During the first phase the data structure is populated using Statemate extracted data, during the second one this structure is used to create the related XMI code.

In bigger detail, the two phases include the following activities:

1. data acquisition from Statemate database using the Dataport library and population of dedicated C data structures;
2. data structures reading and generation of XMI output.

4.3.3 Data structures

Data structures are required because to create the XMI code it is preferred to use structured data. The xmi_exporter.h file contains definitions of data structures related to statecharts and activity charts.

The most used structures are:

- PROJECT;
- STATECHART;
- STATE;
- TRANSITION;
- EVENT;
- ACTIVITYCHART;
- ACTIVITY;
- FLOWLINE.

The definitions of some of them is shown below:

```
Typedef structs_project {
char*name;
XMID xmi_id;
PTR_LIST*statechart_list;
} PROJECT;
```

name: project name

xmi_id: XMI project id

statechart_list: list of statecharts of the project

```
typedefstructs_chart{
char*name;
stm_idchart_id;
XMIDxmi_id;
XMIDxmi_top_id;
PTR_LIST*top_state_list;
PTR_LIST*transition_list;
PTR_LIST*event_list;
} STATECHART;
```

name: statechart name

chart_id: statechart id (defined by Statemate)

xmi_id: statechart XMI id

xmi_top_id: parent state XMI id

list: state list belonging to the statechart

transition_list: transition list belonging to the statechart

event_list: event list belonging to the statechart

```
typedef structs_state{
char*name;
stm_idstate_id;
XMIDxmi_id;
XMIDparent_id;
PARENT_TYPEparent_type;

STATE_TYPEtype;
boolean pseudo;
STATECHART*instance;

char*reactions;

PTR_LIST*incoming_transition_list;
PTR_LIST*outgoing_transition_list;

PTR_LIST*child_list;
}STATE;
```

name: state name

state_id: state id (defined by Statemate)

xmi_id: state XMI id

parent_id: parent XMI id

parent_type: parent type (StateChart or CompositeState)

type: state type (AND, OR, BASIC, JUNCTION....)

pseudo: if it is a connector than it is true (pseudo state)

instance: reference to the generic StateChart

reactions: expression containing triggers, guard-conditions and actions

incoming_transition_list: transition list incoming to this state

outgoing_transition_list: transition list outgoing from this state

child_list: child state list (for CompositState only)

```
typedefstructs_transition{
stm_idtransition_id;
XMIDxmi_id;
XMIDchart_id;
char*expression;
structs_state*source;
structs_state*target;

} TRANSITION;
```

transition_id: transition id (defined by Statemate)

xmi_id: transition XMI id

chart_id: chart XMI id

expression: expression containing triggers, guard-conditions e actions

source: transition source

target: transition target

4.3.3.1 Statemate database access

Before accessing to the Statemate database it is required to initialize the transaction using the *stm_init_uadche* function, which includes 2 input parameters (workarea, project name).
After the initialization it is possible to use query functions. The functions that have

been used are:

VisitaStateCharts: it looks for all statecharts belonging to workarea, saving them into the PROJECT structure;

VisitaStates: it looks for states and connectors, and save them into proper statecharts respecting the nesting;

VisitaTransitions: it looks for all transitions and save them into proper statecharts;

VisitaEvents: it looks for all events and save them into proper statecharts.

Referring to activity charts, the most used functions are the following ones:

VisitaActivityCharts: it looks for all activity charts belonging to workarea saving them into the PROJECT structure;

VisitaActivities: it looks for activities (top-activity, external activities, data-store and connectors), and save them into a correct nested structure;

VisitaFlowlines: it looks for all flow-lines and save them into proper activity;

VisitaEvents: it looks for all events and save them into proper structure.

4.3.3.2 XMI file creation

The main function that permit to generate an XMI code starting from a PROJECT is CreaXMI. This function reads data previously extracted from Statemate using other functions that are dedicated to the creation of XMI elements. As a result this function outputs the XMI code.

4.4 Output example

A brief example of output generated by the XMI exporter tool is shown below.

The example shows a very simple model created by using Statemate and the corresponding output generated by XMI Exporter tool:

