Department of Computer Systems

DoCS

UPPSALA UNIVERSITET

UL^{++} : an Object Oriented Language for UPPAAL

Preface

This report is the product of my Master's project, performed under the period June-October 1998.

The aim of this project was to design and implement a new language for UPPAAL. UPPAAL is a verification tool based on timed automata, developed jointly by Uppsala University and Aalborg University. Such a project was not a new idea, but this time the goal was to provide hierarchy and object orientation. Furthermore, we wanted to simulate our systems which implied that we wanted to translate our new language to the existing one in order to apply model checking without designing a new kernel for UPPAAL. The design of such a new kernel is planned in the future, but this work was not in the scope of this project. We have called the language UL^{++}, which stands for UPPAAL language^{++}.

I would like to thank my supervisor Wand Yi and also in alphabetical order Johan Bengtson, Fredrik Larsson, Justin Pearson and Paul Pettersson in Uppsala and Carsten Weise in Aalborg.

Alexandre DAVID
Uppsala University
Department of Computer Systems
adavid@student.docs.uu.se
October 31, 1998

Contents

List of Figures

List of Tables

Chapter 1

The Object Oriented Model

This work has been carried out with the experience of UPPAAL, which is the target of this language, and the object oriented language Java. The concepts we have taken come from both worlds. In this chapter, we intend to show the interest of UL^{++}. Then we will explain how we introduce the object model in the modeling of real time systems. Finally, we will go further into the core of UL^{++}.

1.1 Motivations for UL^{++}

1.1.1 Hierarchy

UPPAAL is a verification tool for real time systems and is appropriate for those which can be modeled as a collection of non deterministic processes, with finite control structures, clocks, synchronization and communication through channels and shared variables. UPPAAL is designed mainly to check invariants and reachability properties by exploring the state space of a system. However complex systems are difficult to model because UPPAAL only deals with simple states. A more precise description of UPPAAL can be found in UPPAAL *in a Nutshell* ([LPW97b]).

The introduction of hierarchy has two goals :

- Firstly, we want to deal with big systems in a comprehensive manner. This is to ease and to structure the modeling task.

- Secondly,we want to guide the verification task. This implies guiding the search in the state space by using static information, and helping the debugging process by providing hierarchical traces.

We illustrate these reasons with the DACAPO TDMA start-up protocol. This protocol is described in [LP97]. and was modeled in this paper by a test process, an init process, a bus process and three stations (a process each) which access the bus. Figure 1.1 gives the model of a station. With no hierarchy, it is not obvious to distinguish the different parts of this station.

It would be nice to have a language, with an associated graphical representation, which would provide a hierarchical overview as depicted in figure 1.2. It would be obvious that the stations have three different modes, and we could see the transitions between these modes. These modes are *normal mode, resynchronization mode* and *recover mode.*

Concerning the trace that we get when we verify the system, the user could get a trace like in figure 1.3. This long sequence of transitions hides the relevant information. One would certainly prefer a high level trace like in figure 1.4. In such a trace, the user could open or close a “mode” and navigate through the different levels of the hierarchy.

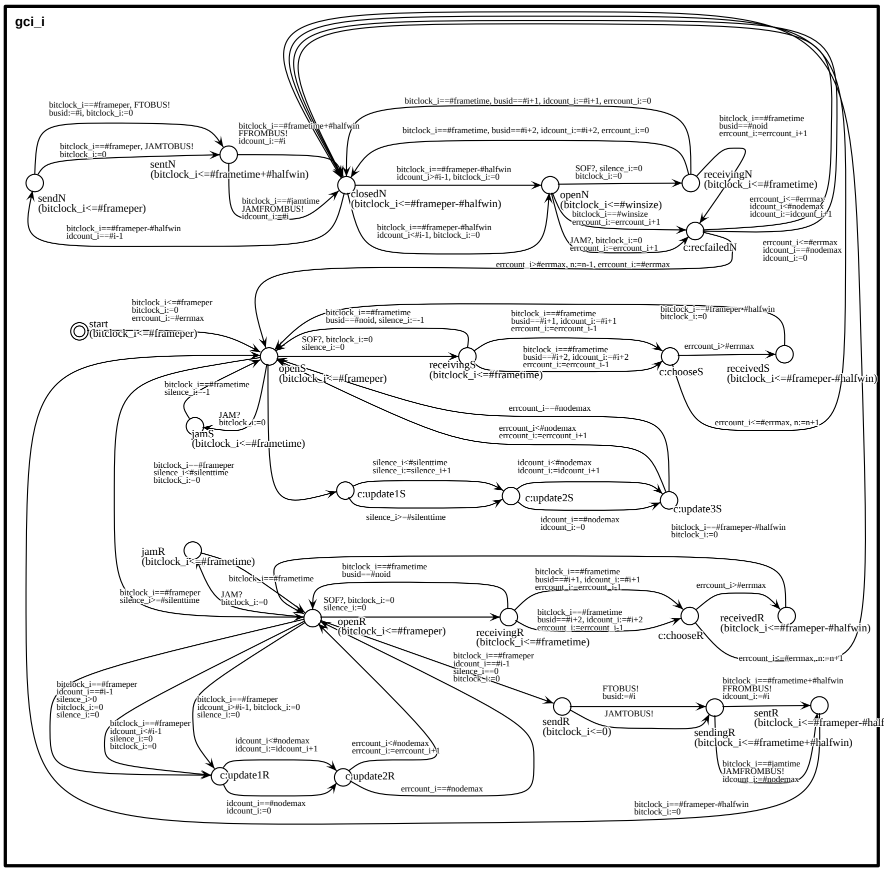


Figure 1.1: The station of the DACAPO TDMA start-up protocol

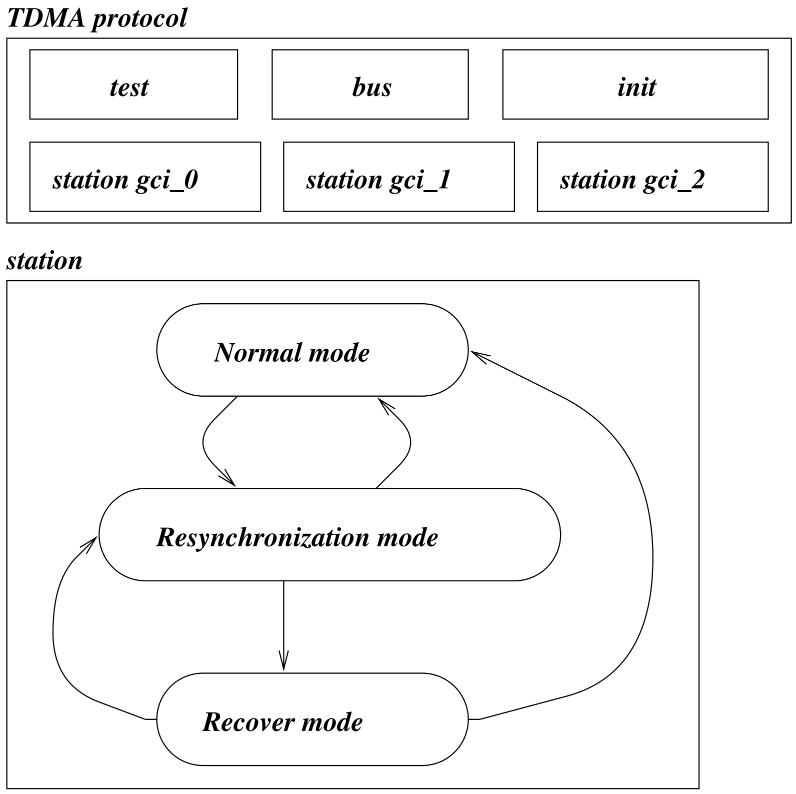


Figure 1.2: Hierarchical overview of the DACAPO TDMA start-up protocol

$$\left.\begin{array}{c}
(bus.idle, test.start, gci_0.sendN, gci_1.sendN, gci_2.sendN, \dots) \\
\downarrow \\
(bus.idle, test.normal, gci_0.sendN, gci_1.sendN, gci_2.sendN, \dots) \\
\downarrow \\
(bus.idle, test.normal, gci_0.sentN, gci_1.sendN, gci_2.sendN, \dots) \\
\downarrow \\
(bus.idle, test.normal, gci_0.sentN, gci_1.sentN, gci_2.sendN, \dots) \\
\downarrow \\
(bus.idle, test.normal, gci_0.sentN, gci_1.sentN, gci_2.sentN, \dots) \\
\downarrow \\
\vdots
\end{array}\right\} Normal\ mode$$

Figure 1.3: A possible trace of a system

$$\left.\begin{array}{c} Normal\ mode \\ \downarrow \\ Resynchronization\ mode \\ \downarrow \\ Recover\ mode \\ \downarrow \\ Normal\ mode \\ \downarrow \\ \vdots \end{array}\right\} hierarchical\ trace$$

Figure 1.4: A hierarchical trace of the same system

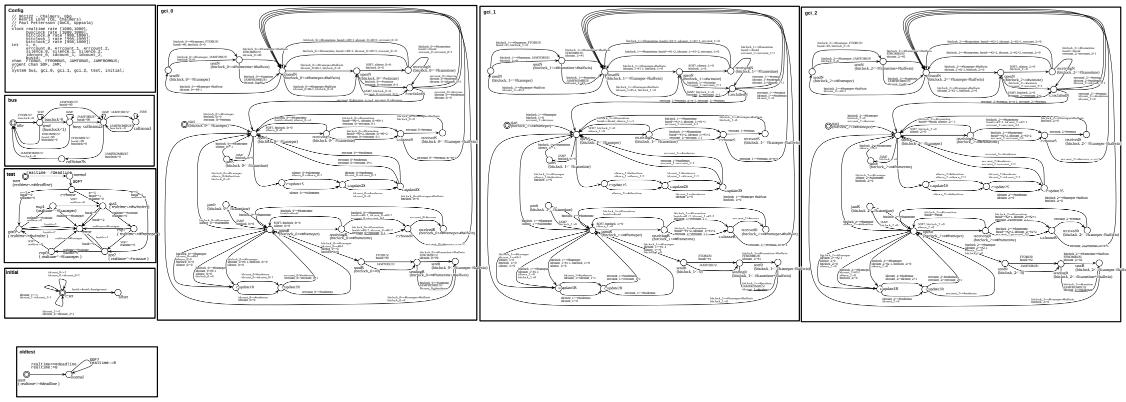


Figure 1.5: The complete model of DACAPO TDMA start-up protocol

1.1.2 Object Orientation

Modeling language without object concept are used, so why do we introduce objects? If one looks at the whole model of the previous example in figure 1.5, one notice that the three stations are the same, and this complete overview does not provide any abstraction like in figure 1.2. Introducing objects in our language has the two following goals :

- Firstly, we want to provide abstraction in the modeling process. One would like to look at processes at a high level, to see the interactions between them, for example the channel synchronizations. Inside the processes, one would like to see relevant sub parts, as we have seen before.

- Secondly, we want to model by modules, which would be connected together by interfaces, like in the object paradigm. To illustrate this idea, one could think about a train control system. It is a big system, with several distinct parts or sub-systems. The team which works on this system is divided into groups of people, who could meet to decide the main structures and connect them with interfaces. They would then work on their modules. Furthermore, this object approach allows to reuse components of a system since they are independant and connected via interfaces.

Another obvious reason for introducing an object oriented language is that this has become widely used in the industry world and the object paradigm is appreciated.

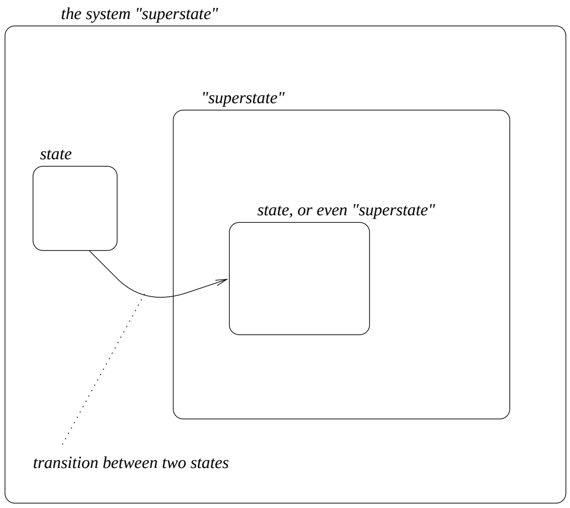


Figure 1.6: Illustration of UML statecharts

1.1.3 Why not UML Statecharts?

One could wonder why we did not adopt UML statecharts since UML has become a standard. See [DBP98] for more information on UML. First of all, UML statecharts is not object oriented, and we wanted an object model, so it did not fit our needs. Secondly, there is no separate module concept that we would like to use. In the same idea, one could notice that transition across different levels of hierarchy are allowed in statecharts, as illustrated in figure 1.6.

This kind of transition forbids the modular design as exposed before by the simple fact that the team which decides to define this transition has to know the target of the transition, which is inside a superstate. Interfaces between modules are thus not defined.

In addition to this, there is no support for “macros”, as we would expect for our TDMA example. It would be usefull to have such constructions, to avoid error prone copy-paste-and-modify.

To conclude on statecharts, it provides only hierarchy, which is not enough for us.

1.2 Introducing Objects

1.2.1 Object Principles

In our everyday life, we are dealing with objects. We classify these objects intuitively with respect to their common properties. If we condider the example of figure 1.7, we can regroup cars and motorbikes in a class called vehicles. This class of objects catches the common properties of cars and motorbikes, and we could imagine that cars are composed of different sub-classes or types of cars.

The idea is the same when we deal with object oriented programming languages, where we have classes as well. These classes are organised in a hierarchical way, and the parent class (at the top) regroups the common properties, or operations and data that the extensions have. So a *class* could be seen as an abstraction of data and operations manipulating them.

Now, if we condider a real time system, as in figure 1.8, we have several boards connected into slots on a mother board. On each boards, we have a processor which is running a program. At the system level, we can see a process on each board, communicating with other processes.

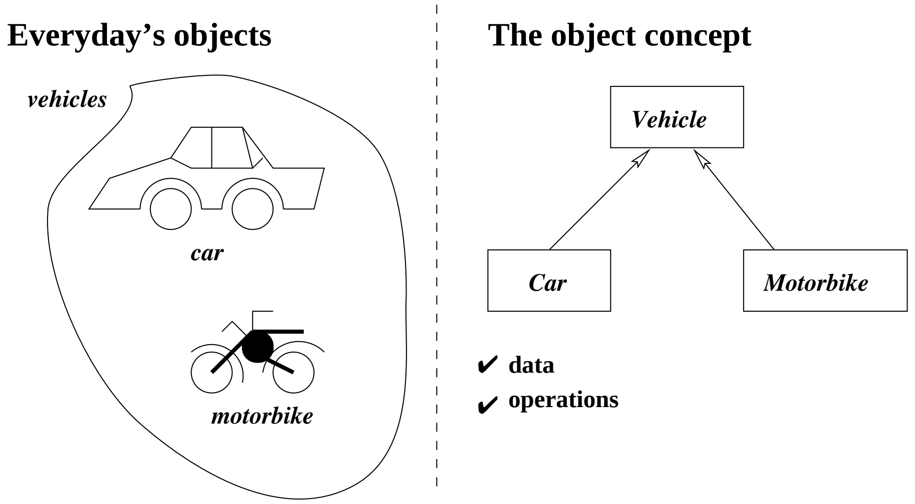


Figure 1.7: Object principles

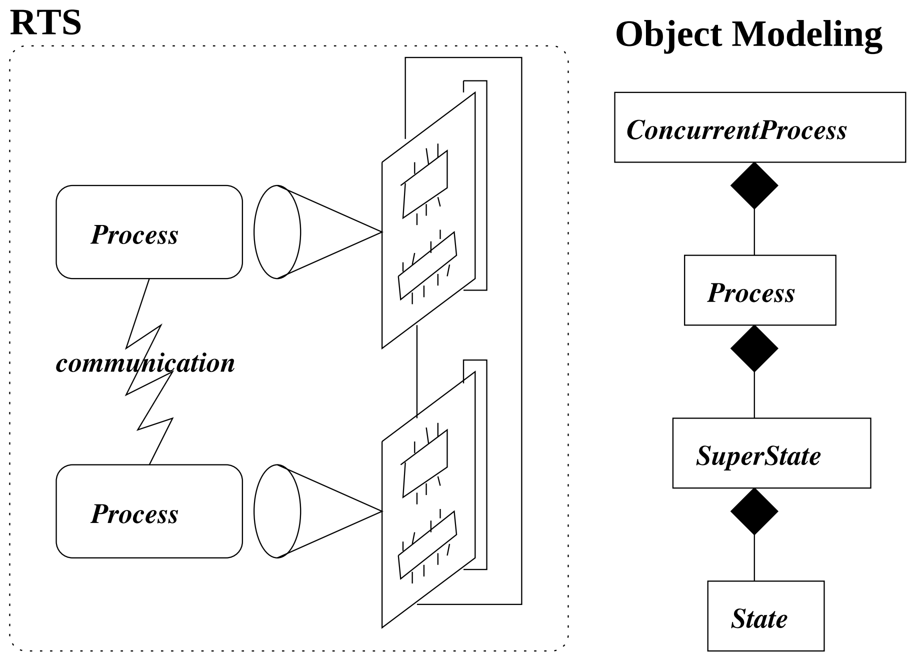


Figure 1.8: Object in real time systems

To make the object equivalence, we can imagine a "super" process (the mother board) which would be a **collection of** processes (the plugged boards). The processes would be collections of superstates (main parts of the execution) and states. UL^{++} is based on this structure.

1.2.2 UL^{++} Object Model

The core of UL^{++} can be divided into three concepts. The first is the hierarchy, with a composition rule which says how to include objects. The second is the two types of objects : the data and the control objects. The last one is the interfaces, or how to use and connect the objects.

Hierarchy In our model, we have two different hierarchies :

- the system hierarchy : the system is a *ConcurrentProcess* or a *Process*. A ConcurrentProcess is a composition of Process objects (and ConcurrentProcess).
- the state hierarchy : a Process is a composition of *SuperState* and *State* objects. A SuperState is a composition of State objects (and SuperState).

We provide by this way abstraction at the system level and at the process level. Modeling a system in UL^{++} consists of extending these primitive classes and composing these extended classes with respect to the above hierarchies. This two hierarchies define the ***composition rule*** of the ***composite objects*** in UL^{++}. The composite classes are : ConcurrentProcess, Process and SuperState.

Inside a class To define a class, one has to declare two kinds of objects :

- the data objects : they are the data variables. They are local variables to the class being defined and they are private to this class. They are clocks, channels or integers for example.
- the control objects : they define the static control structure of our automaton. They are states or transitions for example.

Connecting classes To use a class, when composing, two kinds of interfaces are provided :

- the data interface : it allows to manipulate data from outside, or in another sense, to have different instances of a class. This interface is similar to constructors in programming languages, but is simpler to use. It has a substitution semantics, which will be precised in the semantics chapter.
- the control interface : this interface is used to connect superstates together. As we said before, we use modules, so that one can not make a transition from outside to a state inside a superstate. The superstate provides public interfaces instead, where we can define our transitions. This approach gives an abstraction on what is inside the superstate. We just need to know what these interfaces are used for.

These interfaces allow us to achieve the level of abstraction that we wanted. The system is defined with modules in a rather safe way since the modules are closed and provide interfaces for connections. See the graphical syntax section to get a better idea of this principle.

The explicit class model The figure 1.9 gives an overview on the class hierarchy in UL^{++}, as well as the composition hierarchy which is the composition rule in UL^{++}. The hierarchy is split into two branches : the control and the data objects. On the control branch, one notices the hierarchies of states and processes. Processes and concurrent processes are both process objects, and states and superstates are both state objects. The connections are done with the state connective and the connection objects. They are the main control objects. The data objects are divided into dynamic and static data. The dynamic objects are clocks, channels and urgent channels (with obvious class names). The static ones are integers, bounded integers, boolean and arrays. As we will see later, the implementation follows this class hierarchy.

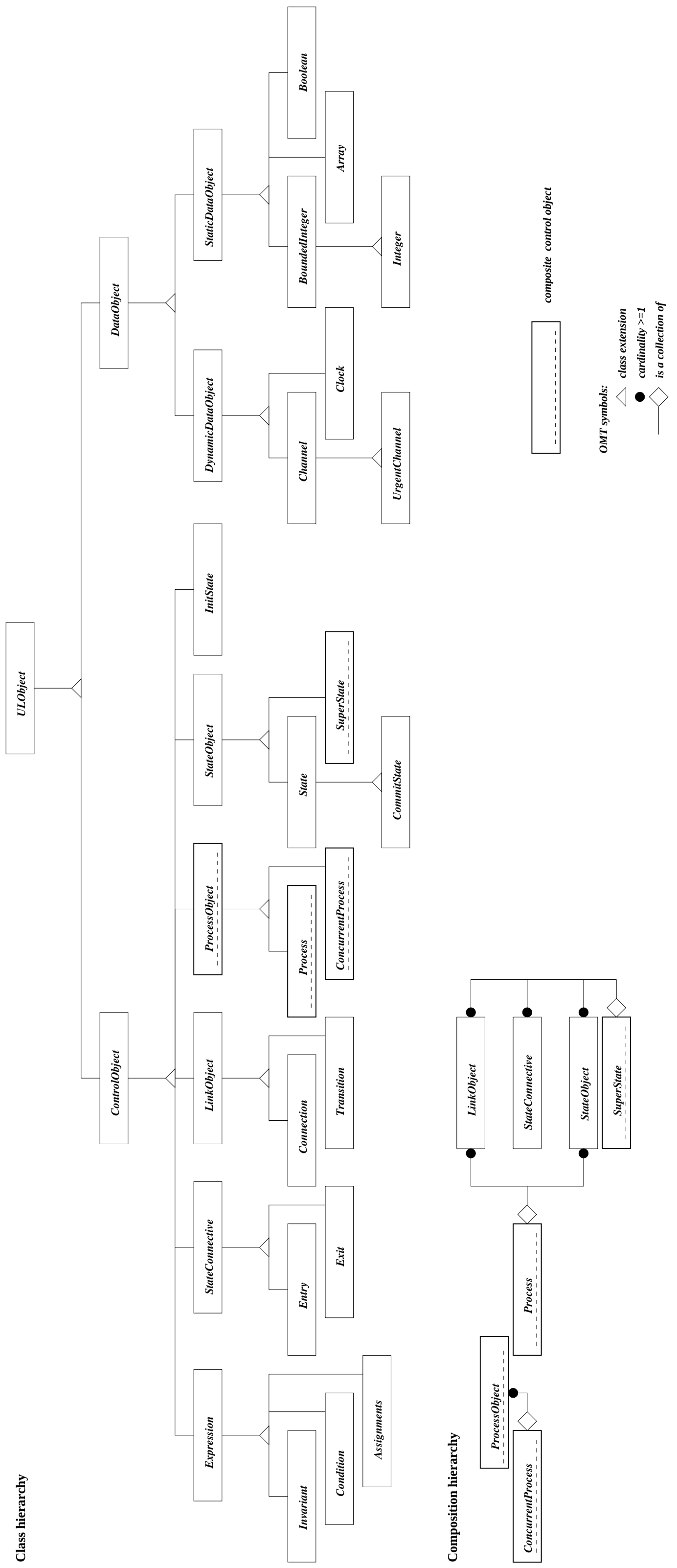


Figure 1.9: Overview of UL++ class hierarchy and the composition rule

Connecting the superstates Superstates may be first considered as states, with a total abstraction of their content. In this view, transitions are defined ordinary, but when defining the superstate, entry points will be defined and the exit points as well, inside the superstate. These kinds of entry/exit points are hidden. Another way of using superstate is via public entry and exit interfaces. A transition can then be defined to an entry or from an exit. Entry and exit points are connected inside the superstate to a state (or a superstate, or another entry or exit). These entry/exit, that we've called connectives are the control interface of a superstate. Note that we define oriented connections from an entry to a state, or from a state to an exit, as an example.

1.2.3 Why do we call UL^{++} an Object Oriented Language?

Sceptical people will argue that we are cheating when using the term Object Oriented. They are not totally wrong, but the language is a modeling language, not a programming language and it takes notions from the latter world. This may be the source of a difficult debate. However, we provide the following features which are proper to Object Oriented languages :

- extension : one can extend the composite classes ConcurrentProcess, Process and SuperState. The modeling task is based on this. The extensions may also be extended. We address inheritance by the same manner : we allow overriding with some rules and shadowed variables are taken into account. This will be exposed into details in the syntax and semantics sections.

- encapsulation : data and control members are private, and the modeling task respects the concepts of Object Oriented languages. The definition of interfaces is similar to the public declaration of methods, which makes them available to the world, in the C^{++} language for example.

Concerning the other objects, we look at them as objects, conceptually, in order to have a uniform language.

Chapter 2

Syntax of UL^{++}

UL^{++} has a graphical and a textual syntax. The graphical representation is intuitive, which is important since the main part in the implementation of UL^{++} was the graphical editor. There is an equivalence of course with the textual representation which may remind Uppaal language.

2.1 Graphical Syntax

Every control object has a corresponding graphical representation. We propose as well a representation for channels, even if they are not control objects. The graphical representation is depicted in figure 2.1.

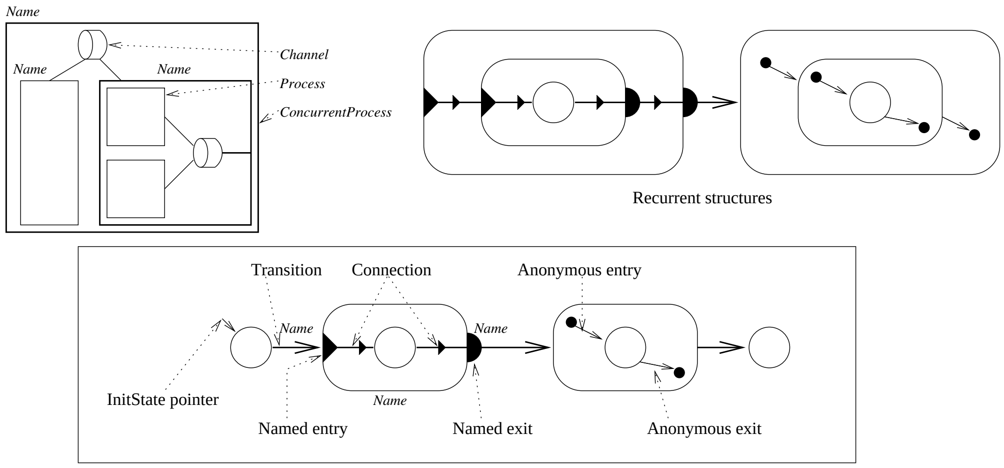


Figure 2.1: UL^{++} graphical syntax.

ConcurrentProcess objects are represented by thick rectangles. They may contain Concurrent-Process and Process objects. They have any *DataObject* objects as data.

Process Objects are represented by rectangles. They may contain SuperState and State objects. They have any DataObject objects as data, except channels. They have an initial state, indicated by one *InitState* pointer, which is a thin arrow.

SuperState objects are represented by rounded rectangles. They may contain SuperState and State objects. They have only StaticDataObject objects.

ConcurrentProcess, Process and SuperState objects are **composite objects** and they have a name. The corresponding classes have a name as well[1]. Data members are not represented

[1] An object is an instance of a class, so a class has a name and an instance too.

on the graphics, except for the channels. They should be given separately, when adressing the corresponding owner object, in order not to pollute the graphics.

State objects are represented by circles. They have a name and may have an invariant, represented by a label.

Transition objects are thick arrows, from the source state to the destination state. They may have a name, a condition (guard and synchronization terms) or/and an action. Attached Condition and Action objects are represented by labels. A Transition object may also be defined from a pre-connected exit point or to a pre-connected entry point. They may also have a superstate as source or destination[2].

Connection objects are oriented and are represented by thick lines with a triangle in the middle to indicate the direction. They have no name and no attribute.

Entry objects may be anonymous and are then represented inside a superstate by a thin arrow from a point to the desired state. Pre-connected named Entry objects are represented by filled triangles.

Exit objects may be anonymous and are then represented inside a superstate by a thin arrow from the desired state to a point. There may be several exit points. Otherwise, they are named and pre-connected and are represented by half discs ouside the superstate. Table 2.1 gives the connection and transition rules.

Source	Destination	Link	Location
state	entry	transition	outside
exit	state	transition	outside
entry	state	connection	inside
state	exit	connection	inside

Table 2.1: Connection and transition rules. The indicated location is relative to the superstate.

We define an alternative representation for Entry,Exit and Connection objects, which is simpler and may be useful in a graphical editor which would use scrolled frames. The connections are not represented and we stick a copy of the entry to the state. This stresses the fact that our entries and exits are connected. Anonymous entries may have the same representation than the not anonymous ones, but with a different color to show the difference and they should have no label. The alternative representation is depicted on fig. 2.2.