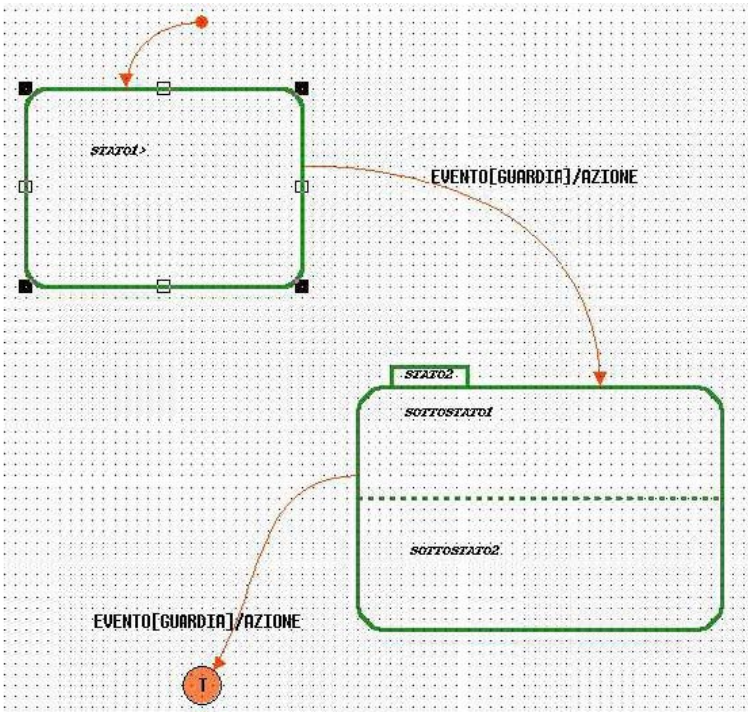


Figure 9 – An example of Statechart

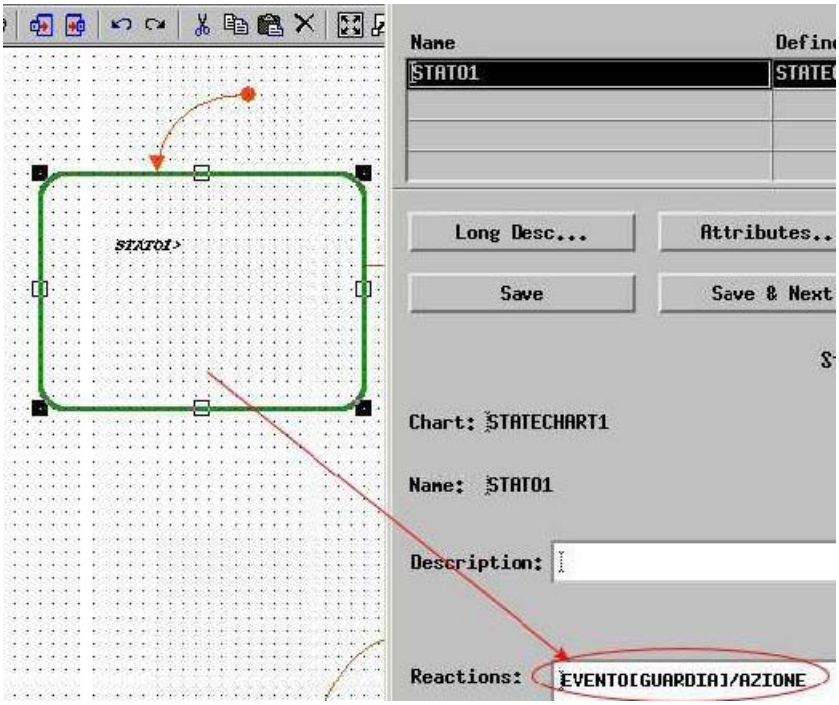


Figure 10 – Reactions “STATO1”

The XMI exporter

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE XMI PUBLIC "Harel" "./harel.dtd">
<XMI xmi.version="1.2">
       <XMI.header>
      <XMI.documentation>
         <XMI.exporter>xmi_exporter</XMI.exporter>
         <XMI.exporterVersion>1.0</XMI.exporterVersion>
      </XMI.documentation>
      <XMI.metamodel xmi.name="Harel.dtd" xmi.version="1.3"/>
   </XMI.header>
   <XMI.content>
      <HAREL xmlns:HAREL='./Harel.dtd'>
         <HAREL:Project name="PROGETTO" xmi.id="xmi.2">
            <HAREL:ownedElement>
               <HAREL:StateChart name="STATECHART1" xmi.id="xmi.3">
                  <HAREL:StateChart.context>
                     <HAREL:Link xmi.idref="xmi.2"/>
                  </HAREL:StateChart.context>
                  <HAREL:StateChart.States>
                     <HAREL:State name="STATO1"  xmi.id="xmi.5">
                        <HAREL:State.stateChart>
                           <HAREL:StateChart xmi.idref="xmi.3"/>
                        </HAREL:State.stateChart>
                        <HAREL:State.Reaction>
                           <HAREL:Expression>EVENTO[GUARDIA]/AZIONE</HAREL:Expression>
                        </HAREL:State.Reaction>
                        <HAREL:StateVertex.outgoing>
                           <HAREL:Transition xmi.idref="xmi.11"/>
                        </HAREL:StateVertex.outgoing>
                        <HAREL:StateVertex.incoming>
                           <HAREL:Transition xmi.idref="xmi.12"/>
                        </HAREL:StateVertex.incoming>
                     </HAREL:State>
                     <HAREL:CompositeState name="STATO2" isConcurrent="true" xmi.id="xmi.6">
                        <HAREL:State.stateChart>
                           <HAREL:StateChart xmi.idref="xmi.3"/>
                        </HAREL:State.stateChart>
                        <HAREL:State.Reaction>
                           <HAREL:Expression></HAREL:Expression>
                        </HAREL:State.Reaction>
                        <HAREL:StateVertex.outgoing>
                           <HAREL:Transition xmi.idref="xmi.13"/>
                        </HAREL:StateVertex.outgoing>
                        <HAREL:StateVertex.incoming>
                           <HAREL:Transition xmi.idref="xmi.11"/>
                        </HAREL:StateVertex.incoming>
                        <HAREL:CompositeState.SubVertex>
                           <HAREL:State name="SOTTOSTATO1" xmi.id="xmi.7">
                              <HAREL:StateVertex.parent>
                                 <HAREL:CompositeState xmi.idref="xmi.6"/>
                              </HAREL:StateVertex.parent>
                              <HAREL:State.Reaction>
                                 <HAREL:Expression></HAREL:Expression>
                              </HAREL:State.Reaction>
                           </HAREL:State>
                           <HAREL:State name="SOTTOSTATO2" xmi.id="xmi.8">
                              <HAREL:StateVertex.parent>
                                 <HAREL:CompositeState xmi.idref="xmi.6"/>
                              </HAREL:StateVertex.parent>
                              <HAREL:State.Reaction>
                                 <HAREL:Expression></HAREL:Expression>
                              </HAREL:State.Reaction>
                           </HAREL:State>
                        </HAREL:CompositeState.SubVertex>
                     </HAREL:CompositeState>
                     <HAREL:Connector value="" xmi.id="xmi.9">
                        <HAREL:State.stateChart>
                           <HAREL:StateChart xmi.idref="xmi.3"/>
                        </HAREL:State.stateChart>
                        <HAREL:Connector.kind xmi.value="default" />
```