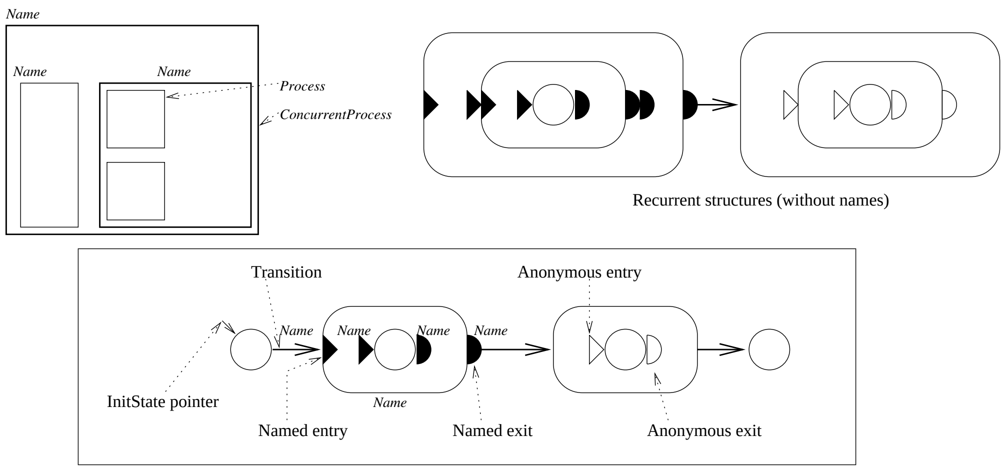


Figure 2.2: Alternative graphical notations

[2] Appropriate semantics is defined in the next sections.

Channel objects have a name and are represented by small tubes and are connected by thin lines to the ConcurrentProcess or Process using them (this is defined in their data interface). In the case of a ConcurrentProcess object, a channel defined in the data interface is drawn and connected to its container (the ConcurrentProcess owner) by a thick line.

An alternative solution for channels is not to represent them. The representation is optional, since channels are data objects and the graphical objects correspond to control structures.

CommitState and UrgentChannel objects differ from their ordinary peers with a small flash.

As a rule of thumb, a graphical editor for UL^{++} should edit component by component, hidding the content of a sub component, but showing its type. This is to encourage a hierarchical design with respect to the structure of UL^{++}.

2.2 Informal Textual Syntax

2.2.1 Composite Objects

As we have seen earlier, one extends the ConcurrentProcess, Process or SuperState class to define the objects of a system. It is allowed to extend existing extensions as well. These class definitions concern the control structure of the automaton and are composite classes, as precised earlier. The syntax is as follows :

```
class Name([data interface]) extends ParentClass {
  data:
     [//data objects declaration]
  control:
     [//control objects declaration]
}
```

Squared braces refer to optional arguments. The data interface is defined as follows :

```
DataObject_instanciable_class_name name1, ...
```

This field is optional and it is possible to have empty interfaces. The interface itself is a list of comma separated `ClassName instanceName` declarations. It is possible to have also `int` as a scalar type, which is a constant. This means that the class will be instanciated with an integer constant.

A composite class has two optional fields concerning the data definition and the control definition. We give in table 2.2 the allowed content per type of class. An extension of a class is a simple extension, there is no multiple inheritance. Note that the declaration is not optional (`data:` and `control:`), but the content is.

DataObject:

Class	Clock	Channel	StaticDataObject
ConcurrentProcess	x	x	x
Process	x		x
SuperState			x

ControlObject:

Class	InitState	StateConnective	LinkObject	StateObject	ProcessObject
ConcurrentProcess					x
Process	x		x	x	
SuperState		x	x	x	

Table 2.2: Composition rules for data and control objects.

The order of the declarations is important : one may not declare a transition between two states before declaring the states. There is no forward features in UL^{++} to oblige to have some structure and order in the declarations.

2.2.2 Data Object Declaration

The syntax of these declarations is straightforward :

```
ClassName name1[,name2][,name3][...];
```

This concerns Channel, UrgentChannel, Clock, Boolean and Integer classes. BoundedInteger classes have a constructor :

```
BoundedInteger(min,max) name1[,name2][...];
```

All names separated by a comma have the same type, defined as the class name and these names will refer to instances when the composite object is instanciated. Array definitions are particuliar :

```
ContentTypeClass[min,max] name1{,name2}{..};
```

The curled braces refer here to optional arguments and the squared braces to the dimension and their content to the dimension definition, from a minimum index to a maximum one. As precised before, Array objects are matrices of one simple type of objects, and with one dimension, for compatibility with Uppaal.

2.2.3 Control Object Declaration

InitState They are defined as follows :

```
InitState name;
```

There must always be one and only one InitState declaration per Process object. The name refers to a “virtual state” (see virtual state). The `name` has to be defined before.

Entry These objects are declared as follows :

```
Entry publicName1=stateName1[,..];
Entry stateName2[,..];
```

We can associate in the definition the public name of the entry and its connection inside the superstate : we have in this case a pre-connected entry. Otherwise, the name must be predeclared as a “virtual state” and it is then an anonymous entry . An entry is always expected to be the target of a transition.

Exit These objects are defined as follows :

```
Exit publicName1=stateName1[,..];
Exit stateName2[,..];
```

We can associate in the definition the public name of the exit and its connection inside the superstate : we have in this case a pre-connected exit. Otherwise, the name must be declared as a “virtual state” and it is then an anonymous exit (several ones per superstate). An exit is always expected to be the source of a transition.

Virtual state It is a view of entries and exits, or state and superstate. This term refers eventually to a state in the sense that an entry or an exit point will give a state or a superstate if we follow the connection(s). In the case of a superstate, we will find an anonymous entry point ... When a connection is defined, there is of course only one connected “virtual state”. The term “virtual state” is useful also, when considering that we define a transition to and from a “virtual state”, regardeless of its real nature, because we know that there is a state connected to.

Connective objects They make a bigger family than the two pre-connected entry and exit points that we have defined. We can imagine un-pre-connected entry and exit as well. As one can guess, these interface points require to be connected to a state (explicitely, and not with their declaration). Figure 2.3 describes all types of possible connectives. In UL^{++}, however, we focus on **pre-connected** entries and exit, in order to get this notion of "virtual state", which is not possible with their un-pre-connected peers.

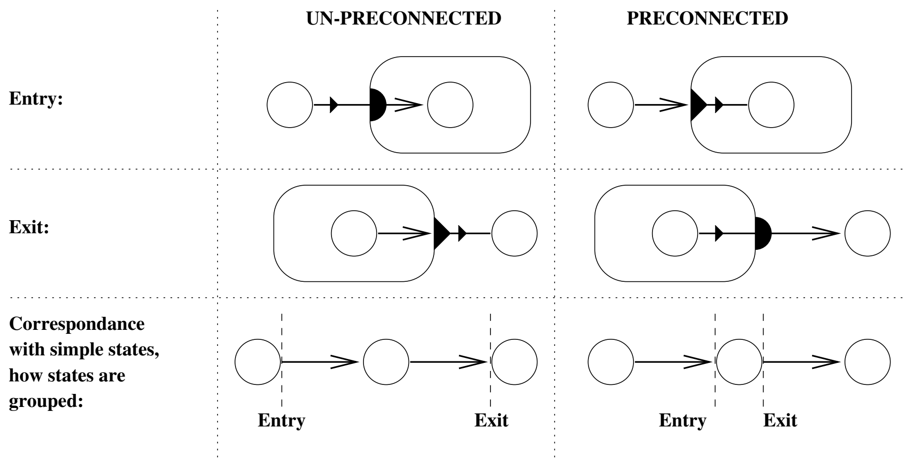


Figure 2.3: Connective types. Only the pre-connected ones are used in UL^{++}.

The choice of the different symbols are motivated in figure 2.4 where we can see the compatibility between connectives and their possible cascade combinations. When we instanciate our system, the connections are "stuck" together as we can see on the figure, and the symbol is then easy to understand and to use. Note that we have indicated in parenthesis the un-pre-connected alternatives, which is not used in UL^{++}, but the symbols are general.

Connection There is no way to define a connection independently from the control interface definition. As we focus on pre-connected entries and exits, they are always connected when they are declared. We can just note that a flat and instanciated model of UL^{++} results in a traditional automaton model.

Transition They are defined as follows :

```
Transition [name=][{condition}]source->destination[{assignments}][,..];
```

A transition may be named, which is optional. The `condition` term is a Condition object, defined as a conjunction of a guard expression and a channel synchronization. The guard expression uses Clock and StaticDataObject objects and their associated operators (see semantics and BNF) and the channel synchronization is of the form :

`name!` for a simple synchronization request, from the sender side.
`name!(`*list of StaticDataObject objects*`)` for a synchronization with message passing, from the sender side.
`name?` and `name?(`*list of StaticDataObject objects*`)` are the corresponding syntax for the other side (the receiving peer which synchronizes with the sender).

A condition could then be defined as `{(a>b) && c!}`. The assignments are an Assignments object. An assignment is done via the operator `:=` defined for the objects. An assignment is the only operation which modifies a static data object, hence their name.

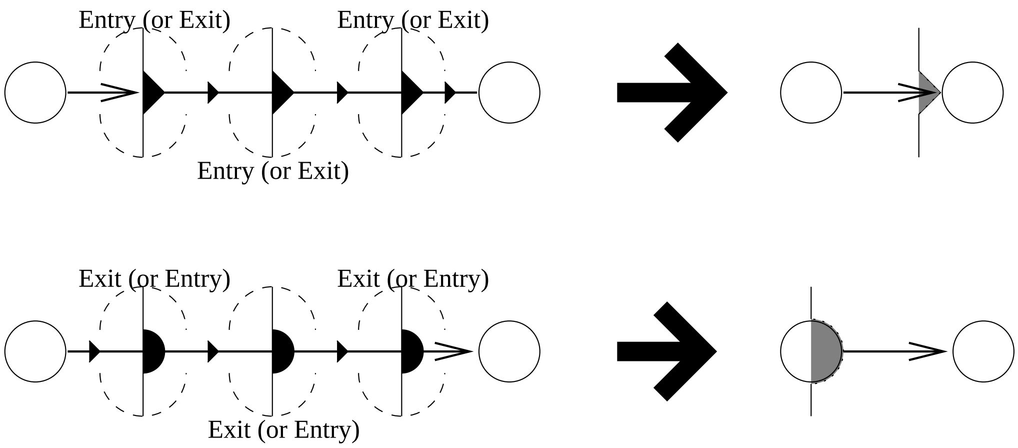


Figure 2.4: Connective cascade rules and tips about the compatibility rules. Un-pre-connected alternatives are in parenthesis.

State They are the basic constituants of our automaton. They are defined as follows :

```
State name[{invariant}][,..];
```

The invariant is an Invariant object, written as a boolean expression.

Extended classes They are the composite structures that the user defines to build the model. They are used as follows :

```
ExtensionName([interface arguments]) name1[,name2];
```

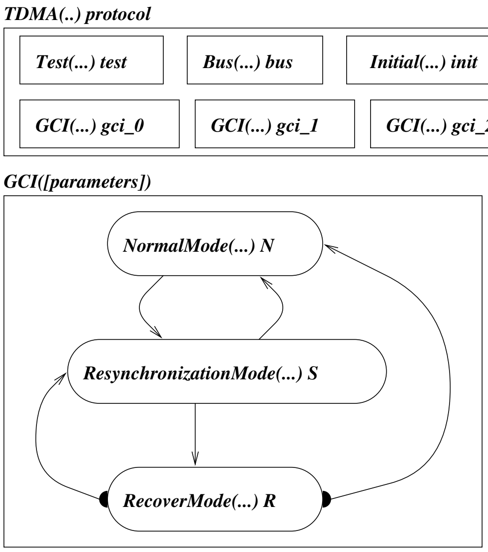
If the composite structure is a superstate, then one can add an invariant term, as for states. The `interface arguments` term corresponds to the required objects to instanciate the class `ExtensionName`. They are defined in the interface of `ExtensionName`, which is defined **earlier** by the user.

Before giving the formal syntax we would like to give a taste of the language, by using a sketch of what would be a UL^{++} model of the previous example DACAPO TDMA start-up protocol. The example is shown in figure 2.5

We distinguish the main system which is a concurrent process with the test, bus and init processes. We notice three stations which are instances of the same class model. Parameters are not given for sake of simplicity. We notice the three superstates in the station class model. Transitions are defined from/to superstates, with the level of abstraction that we wanted before (chapter one). We have also specific exit points, from the recover mode.

2.3 Formal Syntax : BNF

We use the symbol `::=` to define entities and the dot to concatenate characters and strings. '.' refers to the dot character. `A ... Z` refers to `A | B | C ... Z`, and similarly `a ... z`. These letters are with or without accents. In practice, we use the java functions `isLetter()` and `isDigit()` to identify letters and digits since Java deals with Unicode and in fact we have some more characters which are recognized as letters. ϵ refers to "empty". The *int* definition is not given and is an ordinary signed integer bounded between -2^{15} and 2^{15}. The notation `int` refers to a keyword, and words in this style refer to keywords as well. We notice that `$` is a valid character in the following definition, but has special semantics (see semantics). The defined operators have the precedance order `/ * - +`. There are rules to define data and control objects, depending on




```
class TDMA(..) extends
 ConcurrentProcess {
 data: ...
 control:
 Test(..) test;
 Bus(..) bus;
 Initial(..) init;
 GCI(..) gci_0;
 GCI(..) gci_1;
 GCI(..) gci_2;
}
class GCI(..) extends
 Process {
 data: ...
 control:
 NormalMode(..) N;
 ResynchronizationMode(.) S;
 RecoverMode(..) T;
}
class NormalMode(..)
 extends SuperState { .. }
```

Figure 2.5: Illustration of UL^{++} with DACAPO TDMA start-up protocol.

the nature of the composite object. This does not appear here, since it is only simple syntax description.

We define some shortcuts in the BNF representation :

- *B* ::= (*A*)* $\equiv$ *B* ::= ϵ $\mid$ *A B*
- *B* ::= (*A*)? $\equiv$ *B* ::= ϵ $\mid$ *A*
- *B* ::= (*A* $\mid$ *C*) *D* $\equiv$ *B* ::= *A D* $\mid$ *C D*

To make things clear, the *italic* characters refer to non terminals or shortcut constructions, whereas the `typewriter` charaters refer to terminals.

2.3.1 Expressions

Expressions are invariants, assignments and conditions. We define invariants as follows :

invariant	::=	*invComp* (`&&` *invComp*)*
invComp	::=	*name* (`<` $\mid$ `<=`) *sum* $\mid$ `(` *invComp* `)`
sum	::=	*term* ((`+` $\mid$ `-`) *term*)*
term	::=	*unary* ((`*` $\mid$ `/`) *unary*)*
unary	::=	`-` *element* $\mid$ *element*
element	::=	`int` $\mid$ *name* (`[` *sum* `]`)? $\mid$ `(` *sum* `)`
		$\mid$ `(` *comp* `?` *sum* `:` *sum* `)`
name	::=	*FirstChar* $\mid$ *FirstChar.EndName*
FirstChar	::=	`a` ... `z` $\mid$ `A` ... `Z`
EndName	::=	ϵ $\mid$ *EndChar.EndName*
EndChar	::=	*FirstChar* $\mid$ `$` $\mid$ `_` $\mid$ `0` ... `9`
comp	::=	`-` *comp* $\mid$ `(` *sum* (`<` $\mid$ `>` $\mid$ `<=` $\mid$ `>=` $\mid$ `==` $\mid$ `!=`) *sum* `)`
		$\mid$ `true` $\mid$ `false` $\mid$ *name* (`[` *sum* `]`)? $\mid$ `(` *comp* `)`

As we can notice, we define invariant as a conjunction of the form $x < f(..)$. Eventually, when we instanciate, the $f(..)$ should give a constant, or it would be an instanciation error. We define assignments as follows :

$$
\begin{array}{rcl}
\mathit{assignment} & ::= & \mathit{assign}\ (\ \mathit{assign}\ )* \\
\mathit{assign} & ::= & \mathit{name}\ (\texttt{[}\ \mathit{sum}\ \texttt{]})?\ \texttt{:=}\ (\mathit{comp} \mid \mathit{sum})\ \texttt{;}
\end{array}
$$

The assignment to *comp* or *sum* is dependant from the nature of *name*. We finish the expression definition with the condition definition :

$$
\begin{array}{rcl}
\mathit{condition} & ::= & \mathit{condTerm}\ (\ \texttt{\&\&}\ \mathit{condTerm}\ )* \\
\mathit{condTerm} & ::= & \mathit{channelExpr} \mid \mathit{comp} \\
\mathit{channelExpr} & ::= & \mathit{channelDecl}\ (\texttt{(}\ \mathit{parameter}\ (\ \texttt{,}\ \mathit{parameter}\ )*\ \texttt{)})? \\
\mathit{parameter} & ::= & \texttt{true} \mid \texttt{false} \mid \mathit{sum} \mid \mathit{name} \\
\mathit{channelDecl} & ::= & \mathit{name}\ (\ \texttt{?} \mid \texttt{!}\ )
\end{array}
$$

As we can notice, we use optional parameters in the channel expression. We use value passing with synchronization and the parameters represent what we want to send. However, the receiver may have only a name as a recipient, not a formula or `true` or `false`.

2.3.2 Class Declaration

A class has, as precised earlier, data and control members. We find these two fields in the body of a class. The data interface is defined in parenthesis just after the name of the class, and the parameters are typed. In the body come the different member specific declarations.

$$
\begin{array}{rcl}
\mathit{classDef} & ::= & \texttt{class}\ \mathit{name}\ \texttt{(}\ (\ \mathit{dataDeclList}\ )?\ \texttt{)}\ \ \texttt{extends}\ \mathit{name}\ \{\ \mathit{classBody}\ \} \\
\mathit{dataDeclList} & ::= & \mathit{intOrDataDeclaration}\ (\ \texttt{,}\ \mathit{intOrDataDeclaration}\ )* \\
\mathit{intOrDataDeclaration} & ::= & \texttt{int}\ \mathit{name} \mid \mathit{dataDeclaration} \\
\mathit{dataDeclaration} & ::= & \texttt{Clock}\ \mathit{name} \mid \texttt{Channel}\ \mathit{name} \mid \texttt{UrgentChannel}\ \mathit{name} \\
 & & \mid \mathit{simpleStaticDataType}\ \texttt{[}\ \mathit{constant}\ \texttt{,}\ \mathit{constant}\ \texttt{]}\ \mathit{name} \\
 & & \mid \mathit{simpleStaticDataType}\ \mathit{name} \\
\mathit{simpleStaticDataType} & ::= & \texttt{Boolean} \mid \texttt{Integer} \mid \texttt{BoundedInteger}\ \texttt{(}\ \mathit{constant}\ \texttt{,}\ \mathit{constant}\ \texttt{)} \\
 & & \\
\mathit{classBody} & ::= & \texttt{data:}\ (\ \mathit{dataDeclaration}\texttt{;}\ )*\ \texttt{control:}\ (\ \mathit{controlDeclaration}\texttt{;}\ )* \\
\mathit{controlDeclaration} & ::= & \texttt{State}\ \mathit{stateName}\ (\ \texttt{,}\ \mathit{stateName}\ )* \\
 & & \mid \texttt{CommitState}\ \mathit{stateName}\ (\ \texttt{,}\ \mathit{stateName}\ )* \\
 & & \mid \texttt{Transition}\ \mathit{transitionName}\ (\ \texttt{,}\ \mathit{transitionName}\ )* \\
 & & \mid \texttt{Entry}\ \mathit{connectName}\ (\ \texttt{,}\ \mathit{connectName}\ )* \\
 & & \mid \texttt{Exit}\ \mathit{connectName}\ (\ \texttt{,}\ \mathit{connectName}\ )* \\
 & & \mid \texttt{InitState}\ \mathit{extName} \\
 & & \mid \mathit{name}\ \texttt{(}\ (\mathit{argList})?\ \texttt{)}\ \mathit{stateName}\ (\ \texttt{,}\ \mathit{stateName}\ )* \\
\mathit{argList} & ::= & \mathit{sum}\ (\ \texttt{,}\ \mathit{sum}\ )* \\
\mathit{stateName} & ::= & \mathit{name}\ (\ \{\ \mathit{invariant}\ \}\ )? \\
\mathit{transitionName} & ::= & (\mathit{name}\ \texttt{=})?\ (\{\mathit{condition}\})?\ \mathit{extName}\ \texttt{->}\ \mathit{extName}\ (\{\mathit{assignment}\})? \\
\mathit{connectName} & ::= & (\mathit{name}\ \texttt{=})?\ \mathit{extName} \\
\mathit{extName} & ::= & \mathit{name.name} \mid \mathit{name}
\end{array}
$$

2.3.3 File Definition

To conclude with the syntax, we define the format of a file containing UL^{++} declarations. It consists of importation definitions, class definitions and one (optional) system declaration.

$$
\begin{array}{rcl}
\mathit{file} & ::= & (\ \mathit{importDef} \mid \mathit{classDef}\ )*\ (\ \mathit{name}\ \texttt{(}\ (\mathit{argList})?\ \texttt{)}\ \mathit{stateName}\texttt{;}\ )? \\
\mathit{importDef} & ::= & \texttt{import}\ (\ \mathit{name} \mid \texttt{*}\ )\ \texttt{from}\ \mathit{extName}\texttt{;}
\end{array}
$$

Chapter 3

Semantics

We give first an informal semantics to explain what our language means, and what it does. Then we give an operational semantics to express the formal behaviour of a system modeled with UL^{++}.

3.1 Informal Semantics

3.1.1 Classes and inheritance

Classes Classes in UL^{++} are used only to model a system and there is no user's defined data class at the moment. They are complex structures with two fields, to separate the data (`data:`) from the control (`control:`) declarations. There is no method declaration. The access rights are always "protected" for the classes, which means that a class can not address the content of another class. The content is not visible. When a class extends another one (we call it extension), it has access to its parent content, as if it were its content. When one extends a class, one takes the whole class with all its content and one can add or modify the content.

When we deal with inheritance, overriding and overloading issues are then addressed. Overriding consists of redefining an entity and overloading consists of adding new behaviours to this entity.

Data members We allow overriding and the former data variables are shadowed. This means that the objects coming from the parent (which address the variable), address the original one, and the new objects addressing it in the extension, address the new one.

Control members Transitions may be overloaded, in the sense that we can add transitions from a source to a destination, though there was already one (or possibly more) defined. However, if they are named, different names are to be given. One may override a named and predefined transition at the condition that one keeps the same source and the same destination. One may override only the condition and the actions of a transition.

Connections are final and so are entries and exits, which is logical regarding to transitions because a connection has no condition nor action to be overriden. Overloading a connection gives inconsistency and has no meaning in UL^{++} : a control interface has to be connected to a real state and only one, since it represents, in a sense, a virtual state. No overloading nor overriding for connections (entries and exits).

A state may be overriden into a superstate or a state. Overriding a state by a state has sense only if the invariant is overloaded. This means that it is possible to add an invariant, and the new one is a conjunction of the old and the added one. We use this conjunction instead of overriding because the invariant is considered as a safety property of the state and overriding a state should not destroy this safety property. So the overriding definition for a state is special.

When a superstate overrides a state, anonymous entries and exits are automatically used, for compatibility reasons with the previous state. A superstate may be overriden, but only by itself or by one of its extension, and rules with invariant are the same as for states. Overriding by an extension is the only way to garantee some consistency with the new definition, in particular for the interface definition which must be the same.

The restriction on overriding a state by a superstate (anonymous interface, which can be extended however by a named one) or overriding a superstate by one extension is to respect "Liskov Substitution Principle" which says that a subclass must be freely substitutable for its superclass. Then we apply this on an overriden state (or superstate). The system from the parent component's point of view looks the same.

Validity An object is valid, at the instanciation, if it respects the composition rules given in figure 1.9. For example an empty SuperState object with an entry connected to an exit is not possible.

3.1.2 Interfaces

Data Interface When defining a class, the variables defined in the data interface are used as if they were part of the data body definition. For this reason, they must have unique names inside the class.

The semantics of this interface comes when the class is instanciated. It is a substitution semantics. The class is instanciated with the appropriate arguments corresponding to the interface with the same types, and they may be only data objects. Then the variables in the interface are substituted by the argument names, with their absolute name definition, to avoid name conflict.

The absolute name of a variable is defined as `container.container ... variable`. However there may be an ambiguity when refering to a shadowed variable. In this case, the absolute name carries the class name information : `className$variable`. The same mechanism is used when one wants to address a shadowed variable when defining an extension class.

Concerning inheritance, an interface is inherited from the parent class and may not be overriden. It may be overloaded only and the interface which is defined in a class declaration is only the extension part, namely, the interface which will be used when intanciating is the union of the parent and the child interfaces, with the parent members first.

Control Interface This interface is composed of entry and exit definitions, which may be anonymous or pre-connected and named. The semantics of entries and exits is a "public relay" semantics. When a class is instanciated, transitions from, or to, such a relay, will ask the relay for the corresponding state and the relay will return the absolute named state. The semantics is the same when using cascade of compatible relays : if a relay is connected to another relay, then it returns the result of its connected relay's call. Figure 2.4 shows how the connections are resolved, in the general case. However, in UL^{++}, we focus only on pre-connected entries and exits.

3.1.3 Control Objects

Connection The connection contains the information needed by the "relays" to find the required "real" state. Connections are oriented because they take place between a state and a transition, or a transition and a state, but always in the transition direction.

InitState This object indicates the initial control node of a process. The system is in initial state at the beginning of the verification (or simulation) and each process has its initial control node "active". The InitState object is not a transition, it is rather a pointer. Overriding an InitState declaration is allowed, it is just to redeclare the new initial state.

State and SuperState Our modeled system has semantic states, which are formally of the form $(\bar{l}, v)$ where $\bar{l}$ is a control vector indicating the current control node and v is an assignment giving the current value for each Clock and StaticDataObject variable (in the scope). The control node may be a State object, and then $\bar{l}$ refers to this State object, or it can be a SuperState object, and it refers then to another $(\bar{l}_s, v_s)$ substate where $\bar{l}_s$ refers to the current sub-control node and v_s refers to the assignment v extended by local variable definitions.

Process and ConcurrentProcess A Process object corresponds to a process in a real time system. It has clocks attached to it. A ConcurrentProcess object is a group of processes (which are concurrent) or sub-groups of processes. It has clocks as well and channels for inter-process communication and synchronization. Data objects are considered shared variables between the processes (if defined in a ConcurrentProcess object).

Transition A Transition object represents an action transition of a system, between control nodes (state or superstate). It is enabled if the condition term, associated with, is true. When a transition is fired (see Channel), the actions associated with are executed, leading to a new state, after the execution.

In UL^{++}, transitions are defined between "virtual states" because they may be directed to a control interface. However, this is resolved to a state when instanciating, so one could think that a transition is always between two states. It is almost true. In fact, if one pays attention to the superstate semantics, a transition (after instanciation) may be defined from, or to, a superstate. If we consider the sequence : *state*(**A**)->*superstate*(**S**)->*state*(**B**), then the transition **S->B** can be done if it exits a transition path between the entry of **S** and the exit of **S** and the transition **S->B** is enabled. However, before firing **S->B**, time may progress because all internal transitions of **S** must be done. The transition **A->S** can be seen at two levels : a transition to the superstate, or to the hidden control node, nested inside the superstate.

CommitState They are special State objects with the constraint that they must be left immediately. This means that the next state of the system will have the current "visited" commit state node left. If this is not possible, then we have a deadlock.

3.1.4 Data Objects

Channel A Channel object is a synchronization mean between processes. It may carry information. We call it communication channel. It is used in the condition term of a transition. The channel term returns true if the synchronization is possible, false otherwise. The synchronization works by pairs, which means that there is a sender, which asks for a receiver peer to synchronize with. If there are several senders and receivers on the channel, pairs may be created non deterministically. When a synchronized transition "fires", both synchronized transitions fire at the same time. The pairs are possible only with channels of the same *signature*, which means that want to synchronize with the same value passing signature. The value passing is atomic when the transition is fired and the peer which does a receive synchronization has its variables updated.

UrgentChannel In a state where processes may synchronize on an urgent channel, no further delay is allowed. The corresponding transitions must be fired.

Clock A clock in a process simulates time and is incremented in delay transitions, when no action transition is taken. This is allowed as long as none of the invariants of the current control nodes are violated. A Clock object has the following operators :

- `:=` assignment to a positive `int` constant.
- `> < >= <= == !=` comparisons to `int`, Clock or Integer, return a Boolean object.

BoundedInteger A BoundedInteger is an encapsulation of a bounded integer number. It has a name, which can be useful in the verification process to manipulate it symbolically. It has the same operators as Clock objects plus :

- `+ - * /` operators to `int` or BoundedInteger, return a BoundedInteger object.
- `:=` assignment to a BoundedInteger.

If a bounded integer is asked to take a value out of its allowed range, it is a system error.

Integer An Integer object is a bounded integer between -2^{15} and 2^{15}. It is a sub-type of BoundedInteger.

Boolean This is a traditional boolean, which can be true or false. It has a name, like BoundedInteger and Integer. It can be seen as BoundedInteger(0,1), but a boolean is not an integer. This view is only to give some suggestion about the implementation or the translation to UPPAAL.

Array They are matrices of Integer, BoundedInteger or Boolean objects of one dimension (for compatibility with UPPAAL). It is in fact a simple vector, but we use the term matrix, because this could be generalized in the future to any dimension.

3.1.5 Importation

An `import` command in a UL^{++} file has the effect to load the requested objects at the location of the `import` in the file. We add some more semantics : the loaded object has an identity *file.object*. When an object has to be loaded, UL^{++} checks if it is already loaded or not, to avoid to load it twice. However, if a conflict appears, namely *file1.object* and *file2.object* are requested, there is an error, because they are two different definitions of the same object. When the special character `*` is used, this is applied for every object in the requested file.