```
                        <HAREL:StateVertex.outgoing>
                           <HAREL:Transition xmi.idref="xmi.12"/>
                        </HAREL:StateVertex.outgoing>
                     </HAREL:Connector>
                     <HAREL:Connector value="" xmi.id="xmi.10">
                        <HAREL:State.stateChart>
                           <HAREL:StateChart xmi.idref="xmi.3"/>
                        </HAREL:State.stateChart>
                        <HAREL:Connector.kind xmi.value="termination" />
                           <HAREL:StateVertex.incoming>
                              <HAREL:Transition xmi.idref="xmi.13"/>
                           </HAREL:StateVertex.incoming>
                        </HAREL:Connector>
                     </HAREL:StateChart.States>
                     <HAREL:StateChart.Transitions>
                        <HAREL:Transition xmi.id="xmi.11">
                           <HAREL:Transition.statechart>
                              <HAREL:StateChart xmi.idref="xmi.3"/>
                           </HAREL:Transition.statechart>
                           <HAREL:Expression>EVENTO[GUARDIA]/AZIONE</HAREL:Expression>
                              <HAREL:Transition.source>
                                 <HAREL:StateVertex xmi.idref="xmi.5"/>
                              </HAREL:Transition.source>
                              <HAREL:Transition.target>
                                 <HAREL:StateVertex xmi.idref="xmi.6"/>
                              </HAREL:Transition.target>
                           </HAREL:Transition>
                           <HAREL:Transition xmi.id="xmi.12">
                              <HAREL:Transition.statechart>
                                 <HAREL:StateChart xmi.idref="xmi.3"/>
                              </HAREL:Transition.statechart>
                              <HAREL:Expression></HAREL:Expression>
                              <HAREL:Transition.source>
                                 <HAREL:StateVertex xmi.idref="xmi.9"/>
                              </HAREL:Transition.source>
                              <HAREL:Transition.target>
                                 <HAREL:StateVertex xmi.idref="xmi.5"/>
                              </HAREL:Transition.target>
                           </HAREL:Transition>
                           <HAREL:Transition xmi.id="xmi.13">
                              <HAREL:Transition.statechart>
                                 <HAREL:StateChart xmi.idref="xmi.3"/>
                              </HAREL:Transition.statechart>
                              <HAREL:Expression>EVENTO[GUARDIA]/AZIONE</HAREL:Expression>
                              <HAREL:Transition.source>
                                 <HAREL:StateVertex xmi.idref="xmi.6"/>
                              </HAREL:Transition.source>
                              <HAREL:Transition.target>
                                 <HAREL:StateVertex xmi.idref="xmi.10"/>
                              </HAREL:Transition.target>
                           </HAREL:Transition>
                        </HAREL:StateChart.Transitions>
                        <HAREL:StateChart.Events>
                           <HAREL:Event name="AZIONE" xmi.id ="xmi.14" />
                              <HAREL:Event name="EVENTO" xmi.id ="xmi.15" />
                           </HAREL:StateChart.Events>
                        </HAREL:StateChart>
                     </HAREL:ownedElement>
                  </HAREL:Project>
               </HAREL>
            </XMI.content>
         </XMI>
         <HAREL:StateChart.Events>
            <HAREL:Event name="AZIONE" xmi.id ="xmi.14" />
            <HAREL:Event name="EVENTO" xmi.id ="xmi.15" />
         </HAREL:StateChart.Events>
      </HAREL:StateChart>
   </HAREL:ownedElement>
</HAREL:Project>
</HAREL>
</XMI.content>
```

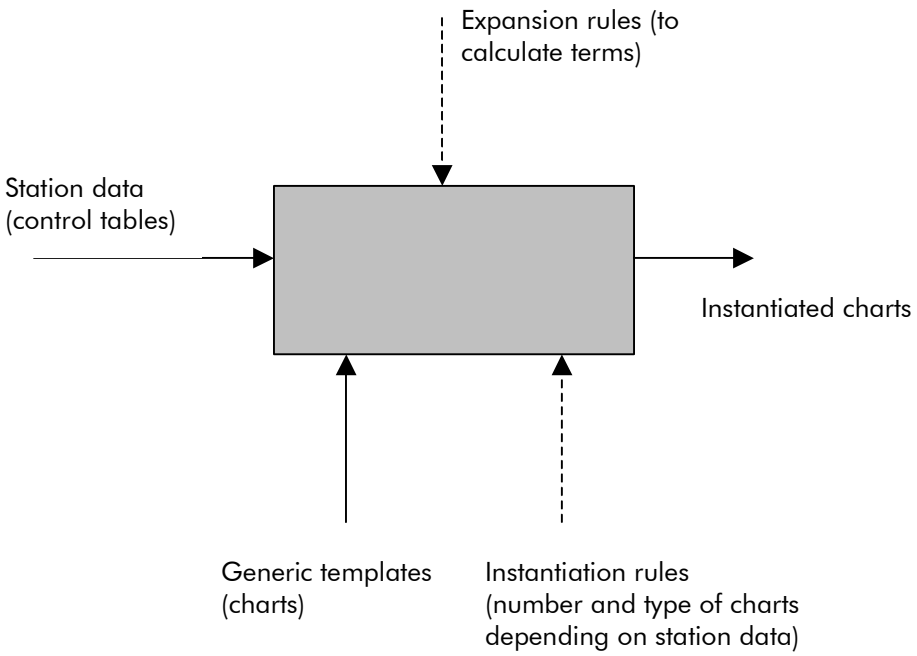
The XMI exporter