3.2 Operational Semantics

3.2.1 Abstract Syntax

We define an abstract syntax to describe in a compact manner our control structure hierarchy. `S` refers to a general state, which can be a superstate or a simple node (`N`). The arrow $\xrightarrow{g,\alpha,r}$ represents a syntactic transition from a state to another, decorated with a guard, an action and a reset. `P` refers to a concurrent process or a process. `P` $\|$ `P` refers to two processes in parallel. The definitions are :

$\mathtt{S} ::= \mathtt{S} \xrightarrow{g,\alpha,r} \mathtt{S} \mid \mathtt{N}$

$\mathtt{P} ::= \mathtt{S} \mid \mathtt{P} \parallel \mathtt{P}$

We define also : for $l \in S$, S_l is the set of (general) states of l, $\triangleright S_l$ the set of entry states of l and $S_l\triangleright$ the set of exit states of l. Furthermore we make a reasonable assumption to have a consistent model : for all superstates l, there exists a path through the nested components such that there is always a node inside, which can be written as :

$\forall l \in S, l \notin N, S_l \neq \varnothing, \forall l' \in S_l, l' \notin N, \exists l_1, l_2 \ldots l_k, k > 1,$

$l_1 \in N, l_2 \in S_{l_2}, \ldots l_{k-1} \in S_{l_k}, l_k = l'$

We make two other assumptions which are : for any transition to a superstate, an entry exists, and for any transition from a superstate, an exit exists. We can write this as :

$\forall l' \xrightarrow{g,\alpha,r} l, l \in S, l \notin N, l' \in S, S_l\triangleright \neq \varnothing$

$\forall l \xrightarrow{g,\alpha,r} l', l \in S, l \notin N, l' \in S, \triangleright S_l \neq \varnothing$

We define similar notions at the process level. We define $\mathcal{L}_i$ as a hierarchical control vector :
$\mathcal{L}_i = \mathcal{L}'_1||\mathcal{L}'_2||\dots||\mathcal{L}'_n$
or $\mathcal{L}_i = l_i$
Similarly, we define $\mathcal{V}_i$ as the hierarchical "memory" with
$\mathcal{V}_i = \mathcal{V}'_1||\mathcal{V}'_2||\dots||\mathcal{V}'_n$
or $\mathcal{V}_i = v_i$
These definition aim at introducing hierarchy at the process level and manipulating a process in an abstract way, in the sense that a process could be a set of processes.

3.2.2 Transition Rules at the State Level

Action transitions We distinguish four kinds of transitions, which are illustrated in figure 3.1. The transition node to node is the simplest one and it is the "low level" transition. Node to superstate or superstate to node are similar and they concern the entry/exit points of a superstate. These transitions are useful to define a transition from a superstate or to a superstate. We notice that in these cases, the node could be a simple node or a general state, as expressed in the rules. The last rule expresses the fact that a transition is inside a superstate. We stay in the same superstate, but there is an action, it is not a wait transition. These four rules are given in figure 3.2.

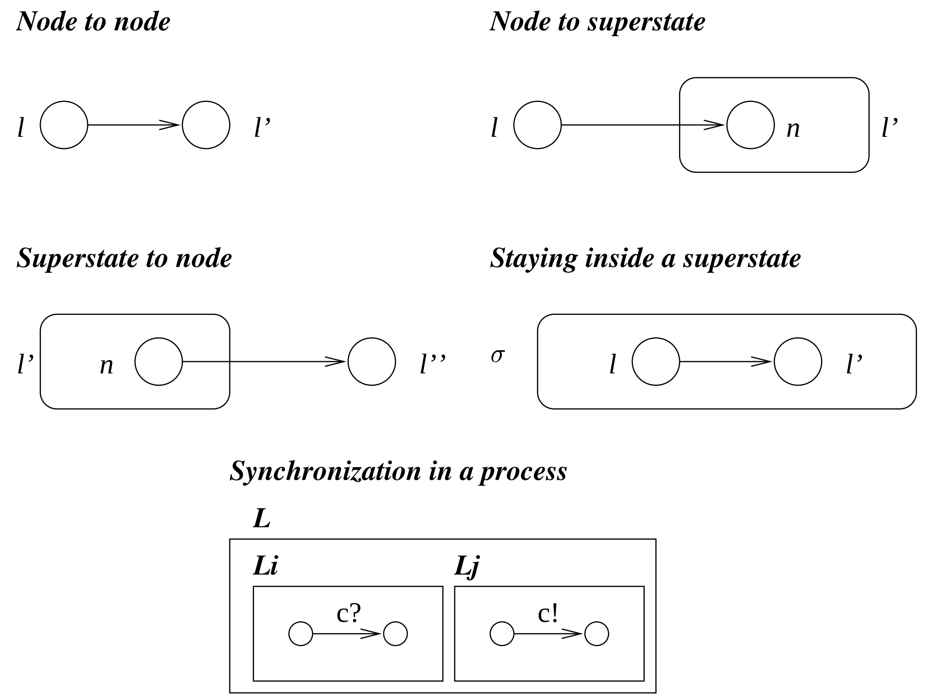


Figure 3.1: Illustration of the inductive rules.

Delay transitions We distinguish two kinds of delay transitions : one at the low level, namely the simple node level, and the other one at the superstate level. A delay transition may occur in a state if the invariant is satisfied. We repeat the same at the superstate level adding that the superstate invariant has to be satisfied as well. This construction aims at setting a global invariant for a set of states, i.e. the superstate.

We give the rules for these two delay transitions in figure 3.3.

The semantics enforces the invariants of a set of states (general states). If we consider SS_d, the delay transition occurs if the "super invariant" $Inv(l)(v + d)$ is satisfied and if the delay

$$NN_\alpha : \frac{l \xrightarrow{g,\alpha,r} l' \quad g(v) \quad v' = \gamma(v) \quad l, l' \in N}{(l, v) \xrightarrow{\alpha} (l', v')}$$

$$NS_\alpha : \frac{n \in \triangleright S_{l'} \quad (l, v) \xrightarrow{\alpha} (n, v') \quad l \notin S_{l'} \quad l, l' \in S}{(l, v) \xrightarrow{\alpha} (l', v')}$$

$$SN_\alpha : \frac{n \in S_{l'} \triangleright \quad (n, v) \xrightarrow{\alpha} (l'', v'') \quad l'' \notin S_{l'} \quad l', l'' \in S}{(l', v) \xrightarrow{\alpha} (l'', v'')}$$

$$SS_\alpha : \frac{(l, v) \xrightarrow{\alpha} (l', v') \quad l, l' \in S_\sigma}{(\sigma, v) \xrightarrow{\alpha} (\sigma, v')}$$

Figure 3.2: Transition rules : NN_α stands for from a node to another node ; NS_α stands for from a state to a superstate ; SN_α stands for from a superstate to a state ; SS_α stands for a transition inside a superstate. α is used to denote that they are action transition rules.

$$NN_d : \frac{Inv(l)(v + d) \quad l \in N}{(l, v) \xrightarrow{d} (l, v + d)}$$

$$SS_d : \frac{n \in S_l \quad Inv(l)(v + d) \quad (n, v_n) \xrightarrow{d} (n, v_n + d)}{(l, v) \xrightarrow{d} (l, v + d)}$$

Figure 3.3: Delay transition for a node (simple state) and a superstate. The d stands for delay.

transition at the state level is enabled, namely if $Inv(n)(v + d)$ is satified. The condition we get is a conjunction of invariants at the node level. We use this result when we flatten our system, to translate to UPPAAL for example. We prove this result in figure 3.4.

$$\dfrac{n \in S_l \quad Inv(l)(v + d) \quad \dfrac{Inv(n)(v_n + d) \quad n \in N}{(n, v_n) \xrightarrow{d} (n, v_n + d)}\, NN_d}{(l, v) \xrightarrow{d} (l, v + d)}\, SS_d$$

$$\Downarrow$$

$$\frac{n \in S_l \quad Inv(l)(v + d) \quad Inv(n)(v_n + d) \quad n \in N}{(l, v) \xrightarrow{d} (l, v + d)}\, NN_d \wedge SS_d$$

Figure 3.4: Invariant deduction

This shows the invariant deduction for one level, but it is obvious to repeat the process for nested states to obtain the conjunction of the different invariants at each level.

3.2.3 Transition Rules at the Process Level

The action transition at the process is like the ordinary one, but with hierarchical control vectors. The delay transition is similar. The synchronization transition is done with a pair of *send*! and *receive*? actions, with hierarchical control vectors. The rules are given in figure 3.5.

$$PP_\alpha : \frac{(\mathcal{L}_i, \mathcal{V}_i) \xrightarrow{\alpha} (\mathcal{L}'_i, \mathcal{V}'_i)}{(\mathcal{L}, \mathcal{V}) \xrightarrow{\alpha} (\mathcal{L}^{[\mathcal{L}'_i/\mathcal{L}_i]}, \mathcal{V}^{[\mathcal{V}'_i/\mathcal{V}_i]})}$$

$$PP_d : \frac{\forall i \ : \ (\mathcal{L}_i, \mathcal{V}_i) \xrightarrow{d} (\mathcal{L}_i, \mathcal{V}_i + d)}{(\mathcal{L}, \mathcal{V}) \xrightarrow{d} (\mathcal{L}, \mathcal{V} + d)}$$

$$PP_s : \frac{(\mathcal{L}_i, \mathcal{V}_i) \xrightarrow{c!} (\mathcal{L}'_i, \mathcal{V}'_i) \quad (\mathcal{L}_j, \mathcal{V}_j) \xrightarrow{c?} (\mathcal{L}'_j, \mathcal{V}'_j)}{(\mathcal{L}, \mathcal{V}) \xrightarrow{\tau} (\mathcal{L}^{[\mathcal{L}'_i/\mathcal{L}_i, \mathcal{L}'_j/\mathcal{L}_j]}, \mathcal{V}^{[\mathcal{V}'_i/\mathcal{V}_i, \mathcal{V}'_j/\mathcal{V}_j]})}$$

Figure 3.5: Transition at the process level : α means action ; d means delay ; s means synchronization.

3.2.4 Example

We use the rules with a toy example described in fig 3.6, which is a nuclear plant with some hypothetical control system communicating with the plant with channels. The goal is to prove that transitions are possible providing the fact that the guards are verified and that we are in the appropriate state.

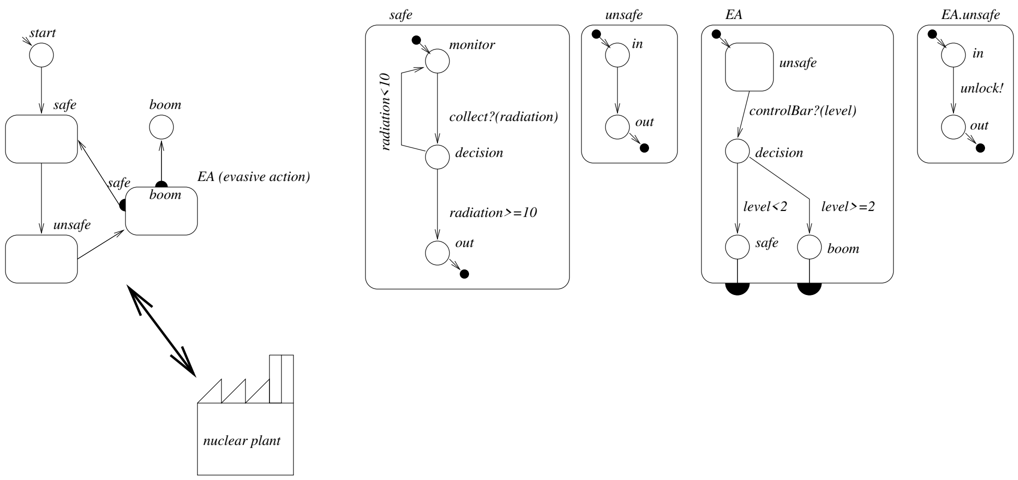


Figure 3.6: Example to illustrate the usage of the inductive rules

To make our proofs, we use several shortcuts which are :

- $s.l$ denotes a state l in the superstate s, and by definition, $l \in s$, so we do not write it twice. This is valid for several nested structures such as $s_1.s_2.l$ for example.
- we use the special notation $(a, v) \xrightarrow{\alpha} (\triangleright S_l, v')$ when $\exists n \in \triangleright S_l, (a, v) \xrightarrow{\alpha} (n, v')$ is true. We use the same equivalent notation with $(S_l \triangleright, v) \xrightarrow{\alpha} (a, v')$.

Transition of the type $\mathtt{S} \xrightarrow{\alpha} \mathtt{S}$ with safe $\xrightarrow{\alpha}$ unsafe :

$$\frac{safe.out \in S_{safe} \triangleright \quad \dfrac{safe.out \xrightarrow{g,\alpha,r} unsafe.in \quad g(v) \quad v' = v}{(safe.out, v) \xrightarrow{\alpha} (unsafe.in, v')} NN \quad unsafe.in \notin S_{safe}}{(safe, v) \xrightarrow{\alpha} (unsafe.in, v')} SN$$

$$\frac{unsafe.in \in \triangleright S_{unsafe} \quad (safe,v) \xrightarrow{\alpha} (unsafe.in,v') \quad safe \notin \triangleright S_{unsafe}}{(safe,v) \xrightarrow{\alpha} (unsafe,v')} NS$$

Transition of the type S $\xrightarrow{\alpha}$ N with EA $\xrightarrow{\alpha}$ boom :

$$\frac{EA.boom \in S_{EA} \triangleright \quad \dfrac{EA.boom \xrightarrow{g,\alpha,r} boom \quad g(v) \quad v' = v}{(EA.boom,v) \xrightarrow{\alpha} (boom,v')} NN \quad boom \notin S_{EA}}{(EA,v) \xrightarrow{\alpha} (boom,v')} SN$$

Transition of the type N $\xrightarrow{\alpha}$ S with start $\xrightarrow{\alpha}$ safe :

$$\frac{safe.monitor \in \triangleright S_{safe} \quad \dfrac{start \xrightarrow{g,\alpha,r} safe.monitor \quad g \quad v' = v}{(start,v) \xrightarrow{\alpha} (safe.monitor,v')} NN \quad start \notin safe}{(start,v) \xrightarrow{\alpha} (safe,v')} NS$$

Transition of the type N $\xrightarrow{\alpha}$ N with safe.decision $\xrightarrow{\alpha}$ safe.monitor :

$$\frac{\dfrac{safe.decision \xrightarrow{g,\alpha,r} safe.monitor \quad g(v) = (radiation < 10) \quad v' = v}{safe.decision,v) \xrightarrow{\alpha} (safe.monitor,v')} NN}{(safe,v) \xrightarrow{\alpha} (safe,v')} SS$$

Transition of the type S $\xrightarrow{\alpha}$ S with unsafe $\xrightarrow{\alpha}$ EA, with a nested superstate :

$$\frac{unsafe.out \xrightarrow{g,\alpha,r} EA.unsafe.in \quad g(v) \quad v' = v}{(unsafe.out,v) \xrightarrow{\alpha} (EA.unsafe.in,v')} NN$$

$$\frac{unsafe.out \in S_{unsafe} \triangleright \quad (unsafe.out,v) \xrightarrow{\alpha} (EA.unsafe.in,v') \quad EA.unsafe.in \notin unsafe}{(unsafe,v) \xrightarrow{\alpha} (EA.unsafe.in,v')} NS$$

$$\frac{EA.unsafe.in \in \triangleright S_{EA.unsafe} \quad (unsafe,v) \xrightarrow{\alpha} (EA.unsafe.in,v') \quad unsafe \notin EA.unsafe}{(unsafe,v) \xrightarrow{\alpha} (EA.unsafe,v')} NS$$

$$\frac{EA.unsafe \in \triangleright S_{EA} \quad (unsafe,v) \xrightarrow{\alpha} (EA.unsafe,v') \quad unsafe \notin EA}{(unsafe.v) \xrightarrow{\alpha} (EA,v')} NS$$

The generalization to any depth is possible because of our assumption which says that we will find eventually a state in any superstate, and when a transition is defined to or from a superstate, a corresponding entry or exit is defined.

Chapter 4

Examples

In this chapter, we aim to give some examples of UL^{++} to illustrate how the language works. The examples have been created with **VisualUL**, the program which implements UL^{++}. We will give UL^{++} representation, the instance representation (called ULi) and the translation to `ta` format[1].

4.1 Fischer Protocol

We model Fischer's mutual exclusion protocol with three processes. This example illustrates the use of the data interface with a well known example. Figure 4.1 shows the overview of the protocol with the process declarations and figure 4.2 shows the Fischer process definition.

We use in the data interface `(delay+1)` in order to test complex instanciation parameters. The UL^{++} source is as follows :

```
// UL++ file

class FischerProcess(Integer id,int i,int c) extends Process {
data:
 Clock x;
control:
 State A;
 State B{(x<=c)};
 State Wait;
 State CS;
 Transition exit=CS->A{id:=0; x:=0;};
 Transition {(id==0)}A->B{x:=0;};
 Transition try={(x<=c)}B->Wait{id:=i; x:=0;};
 Transition recover={(id==0)}Wait->B{x:=0;};
 Transition enter={(x>c) && (id==i)}Wait->CS;
 InitState A;
}

class FischerProtocol(int delay) extends ConcurrentProcess {
data:
 Integer ID;
control:
 FischerProcess(ID,1,(delay+1)) fischer1;
 FischerProcess(ID,2,(delay+1)) fischer2;
 FischerProcess(ID,3,(delay+1)) fischer3;
}
```

[1] the current format of UPPAAL language

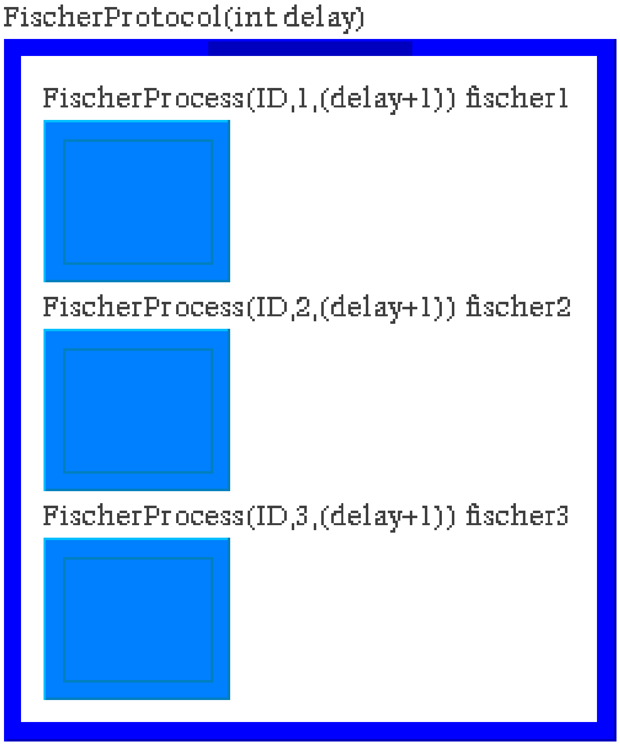


Figure 4.1: Fischer's protocol overview

```
//Instance declaration
FischerProtocol(6) fischer;
```

This source has been produced by the tool, including the comments. As we can see, there is one fischer process class model for the processes and one fischer protocol class model for the system. The instanciation is straight forward :

```
//UL++ instance

ConcurrentProcess fischer {
data:
  Integer fischer.ID;
control:
  Process fischer1 {
  data:
    Clock fischer.fischer1.x;
  control:
    State A;
    State B{(fischer.fischer1.x<=7)};
    State Wait;
    State CS;
    Transition exit=CS->A{fischer.ID:=0; fischer.fischer1.x:=0;};
    Transition {(fischer.ID==0)}A->B{fischer.fischer1.x:=0;};
    Transition try={(fischer.fischer1.x<=7)}B->Wait{fischer.ID:=1; fischer.fischer1.x:=0;};
    Transition recover={(fischer.ID==0)}Wait->B{fischer.fischer1.x:=0;};
    Transition enter={(fischer.fischer1.x>7) && (fischer.ID==1)}Wait->CS;
    IniState A;
  }
```

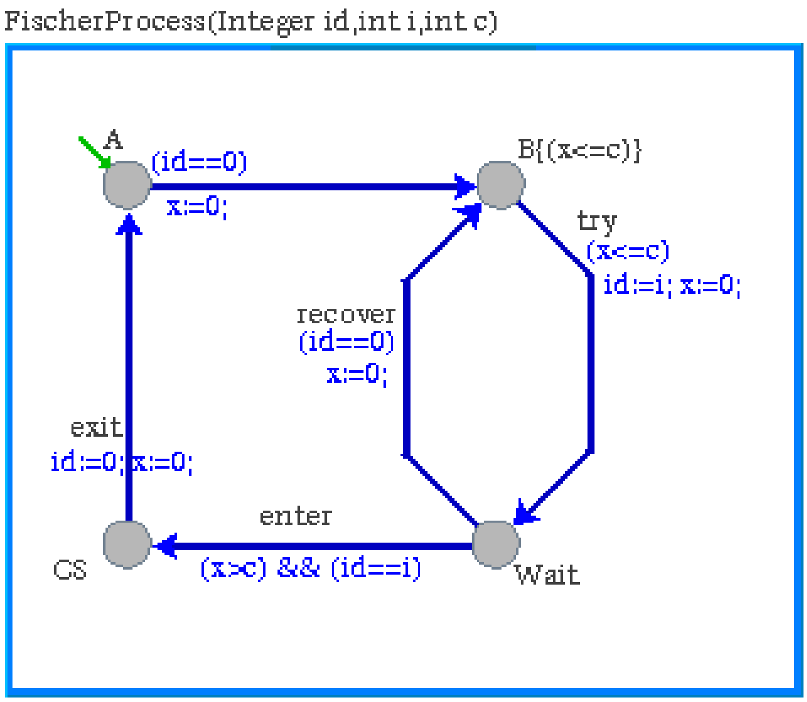


Figure 4.2: Fischer's protocol process

```
Process fischer2 {
data:
  Clock fischer.fischer2.x;
control:
  State A;
  State B{(fischer.fischer2.x<=7)};
  State Wait;
  State CS;
  Transition exit=CS->A{fischer.ID:=0; fischer.fischer2.x:=0;};
  Transition {(fischer.ID==0)}A->B{fischer.fischer2.x:=0;};
  Transition try={(fischer.fischer2.x<=7)}B->Wait{fischer.ID:=2; fischer.fischer2.x:=0;};
  Transition recover={(fischer.ID==0)}Wait->B{fischer.fischer2.x:=0;};
  Transition enter={(fischer.fischer2.x>7) && (fischer.ID==2)}Wait->CS;
  IniState A;
}
Process fischer3 {
data:
  Clock fischer.fischer3.x;
control:
  State A;
  State B{(fischer.fischer3.x<=7)};
  State Wait;
  State CS;
  Transition exit=CS->A{fischer.ID:=0; fischer.fischer3.x:=0;};
  Transition {(fischer.ID==0)}A->B{fischer.fischer3.x:=0;};
  Transition try={(fischer.fischer3.x<=7)}B->Wait{fischer.ID:=3; fischer.fischer3.x:=0;};
  Transition recover={(fischer.ID==0)}Wait->B{fischer.fischer3.x:=0;};
  Transition enter={(fischer.fischer3.x>7) && (fischer.ID==3)}Wait->CS;
  IniState A;
}
```

```
}
```

As we can see in the instance, all the names have been substituted. This source has been obtained with the short name option, which means that the control objects have the local name printed. The numerical parameters have been computed and have replaced the symbolic definition. This source has been produced by the tool as well.

This has been translated to `ta` format as follows :

```
int ID;

process fischer1 {
  clock x;
  state A,B{x<=7},Wait,CS;
  init A;
  trans CS -> A {
          assign ID:=0, x:=0;
        },
        A -> B {
          guard ID==0;
          assign x:=0;
        },
        B -> Wait {
          guard x<=7;
          assign ID:=1, x:=0;
        },
        Wait -> B {
          guard ID==0;
          assign x:=0;
        },
        Wait -> CS {
          guard x>7,ID==1;
        };
}

process fischer2 {
  clock x;
  state A,B{x<=7},Wait,CS;
  init A;
  trans CS -> A {
          assign ID:=0, x:=0;
        },
        A -> B {
          guard ID==0;
          assign x:=0;
        },
        B -> Wait {
          guard x<=7;
          assign ID:=2, x:=0;
        },
        Wait -> B {
          guard ID==0;
          assign x:=0;
        },
        Wait -> CS {
          guard x>7,ID==2;
        };
}

process fischer3 {
```

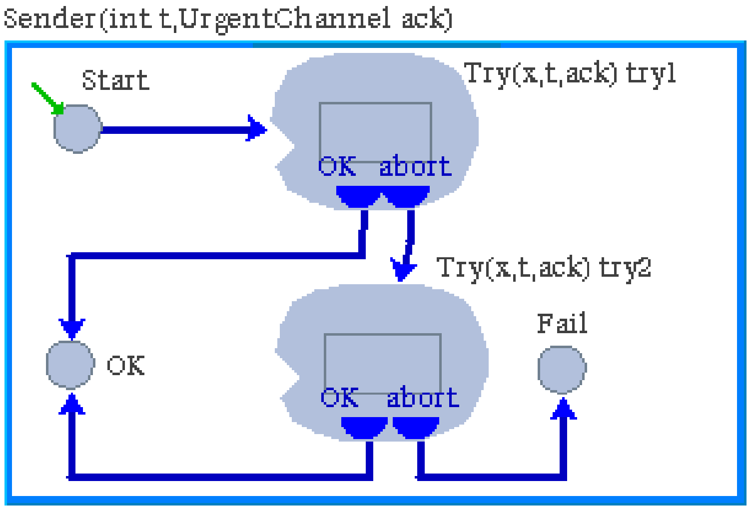


Figure 4.3: The sender process.

```
  clock x;
  state A,B{x<=7},Wait,CS;
  init A;
  trans CS -> A {
          assign ID:=0, x:=0;
        },
        A -> B {
          guard ID==0;
          assign x:=0;
        },
        B -> Wait {
          guard x<=7;
          assign ID:=3, x:=0;
        },
        Wait -> B {
          guard ID==0;
          assign x:=0;
        },
        Wait -> CS {
          guard x>7,ID==3;
        };
}

system fischer1,fischer2,fischer3;
```

We recognize the three processes and we notice the shared variable which belonged to the concurrent process in UL^{++}. In this format, the full names have been shortened by the tool in order to give a representation as simple as possible.

4.2 A Simple Sender

We model a simple sender process in this example. We aim at showing the use of superstates. The main process is shown in figure 4.3 and the superstate in figure 4.4.

The protocol is based on the wait for an acknowlegment. If it comes too late, there is a time out transition and the sender fails. We try this twice, with the same superstate instanciated twice.

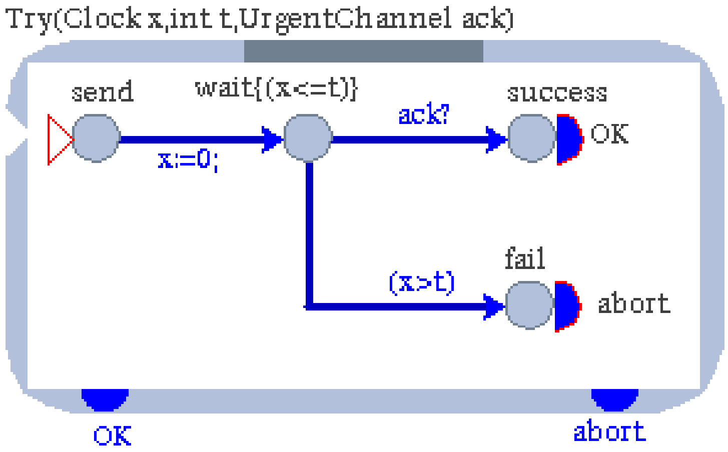


Figure 4.4: The superstate of the sender.

The model here is incomplete since there is no receiver, but this is sufficient for the purpose of the example. We can see on the graphical representation that the alternative graphical notations have been implemented. The UL^{++} source is quite simple and it is easy to recognize the graphical definition :

```
// UL++ file

class Try(Clock x,int t,UrgentChannel ack) extends SuperState {
data:
control:
 State send;
 State wait{(x<=t)};
 State success;
 State fail;
 Exit OK=success;
 Exit abort=fail;
 Entry send;
 Transition send->wait{x:=0;};
 Transition {(x>t)}wait->fail;
 Transition {ack?}wait->success;
}

class Sender(int t,UrgentChannel ack) extends Process {
data:
 Clock x;
control:
 State Start;
 State OK;
 State Fail;
 Try(x,t,ack) try1;
 Try(x,t,ack) try2;
 Transition Start->try1;
 Transition try1.abort->try2;
 Transition try1.OK->OK;
 Transition try2.OK->OK;
 Transition try2.abort->Fail;
 InitState Start;
}
```

```
class Test(int t) extends ConcurrentProcess {
data:
 UrgentChannel ack;
control:
 Sender(t,ack) send;
}

//Instance declaration
Test(5) test;
```

We notice the entry/exit declarations and how we define transitions using these connectives. We can skip the ULi representation and look at the `ta` format directly, which is interesting because it shows how these transitions have been resolved :

```
urgent chan ack;

process send {
  clock x;
  state Start,OK,Fail,try1_send,try1_wait{x<=5},try1_success,try1_fail,try2_send,
try2_wait{x<=5},try2_success,try2_fail;
  init Start;
  trans try1_send -> try1_wait {
          assign x:=0;
        },
        try1_wait -> try1_fail {
          guard x>5;
        },
        try1_wait -> try1_success {
          sync ack?;
        },
        try2_send -> try2_wait {
          assign x:=0;
        },
        try2_wait -> try2_fail {
          guard x>5;
        },
        try2_wait -> try2_success {
          sync ack?;
        },
        Start -> try1_send {},
        try1_fail -> try2_send {},
        try1_success -> OK {},
        try2_success -> OK {},
        try2_fail -> Fail {};
}

system send;
```

The transitions have been “connected” to the real states. This step will be explained in the section about the translation to UPPAAL.