```
</XMI>
```

5 The instantiator tool.

5.1 Brief introduction to implementation

The implemented prototype is described in the following picture:



In this section a short introduction to the implementation of the tool is described, for more information about description of the implementation, refer to the documents *"*inSTanCe Reference Manual 0.1*"* [3]. The dotted lines in the picture above identifies that expansion and instantiation rules have not been managed in a general way. In other words, a simplified management of these rules have been implemented.

5.1.1 Classes

The instantiator tool.

The table below shows a list of some classes used to implement the tool, with a brief description.

Class	Description
ConditionTable	Singleton class recording control table
ConditionTableRow	Class containing control table row information
Dictionary	Singleton class storing the dictionary
DictionaryItem	Class containing information about dictionary items
FileLogger	Class redirecting log messages to file
InitPrint	Singleton class implementing Printable
Logger	Singleton class for logging
Options	Singleton class managing command line options
Output	Singleton class for redirecting the output
Printable	abstact class for representing printable objects
Route	Class representing routes
RouteStateChartInstance	Class representing generic instantiable statecharts
State	Class representing a statechart state
StateChart	Class representing statecharts
StateChartInstance	Abstact class (interface)
StationElement	Abstract class representing station elements
Switch	Class representing a switch
SwitchStateChartInstance	Class representing a instantiable switch statechart
Track	Class representing track circuit (Zone)
Transition	Class representing transitions

5.1.2 Class diagram

In Figure 11 the class diagram in UML format is shown.

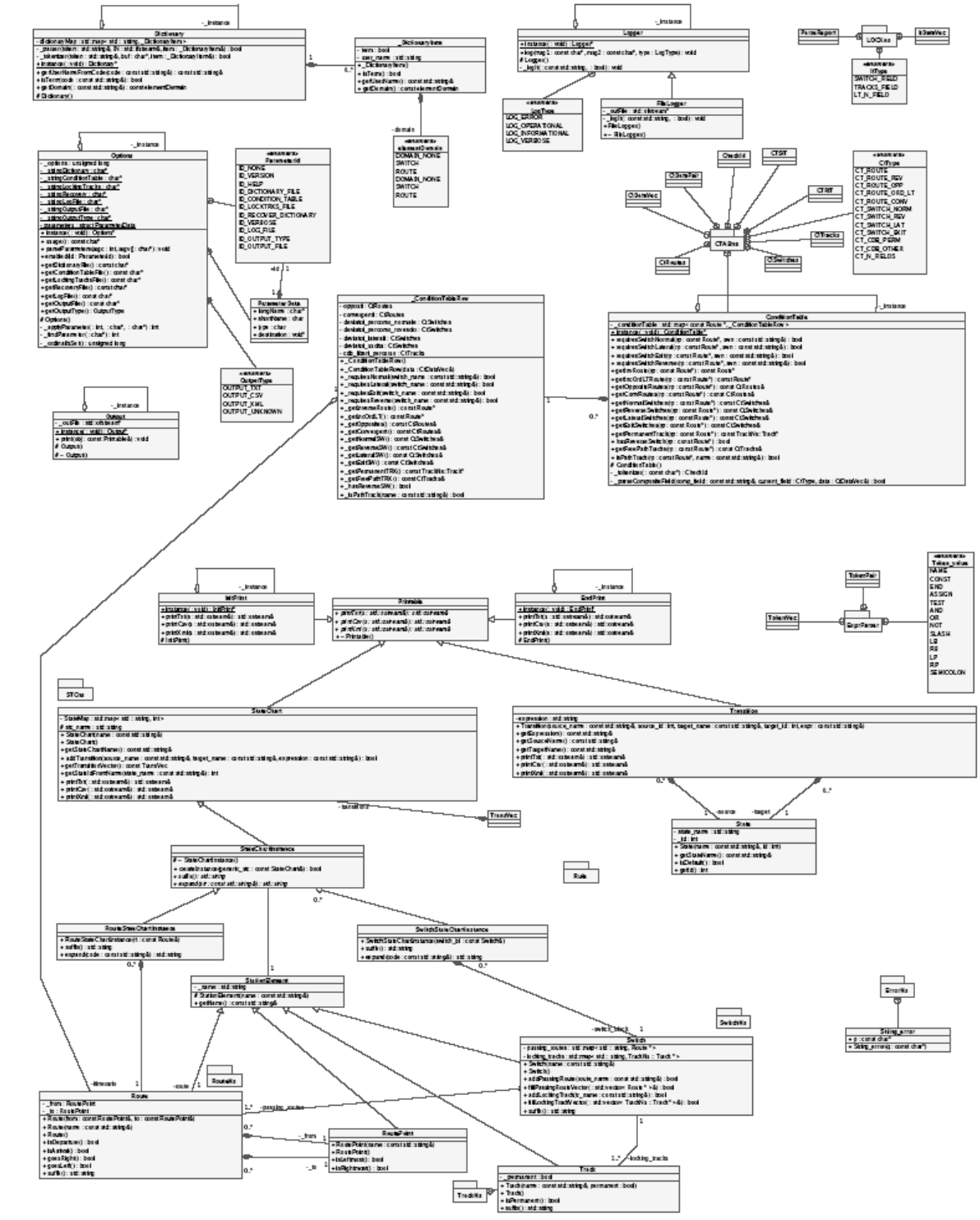


Figure 11 – class diagram

The instantiator tool.

5.1.3 Input files

5.1.3.1 Dictionary

Using the dictionary the user may change the name of terms which will be expanded. This feature is managed using a map which allow to join to an internal predefined code a new symbol user defined.
The choice of coding in the program the rules to expand the terms (*expansion rules*) was made to reduce the implementation time, considering that the goal of the cooperation was to implement a prototype tool. In the next releases, a way allowing the user to describe the expansion rules can be considered.

The dictionary file is in CSV format: each dictionary line includes four fields:

- CODE: represents the internal code used by the program to associate rules with TERMS.

- DOMAIN: it is the symbol domain, example of implemented domains are:

  - ROUTE: the symbol belongs to route domain

  - SWITCH: the symbol belongs to switch domain

  - DOMAIN NONE

  For the domains, in order to reduce the workload, the choice of considering two fixed domains was made, similarly to what was done for the expansion rules. More general domain specifications (*instantiation rules*) can be implemented in the next releases.

- TERM: boolean field, represents which rule has to be used to manage it (general rule or proper expansion rule)

- USER NAME: it is the user defined term corresponding to CODE.

Comment line are delimited by the # symbol.

5.1.3.2 Control table

This is the main input, it is implemented using a CSV file. Several fields have been defined:

The instantiator tool.