Chapter 5

Implementation

In this chapter, we expose the implementation of UL^{++} in the tool VisualUL. We explain the class hierarchy of the program, written in Java. Then we give the outline of the definition of a system step by step. We focus on the problem of the instanciation. Finally, we explain how we make the translation to Uppaal.

5.1 The Class Hierarchy

The class hierarchy of the Java program comes directly from the class hierarchy of UL^{++} and is shown in figure 5.1. There are some slight differences which come from the fact that we have chosen the alternative representation and only the connected connectives, as precised in the specification. For this reason, the Connection class does not appear. Furthermore the InitState class does not appear because it is just one reference per process, which indicates the initial state. We notice the VirtualState Java interface which materializes the notion of virtual state that we exposed in the syntax chapter.

The composite objects are all extensions of HierarchicalObject, because we use common methods for all of them. However, State inherits the methods as well even if we do not need them. This comes from the single inheritance model of Java. We would have preferred some hierarchy with State extends StateObject, and SuperState extends StateObject and HierarchicalObject. Instead, we use a marker interface which is CompositeObject. This is used for the system description (SystemMember), since the system is a composite object, which has one process or one concurrent process member which is the system definition. For this reason as well, SystemMember extends ControlMember.

The class ControlMember represents the control members in the control field of a composite object. It has a name and a class model. This class implements the members of an object and is responsible for the instanciation of the members.

TreeObject is an interface for the tree representation of the classes. State implements this interface because in the early stages of the design, it was thought to use a system tree representation which would include states. It has not been done, but State still implements the interface. This costs almost nothing since the TreeObject interface is very simple.

Concerning the data part, the implementation follows exactly the design of UL^{++}. One can just notice the class IntConstant which is used to represent the `int` type.

In parallel with these classes, there is another important hierarchy which is the graphical peer objects hierarchy. To make things clear, these classes are prefixed with GUL, which stands for graphical UL object. There are a family of classes prefixed with GI, which stands for graphical instances. The difference between these two families is simple : the GI family inhibits properties of the GUL family, so that they may only be moved, with no respect to the current edit mode, valid only for the GUL objects. This hierarchy is given in figure 5.2.

One can notice the GULVirtualState interface which is the equivalent of the VirtualState

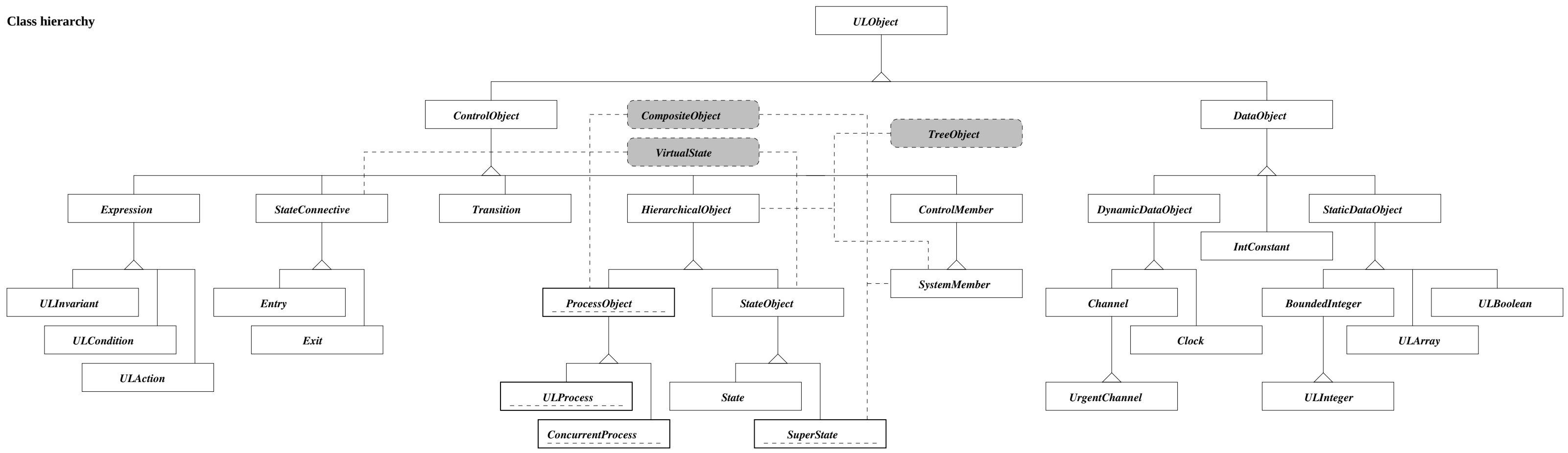


Figure 5.1: Implementation hierarchy.

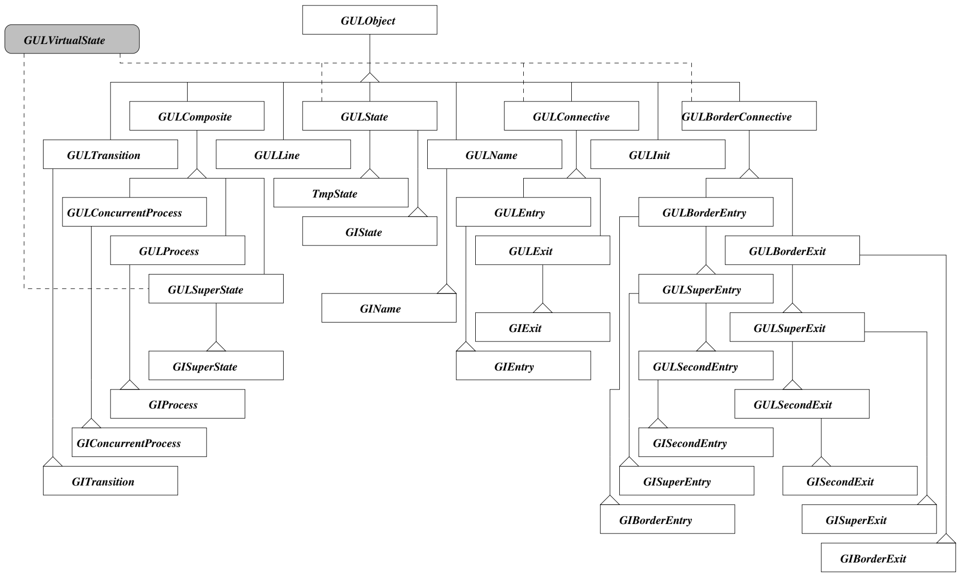


Figure 5.2: The graphical objects hierarchy.

interface for graphical objects. This interface marks classes to draw transitions and to draw the init state pointer. The TmpState class is just used to define a state, it is a temporary object.

Concerning the composite classes, they are all GULComposite extensions, since a composite is basically a rectangle which can be resized and moved.

The complexity of the GUL hierarchy comes from the connectives : we need connectives attached to a state, which are GULConnective classes ; we need others attached to superstates, which are GULSuperEntry, GULSuperExit ; we need connectives attached to connectives, which are GULSecondEntry, GULSecondExit ; we need connectives, which are the images of the previous definitions in order to mark a superstate member. The resulting hierarchy comes from the fact that all these classes have similar graphical behaviours.

A drawback of these hiearchies These choices appeared to be rather good but there is a lack of a Connection class, somewhere between the connective and the virtual state. This appears in the implementation when addressing the connectives, and it was not really a problem because the information is duplicated in the support of the connectives, namely the superstate. For this reason, a Connection class was not created.

5.2 Modeling a system

To model a system, one has to follow three important steps which are extension, composition and instanciation. We address these issues in this section.

Extension As the automaton is built on the three main composite classes ConcurrentProcess, Process and SuperState, one has to extend these classes to create the skeleton of a system. These classes are primitive classes and may not be used directly since they are empty. So, the tool provide a simple *extend* function available in a popup and in the tool bar.

Extending a class implies :

- inheritance, which means that all components of the parent class are inherited. In the tool, this is materialized by a copy of the previous members, such as states, transitions ... which are darker than the non inherited ones. Inheritance in the GUI is more than simple static inheritance since the user defines and delete components. These components are updated dynamically in the whole hierarchy and in the members. This is specially visible when defining a connective. The connective is updated on the corresponding members. When deleting a component, this is even worse and a special algorithm is used to find all dependant objects in order to delete them as well. As an example, one can delete a state, with a connective attached to it ; somewhere a transition is defined to this connective ... we have to remove all these objects.

- inheritance again, for the data members, with the names of the variables in the data list which may change dynamically if the user reuses a name defined in the parent class. The shadowed variables have their names changed automatically. The data list gives the local data and the data from the interface with different colors. They are treated in the same way.
- name checking : the name of the class must be unique and this is checked in a class name space which is kept in the program. The members of a class have their names checked as well. Data members may be shadowed but not the control members. When the user wants to define a name for them, or when a name is produced automatically, then this name is checked in the parent classes and in the extensions as well, in order to prevent name collision.

We prove that our naming convention is consistent, namely the instanciation gives unique names without collision : the full name of any object is defined as *parentName*`[dot]`*parentName* ... *localName*. In order to get a collision, all the parents have to be equal and the *localName* parts as well. The hierarchical paths are equal implies that we are in the same class model, and the same instance member. But, by definition, the local names are unique and are checked so that there is no collision. This is assured for the control members, and the data variables have shadowed names if any collision occurs. So the names are unique. We have to check that the shadowed names keep this property : the '$' character is special and is used only in this context. An equality on two names implies that the original class owner has two local defined data variables with the same name, which is forbidden. So the naming convention is consistent and garantees that the full names are consistent when instanciating, and even when flattening.

Composition The composition process is in two steps : firstly, the user clicks on the compose button in the tool bar, then he has to choose the class model of the member he wants to create. The type of the class is checked in order to respect the composition rules.

Secondly, he has to provide the instance parameters as a list of names, or expressions. The validity of the parameters is checked when instanciating.

Another check is performed when compositing, namely we assure that no recursive definition is created. This means that a component A which includes B is sure that it is not referenced in B or in any of B's component.

Concerning the other members, it is easier : the user chooses his mode, which allows to define states, entries, exits or transitions and the definition is straightforward. The internal representation uses the ControlMember class, as for composition with composite classes. This is a part of the composition task.

Instanciation The instanciation reuses the same class hierarchy as before, but duplicates the originals which were used to create the models. The substitution is done via a hashtable in each composite class which makes the translation from a name to another. The expressions are evaluated and the resulting name is substituted. This is done in the data list and in the invariants, conditions and assignments. The control members have their names changed as well, so that they receive absolute names. When several members of a same class model are declared, then the instanciation process creates a separate objects for each member, so that they become real instanciations. The resulting objects are reconnected together and special instance graphical objects are created for them.

We illustrate the substitution algorithm with a "TrickyNames" examples, where we abuse (on purpose) on the shadowed variables to try to confuse the program. The reader should be carefull in order not to be confused by the names, because the program got it right! Here is the UL^{++} source of this example :

```
// UL++ file

//this is the parent class: we have c and x, x is used locally
class a(Clock c) extends Process {
data:
 Integer x;
control:
 State node0;
 State node1;
 Transition node0->node1{x:=0;};
 InitState node1;
}

//in comments, the inherited interface, this is automatically given
//we redeclare in the interface c and locally x, so that we will get
//shadowed variables in the interface and in the local data
class b(/*Clock c*/Integer c) extends a {
data:
 Clock x;
control:
 State node2;
 //and we use the data locally, as well as the shadowed names
 //note that this file is parsed by the program which checks the
 //validity of the local, inherited and shadowed variables
 Transition {(x>3) && (c==4)}node0->node2{a$x:=0; a$c:=0;};
 InitState node0;
}

//we continue with a third class and we reuse x and c locally, with different
//types
class c(/*Clock a$c,Integer c*/) extends b {
data:
 Clock c;
 Integer x;
control:
 State node3;
 //and we use them all
 Transition {(c>3) && (x==2) && (a$c==2) && (a$x==1)}node0->node3{b$x:=0; b$c:=0;};
 InitState node0;
}

//this class is just to ask for an instance of each type
//and we expect to get the right variables with names like a$c for example
class ab() extends ConcurrentProcess {
data:
 Clock x;
 Integer i;
control:
 a(x) fooA;
 b(x,i) fooB;
 c(x,i) fooC;
}

//Instance declaration
ab() fooABC;
```

We give the ta translation of this system definition :

```
clock x;
int i;

process fooA {
  int x##;
  state node0,node1;
  init node1;
  trans node0 -> node1 {
          assign x##:=0;
        };
}

process fooB {
  int a$x;
  clock x##;
  state node0,node1,node2;
  init node0;
  trans node0 -> node1 {
          assign a$x:=0;
        },
        node0 -> node2 {
          guard x##>3,i==4;
          assign a$x:=0, x:=0;
        };
}

process fooC {
  int a$x;
  clock b$x;
  clock c;
  int x##;
  state node0,node1,node2,node3;
  init node0;
  trans node0 -> node1 {
          assign a$x:=0;
        },
        node0 -> node2 {
          guard b$x>3,i==4;
          assign a$x:=0, x:=0;
        },
        node0 -> node3 {
          guard c>3,x##==2,x==2,a$x==1;
          assign b$x:=0, i:=0;
        };
}

system fooA,fooB,fooC;
```

We assume that the user will use the last version of UPPAAL which accepts '#' and '$' as valid characters. '#' is used to translate the hierarchy and '$' is used for the shadowed variables. In previous examples, '_' has been used in order to make tests with the existing version, which was successful. However this replacement does not garantee the consistency in the name space.

The program uses a trick here in order to keep names simple : '#' is added after the local names when a collision occurs with the global name. If the name comes from some nested hierarchy, this does not happen. The interested reader can check that the names are correct and that we have kept the local names when possible.

Loading and saving a file When we load a file, the parser checks the consistency of all the defined names. Transitions to entries and from exits are checked so that valid connectives are used. The assumption of no forward definition, as precised in the language specification is widely used, and an error may occur[1] if this is not respected. This garantees that there is no recursive definition in the system definition.

When we save a file, we sort the classes such that the no forward rule is respected. This sort is successful if there is no cycle, which is checked during the composition step. The total order rule is as follows, in the order of comparison :

1. if A is an ancestor of B, then $A < B$
2. if A is contained in B, then $A < B$
3. if $depth^2(A) \neq depth(B)$ then $A < B \iff depth(A) < depth(B)$
4. else we take the lexical order of the names, which is correct since class names are all unique

We use a pre-ordering in practice which consists of saving superstates first, then processes and finally concurrent processes. This is obvious with respect to the composition rules.

5.3 Translation to Uppaal

To translate to Uppaal, we assure a minimal compatibility with expressions. Then we need to flatten our model and to perform some syntax translations.

The parser First of all, some words about the parser. To preserve compatibility with Uppaal, but in order to have some flexibility, we have chosen a compromise in the parser. The parser is "clever but not too much". This means that the expressions are checked and parsed with the BNF definition of the language, which is a superset of Uppaal language concerning the expressions (guards, invariants, assignments and synchronizations). Basic type checking is performed in order to prevent big mistakes, but the result of the instance may be refused by Uppaal if the user wants to use the full power, or the weakness of the parser. See [LPW97b] for more information on what Uppaal accepts.

Flattening a model The flattening process consists of breaking all the hierarchical constructions. We extract the objects and put them in the parent container, and we repeat the process until we reach the top level. While opening the components, we keep the full names so that the names stay consistent. We may display short names, if the user chooses this option, but full names are always taken into account.

One important rule when flattening is to take care about the invariants, and specially the "super invariants". As shown before, the resulting invariant of a simple state, is the conjunction of the invariants of this state and the invariants of the superstates that contain it. We say superstates because we include all the hierarchical layers. So flattening implies changing the invariants as well.

We include in this process the connective resolution, namely we connect all the transitions to the states. Doing this for anonymous entries and exits implies to duplicate transition.s We defined in the semantics that the transition to a superstate may be fired non deterministically to any entry point. Furthermore, the transition from one anonymous exit is the same from all anonymous exits, thus the duplication in this case as well.

In order to ease the syntax translation, we make a last transformation in the flattening process, which is to translate the synchronization with value passing to an equivalent structure without value passing. The principle is illustrated in figure 5.3.

Apart from the syntax translations, we make the following changes :

[1] which means the input is rejected.
[2] The depth of the hierarchy.

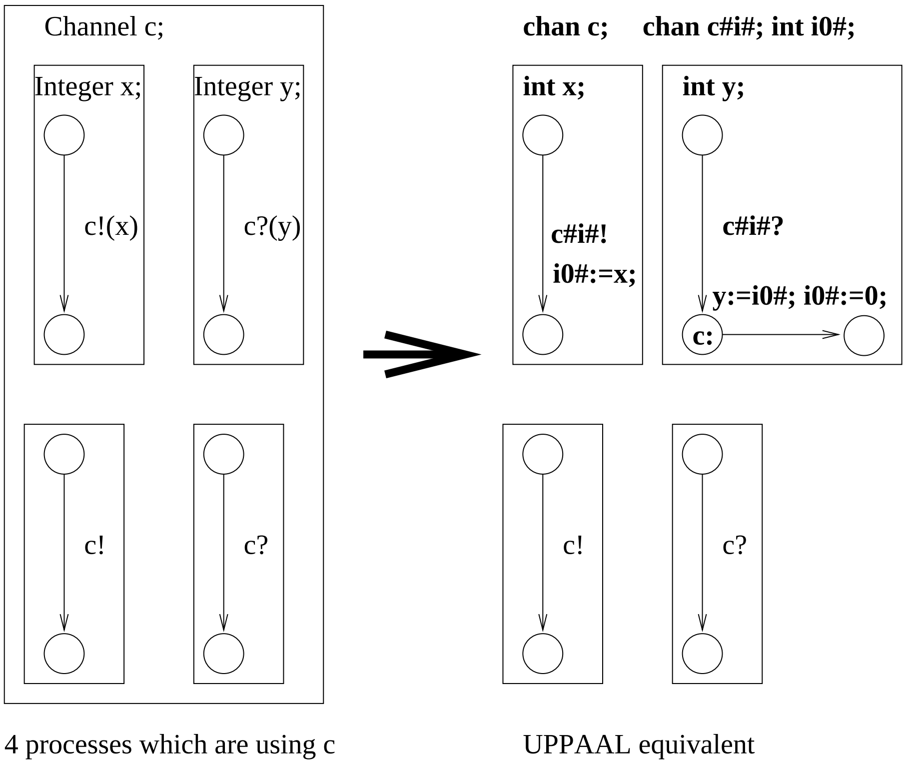


Figure 5.3: The transformation of synchronization with value passing.

- we create a shared temporary variable which carries the information only when we use the channel and which is set to 0 otherwise in order not to expand the state space.
- we create a new channel which replaces the old one. This new channel has in its name the signature of the arguments, so that only channels with the same signature may synchronize together. This is illustrated with the use of the channel c with or without value passing.
- on the sender side, we insert the action `temporary_variable:=value_to_be_passed`. This construction allows to pass complex formulas as well.
- on the receiver side, we insert a commit state, in order to keep the same behaviour as in UL^{++}. The goal of this is to have first the assignment of the temporary variable, and just after, we read this variable and we set it to 0 again. This constitutes the inserted actions on the inserted transition between the created commit state and the target of the original transition.

Syntax transformation The transformations are almost straightforward to UPPAAL. Concerning the data part, we use these rules :

- Integer is translated to int.
- BoundedInteger(min,max) is translated to int(min,max).
- Boolean is translated to int(0,1), true to 1, and false to 0. Simple expressions of the form $boolB$ are translated to ($boolB == 1$) and $!boolB$ to ($boolB == 0$). However assignments of the form $boolB := (x < 3)$ are not translated. They can not be translated to UPPAAL. This is one limitation of the process.

- array[min,max] is translated to array[$max - min + 1$] and the indices used to access these arrays have an offset of $-min$.
- Channel is translated to chan, with the previous transformations if needed.
- UrgentChannel is translated to urgent chan, with the previous transformations if needed.
- Clock is translated to clock.

Concerning the expressions, we transform conditions of the form (*bool*) && (*bool*) && (*sync*!/?) to *guard bool*, *bool*; *sync* ..!/?;. However, a feature of the parser is that guards with urgent channels are accepted in UL^{++}, but not in UPPAAL. This is not resolved and constitutes a limit in the translation. The translation for invariants and assignments are straightforward.

The rest of the transformations are easier : State to state, CommitState to state with a declaration commit, Transition to trans. Exits and entries are resolved by the flattening process, so we do not have any problem.

What is missing? We have not implemented the overriding of a state or a superstate by a superstate. This is an important feature of the language but the implementation is strong enough even without this feature to demonstrate that the principles of UL^{++} work. The goal was to implement a functional prototype before a deadline, which has been done.

The rest of the language specifications have been implemented exactly as specified. However, we do not have yet the possibility to simulate with hierarchical traces. To do so we need a dedicated simulator.

Chapter 6

Conclusion and Further work

6.1 Conclusion

The implementation of UL^{++} has followed the specifications and the GUI is user friendly. This was the goal that we wanted to achieve. Simulation is possible with the new Danish GUI of UPPAAL, so that we can check our models with the `verifyta` of UPPAAL and simulate them as well.

The drawback of a new language is of course a new way of thinking for the modeling task. This may not be so bad since the proposed solution is highly structured and hierarchical.

We give an overview of the GUI in figures 6.1 and 6.2.

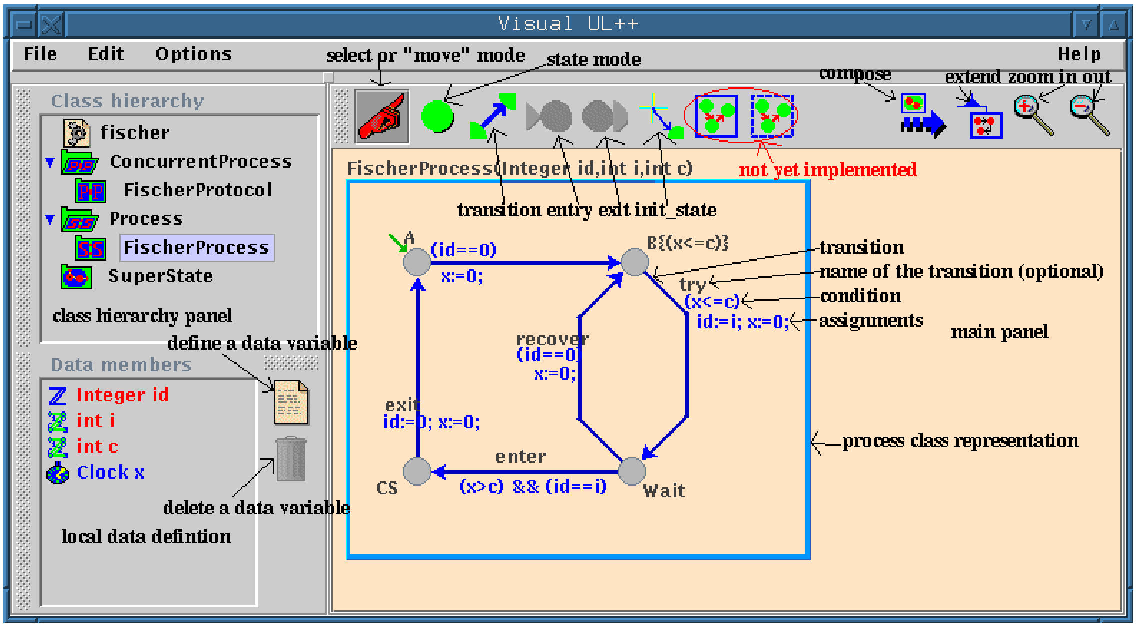


Figure 6.1: Overview of the main window, where the user edits his system.

A quick help web page is available at *http://www.docs.uu.se/~adavid/ULhelp/index.html*. The GUI is able to access this page on-line in one of its own windows.

The implementation of this prototype is not very good in terms of speed. Java has been overestimated and is not as powerful as expected with big programs. Furthermore Java is not yet very stable. The fact that it is a prototype and that Java is not stable is a garantee of the presence of some bugs.

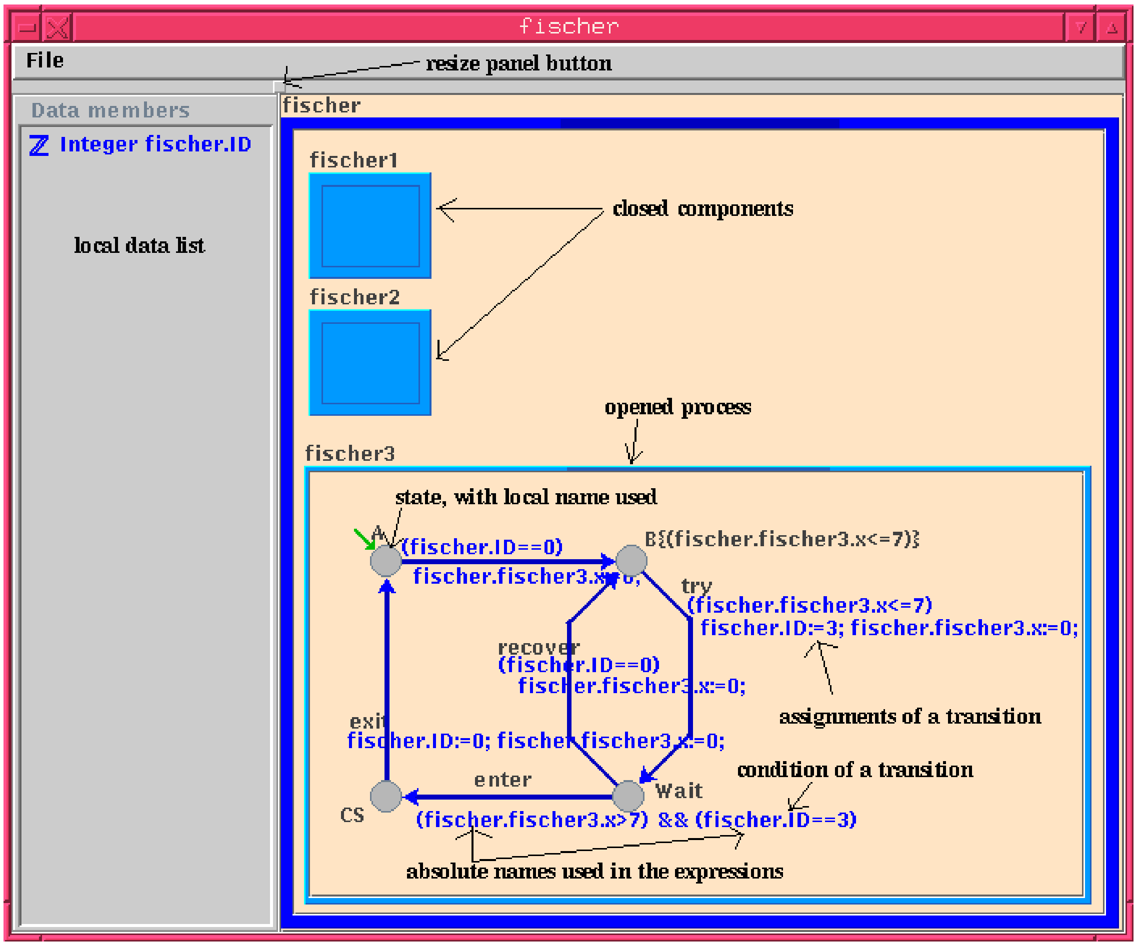


Figure 6.2: Overview of the instance window, where the user visualizes his system.

6.2 Further Work

The first thing to do in order to continue to develop this language is to clean it. This means remove as many objects as possible. The concept of objects is very nice, but it is possible to argue that we are cheating for some objects.

Secondly, it is a good idea to generalize the notion of state, as it was given in the semantics, so that we have this in the basis of the language. This means to suppress the State (or the SuperState) class and condider the SuperState (or the State) as a general state with a function (predicate in semantics counterpart) which says if the state is empty or not. This would make things simpler in the composite object hierarchy.

Thirdly, we want to simulate with the power of the hierarchy, namely to use hierarchical traces. To do so, one could proceed in two steps : first develop a “gate” between the GUI and Uppaal in order to use the existing tool and in order to develop the protocol and solve the specific problems of the GUI when doing such a simulation. This allows to focus on the functionalities that we want to get eventually. Secondly, implement the simulator itself at the place of the gate. This is a modular solution which is possible.

References and Related Sources

References

- [LPW97b] Kim G. Larsen, Paul Pettersson and Wang Yi. "UPPAAL in a Nutshell", *Springer International Journal of Software Tools for Technology Transfer 1(1+2)*, 1997.

- [LP97] Henrik Lnn and Paul Pettersson. "Formal Verification of a TDMA Protocol Start-Up Mechanism", *Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems, pages 235-242*, Taipei, Taiwan, 15-16 December, 1997.

- [DBP98] Douglas, Bruce Powell, "Real-Time UML : Developing Efficient Objects for Embeddded Systems", *Addison-Wesley Object Technology Series* 1998.

Related sources

- Carsten Weise, "Extended UPPAAL textual format", April 24, 1998.

- Per Stoffer Jensen, Jesper Gravgaard Jensen, "Design and Implementation of NewPaal, A Modelling Language for Real-Time Systems", master's thesis, 1997.

- David Harel, Amnon Naamad, "The STATEMATE Semantics of Statecharts", ACM Transactions on Software Engineering and Methodology, Vol 5., No 4. October 1996, Pages 293-333.