```
CT_ROUTE;
CT_ROUTE_REV;
CT_ROUTE_OPP;
CT_ROUTE_ORD_LT;
CT_ROUTE_CONV;
CT_SWITCH_GAUCHE;
SWITCH_DROITE;
SIGNAL_NOT_ORIGIN;
CT_SWITCH_EXIT;
CT_CDB_PERM;
CT_CDB_OTHER;
```

CT ROUTE: route name. This field includes one value.

Sintax: <starting place> - <arrival place>.

CT ROUTE REV: for incompatible routes.

CT SWITCH DROITE: switches required in normal position by that route

CT SWITCH GAUCHE: switches required in reverse position by that route

SIGNAL NOT ORIGIN

CT SWITCH EXIT

CT CDB PERM: it represents the ZONE related to the route

CT CDB OTHER: track circuits required by the route.

5.1.4 Implementation

5.1.4.1 Phase 1: System Setup

The system has been implemented in a general way, in fact it is not related to a specific layout.

Step 1: dictionary loading.
A parser has been used to elaborate the four field structure of this file. Each line of the file is analyzed to verify the syntax, after this verification it is sent to a tokenizer.

The instantiator tool.

It contains also an error handler to manage invalid domains, no boolean value of terms, and so on.

Step 2: generic statecharts loading.
This phase allocates memory space to statecharts objects, these statecharts will be expanded. These statecharts are coded into an external file provided by the XMI exporter.

The procedure required to allocate each generic statechart is the following one:

- an object of StateChart class is instantiated passing to the constructor its name;
- for each transition the addTransition method is called passing it the strings containing the source state and the target and the expression on it;
- relating to the kind of station element the object instantiated is added to the respective list of objects (e.g. route statecharts vector).

Step 3: control table loading.
A dedicated parser has been used. The parsing method is similar to that used for the dictionary, after the parsing the line is sent to a tokenizer. After the identification of the strings an object related to the read string will be instantiated. The constructor of ConditionTableRow instances an object, setting the attributes to the values identified into the string.

5.1.4.2 Phase 2: Instantiation

Instantiation is the core of the program. During this phase the generic charts are instantiated and expanded.

Step 1:
For each generic switch statechart an object of SwitchStateChartInstance class is allocated passing to the constuctor the pointer to the corresponding switch. The createInstance method is then called on the instantiated object, passing it the generic statechart, and the expanded object is added to the output statecharts list.

The same sequence of actions is repeated for each route, allocating a RouteStateChartInstance object.

Step 2: expansion using createInstance methods.
The abstract class StateChartInstance, implemented by two classes (one for switch, SwitchStateChartInstance, and one for routes, RouteStateChartInstance), defines the createInstance method which receives in input the generic statechart that has to be expanded. The expansion requires the following steps:

The instantiator tool.

- the statechart name is generated adding the exact suffix to the generic name (calling the suffix method);
- for each generic transition:
  – the source state name is created adding the correct suffix;
  – the target state name is created adding the correct suffix;
  – the generic expression is analyzed token by token.

The expanded expression is generated running for each token the following procedure:

- if the token is a constant or a special symbol it will be left unchanged;
- otherwise the expand method will be called passing it the token;
- the so generated transition will be added to the expanded statechart.

Step 3: the Rule::expand methods.
The methods act on switches and routes separately. They look for the corresponding rule to the passed term which has to be expanded. The related rule will be called expanding the term. The functions implementing the rules are coded in order to interrogate the Control Table using some methods that return station elements corresponding to the required rules. The set of these elements is passed to a function object that expands them with the correct suffix, adding the separation symbols (==) and constants, and joining them in AND, OR expressions.

6 Post processing activities (graphic representation of instantiated statecharts)

In this section it is illustrated all the output formats that the tool is able to produce. The following examples are related to the layout shown in Figure 12.

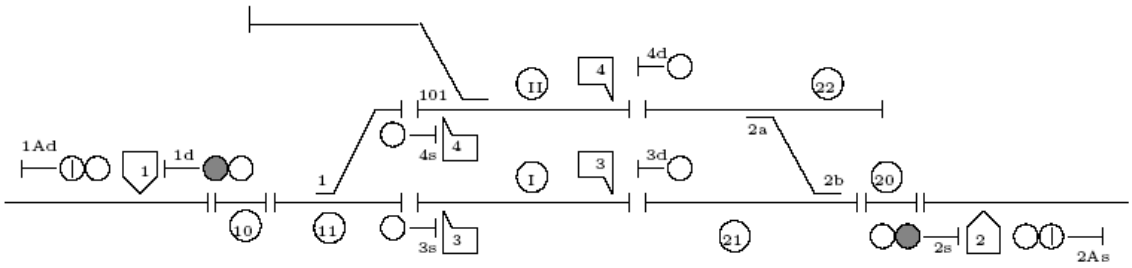


Figure 12 – sample layout

The simplest output manageable by the instantiator is text format, the command to produce this kind of output is the following:

```
$ instance -t txt
```

or

```
$ instance
```

This is an example of output in text format, the result is shown below:

```
...
------------------------------------------------------------
StateChart Name: TENTATO_COMANDO_STC_1
Transition table:
_Source_ _Target_ _Expression_
DEV_1_COMANDATO_NORMALE_1 DEV_1_COMANDATO_ROVESCIO_1 [MDEV_1==1] and
[ADEV_1==1]/DEV_COMANDATO_ROVESCIO_1=1;DEV_COMANDATO_NORMALE_1=0
DEV_1_COMANDATO_ROVESCIO_1 DEV_1_COMANDATO_NORMALE_1 [MDEV_1==0] and
[ADEV_1==1]/DEV_COMANDATO_NORMALE_1=1;DEV_COMANDATO_ROVESCIO_1=0
Default_1 IDLE_1
IDLE_1 DEV_1_COMANDATO_NORMALE_1 [MDEV_1==0] and
[ADEV_1==1]/DEV_COMANDATO_NORMALE_1=1;DEV_COMANDATO_ROVESCIO_1=0
IDLE_1 DEV_1_COMANDATO_ROVESCIO_1 [MDEV_1==1] and
[ADEV_1==1]/DEV_COMANDATO_ROVESCIO_1=1;DEV_COMANDATO_NORMALE_1=0
------------------------------------------------------------
```

```
StateChart Name: COMB_MANOVRA_STC_101
Transition table:
...
```

Sometime it is useful to be able to interface to databases, for this reason the CSV format is provided. We have supposed that the rows are represented by statecharts transitions.

The command to create a CSV file is:

```
$ instance -t csv -o transition.csv
```

This command creates the transition.csv file

The first part of the file is shown below

```
#CSV file generated by instance v. 0.1
#StateChart Name: COMB_MANOVRA_STC_1
#Transition table:
#"SOURCE_STATE" ; "TARGET_STATE" ; "EXPRESSION"

"Default_1" ; "NORMALE_1" ; "/MDEV_1=0"
"NORMALE_1" ; "ROVESCIO_1" ; "[(MEM_IT_1_4_D_ARR==1) or
     (MEM_IT_4_1_S_PAR==1)] and [ADEV_1==0]/MDEV_1=1"
"ROVESCIO_1" ; "NORMALE_1" ; "[(MEM_IT_1_3_D_ARR==1) or
     (MEM_IT_3_1_S_PAR==1)] and [ADEV_1==0]/MDEV_1=0"

#end of COMB_MANOVRA_STC_1
#StateChart Name: AUX_MANOVRA_STC_1
...
```

However, the most interesting output format is the XML format.
To obtain interesting results some XSLT transformation files that have been developed have to be used.
The first step is the creation of an XML file (station.xml) using the tool. To perform this activity the following command has to be typed:

```
$ instance -t xml -o station.xml
```

After this instantiation activity the generated xml file is readable with a browser. Furthermore, if in the directory containing station.xml there is also the file statechart.xsl, a browser (able to read XSLT transformations) shows a graphical format of output as well, as shown in the picture below.

Post processing activities (graphic representation of instantiated statecharts)

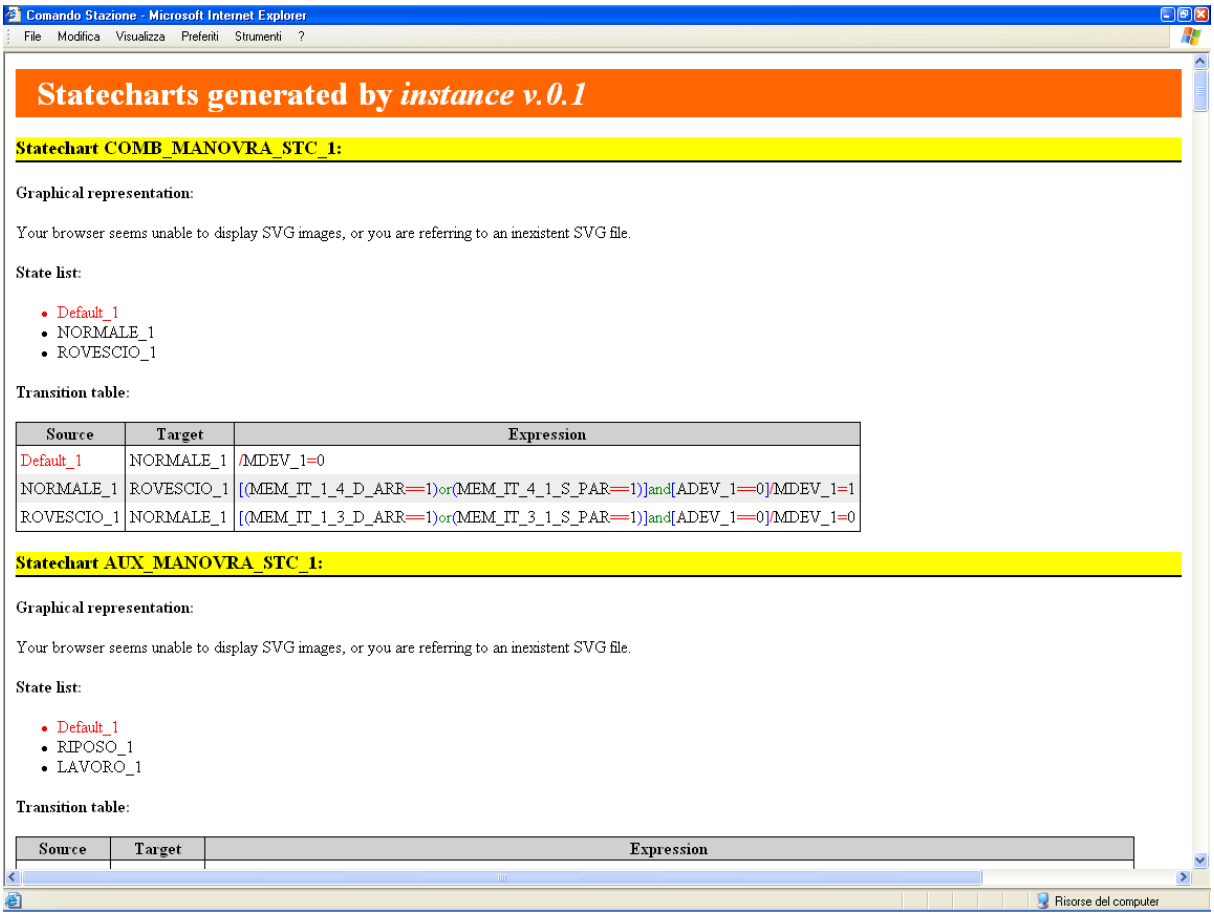


Figure 13 – graphical representation of instantiated charts

By transforming the xml output using the stylesheet svg.xsl and stcsvg.xsl, an HTML file containing also a reduced graphical representation of statecharts can be obtained in svg format.

Furthermore, the XML file can be transformed into a .DOT file using another stylesheet (dot.xsl). The resulting .DOT file can then be supplied to an external COTS tool, named Graphviz, to obtain high quality graphs representing the instantiated statecharts. In Figure 14 and Figure 15 examples of this kind of representation are shown. Furthermore, vectorial statecharts can also be created.

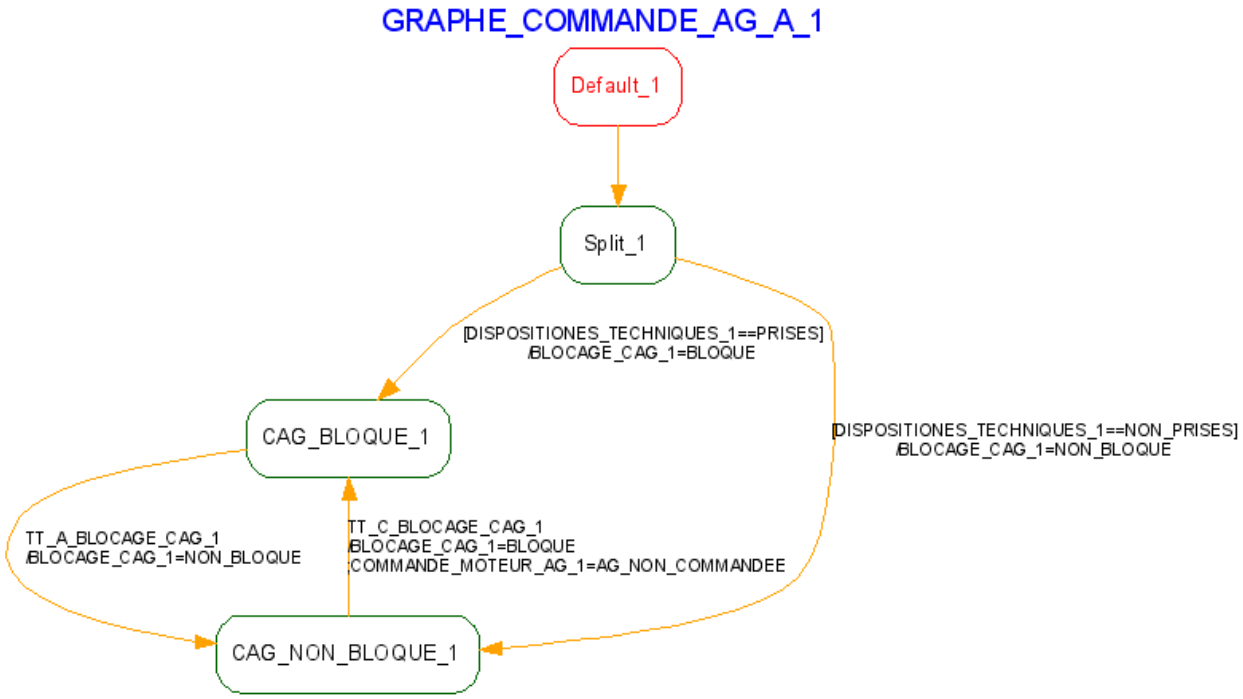


Figure 14 – an example of statechart generated by graphviz

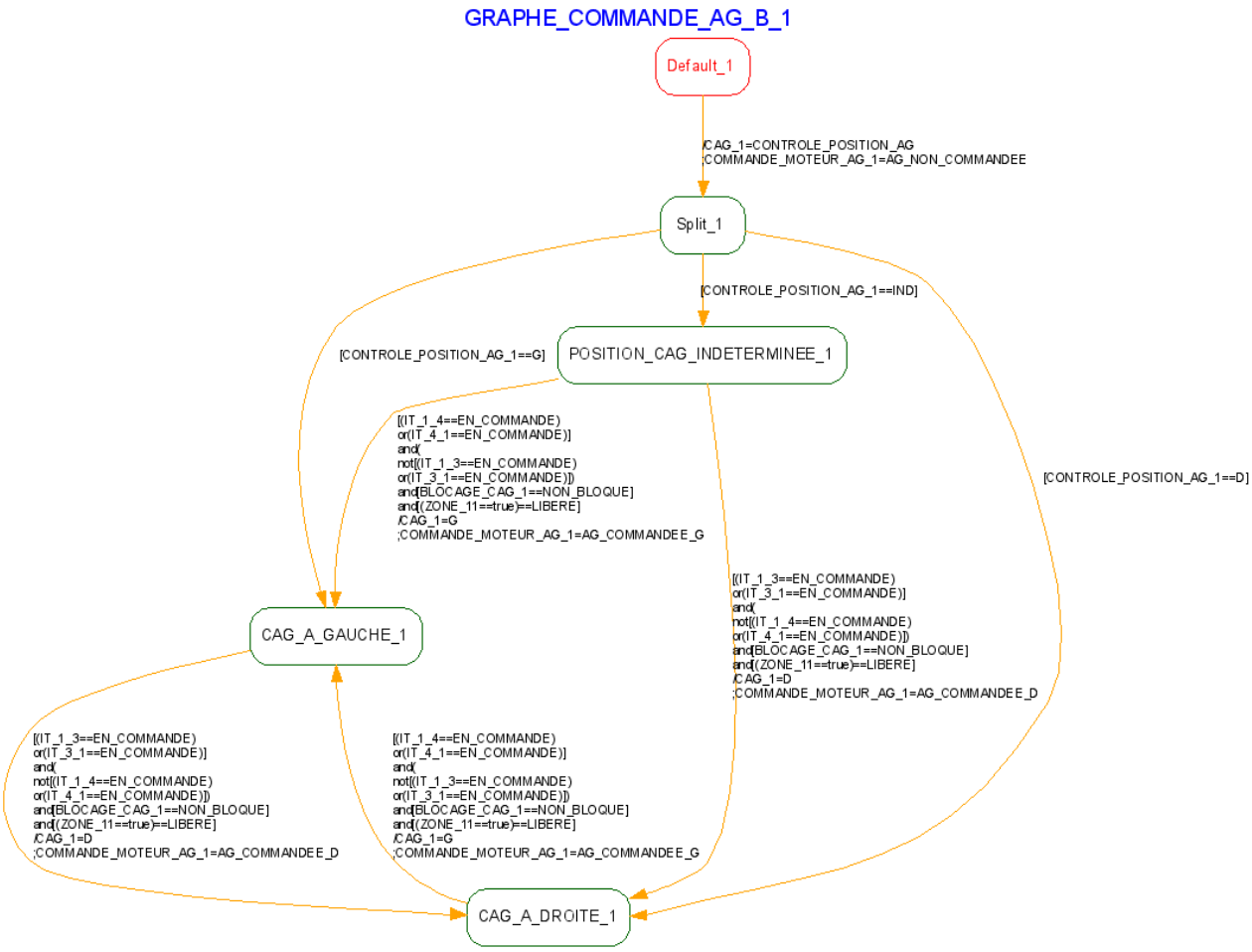


Figure 15 - another example of statechart generated by graphviz

6.1 Postprocessing details

The output produced by the instantiator can also be redirected into a file. The file formats are:

- log file (optional);
- text file, for debug use;
- CSV file, to interface with databases;
- XML file (it could be transformed in different outputs).

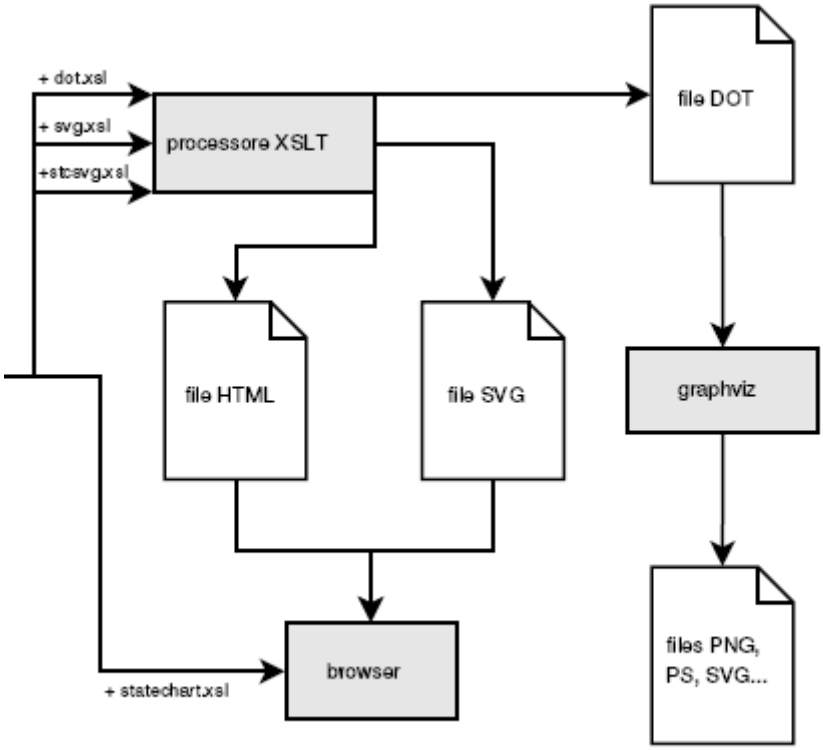


Figure 16 - output flow

The Figure 16 shows the relation between files and the process flow.

It is evident that the XML format may be elaborated to obtain other types of representations.

6.1.1 XSLT transformations

XSLT transformations are the following ones:

- statechart.xsl: it transforms the XML file to a HTML file in which a list of instantiated statecharts is maintained, and for each statecharts the list of states and transition of it is hilighted.
- svg.xsl: it transforms the XML file to an SVG file containing the graphical representation of instantiated statecharts.
- stcsvg.xsl: it transforms the XML file to a HTML file containing also a graphical representation of instatiated statecharts.
- dot.xsl: it transforms the XML file to a DOT file.

Post processing activities (graphic representation of instantiated statecharts)

If the XML output file uses the default stylesheet statechart.xsl, the XML file could be read directly using a browser with XSLT processor (for instance with Gecko engine like Mozilla, Firefox. . . but also Explorer). To use the other created transformations a tool that is able to process transformations is required. During the tests sabcmd (Sablotron package) has been used.

An example of postprocessing is the following:

```
$ sabcmd svg.xsl station.xml > station.svg
```

It produces the instantiated statecharts in SVG format. At this stage the file can be either open or reprocessed:

```
$ sabcmd stcsvg.xsl station.xml > station.html
```

It generates the HTML file. The last transformation is the DOT transformation:

```
$ sabcmd dot.svg station.xml > station.dot
```

The output file could be passed to Graphviz, in order to generate graphical formats (PNG, PostScript, SVG. . .).

6.2 Outputs creation

At the end of the elaboration the tool prints the list of the instantiated statecharts. This function is obtained using the print method of Output class. The output class manages print functions addressing the output in relation to the option chosen by the user (video or file).

6.2.1 XML Output: file structure

The node hierarchy of XML file is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<?xml-stylesheet type="text/xsl" href="statechart.xsl"?>
```

Statechart.xsl has been defined as the XSLT default transformation.
The hierarchy of nodes is shown below:

```
<STCdocument>
```

Post processing activities (graphic representation of instantiated statecharts)

It is the main node of the document, its attributes are:

name document name,
`idGenerator` name of program generating the document,
`vGenerator` program version,
`filename` XML filename.

For each statechart there are:

`<statechart>`

Containing nodes:

`<name>`

Containing statechart name;

`<content>`

It defines the statechart using its children nodes:

`<state-list>`

For each statechart state contains:

`<state>`

`<transition>`

Its attributes are:
`source` transition source state;
`target` transition target name;

`<expression>`

`<exprItem>`

6.2.2 CSV Output: file structure

Post processing activities (graphic representation of instantiated statecharts)

The CSV (Comma Separated Values) is a text format usually used to interface with databases.

- Information fields are separated by semicolon (';');
- Subfields are separated by comma (',');
- Character # represents a comment line.

Each line of CSV file includes three fields representing transitions.

```
#CSV file generated by instance v. 0.1
```

- for each statechart some comment lines can be inserted:

```
#StateChart Name:
```

statechart name

```
#Transition table:
#"SOURCE_STATE" ; "TARGET_STATE" ; "EXPRESSION"
```

- for each transition there are three columns separated by comma, for instance:

```
"Default_1" ; "NORMALE_1" ; "/MDEV_1=0"
...
```

6.3 Output graphical tranformations

Starting from an XML file format some transformations (XSLT) can be used to generate data in a more readable form.
These transformations are described below.

6.3.1 Statechart.xsl file

It transforms the XML file in a HTML file:

- section <head>:
  – field <title> document name;

Post processing activities (graphic representation of instantiated statecharts)

– field <style> attribute type="text/css" defining a Cascading Style Sheet;

- section <body>:
  – field <h1> (header) name and version;
  – for each statechart:
    - field <h3> statechart name;
    - field <h4> states list;
    - structure <ul> containing for each state:
      · field <li> state name;
    - field <h4> it shows the transition table
    - structure <table> (source, target and expression);
    - for each transition:
      - line <tr> containing the transition name;

  – field containing the document footer.

6.3.2 File dot.xsl

This transformation allows to obtain a file in a format corresponding to DOT graphs definition.
The methodology is illustrated below:

- for each statechart:
  – a graph object is defined with name and label related to the statechart name;
  – the attributes of nodes and edges are defined (color, character type, character dimension, and so on);

The obtained file, once processed by Graphviz, allows to create graphical representations of instantiated statecharts in several formats.

7 Conclusion, investigation areas and open topics

The collaboration between ISTI-CNR and ALSTOM has addressed several important issues about Model-based design of Interlocking Systems, but several issues have not been investigated. In the following table a set of open issues that are proposed as possible activities for a further collaboration is presented. Particularly, for each activity the estimated required effort is given.

The following table highlights the open issues identified in the 'Meeting Report' of the 19/4/05.

Proposal	**Id**	**Effort estimation**	**Comments**
Tests using a big French station (e.g. Melun).	**1**	4 m/m	Additional signaling support is required to French team [4 m/m] to evaluate the full SNCF model.
Diversity analysis.	**2**	9 m/m	The estimation includes the formal specification of the instantiation tool [1 m/m], the definition of equivalence between diversely generated graphs [2 m/m] and the development of a tool to verify the equivalence [6 m/m].
De-instantiation of graphs.	**3**	3 m/m	Feasibility analysis.
Formal demonstration of equivalence between Boolean equations and state charts.	**4**	8 m/m	Point (11) of the final report of the previous cooperation (study of the different alternatives and definition of possible tool support).
Off-line interpreter of instantiated charts.	**5**	10 m/m	Point (8) of the final report of the previous cooperation.

Off-line interpreter of charts translated in intermediate language.	**6**	9 m/m	Prototype development.
Reuse of current Boolean interpreter with instantiated charts.	**7**	3 m/m	Feasibility analysis (include the translation of charts in an equivalent boolean format).

The most interesting activities are pointed by numbers 6 and 7.