ISSN 0280-5316
ISRN LUTFD2/TFRT--5681--SE

Statecharts in ABB Control^{IT}

Gert-Ola Carlsson

Department of Automatic Control
Lund Institute of Technology
December 2001

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESES

Date of issue
December 2001

Document Number
ISRN LUTFD2/TFRT--5681--SE

Author(s)
Gert-Ola Carlsson

Supervisor
Karl-Erik Årzén, LTH
Ulf Hagberg and Niklas Sjöberg at ABB Automation Technology Products.

Sponsoring organization

Title and subtitle
Statecharts in ABB Control IT (Statechart i ABB Control IT)

Abstract
A graphical editor for Statecharts has been implemented. The editor is integrated in the ABB Control IT framework. The editor is written in Java/Swing and uses the graphical class package JGo.
The developed Statecharts models are stored on file using XML. The editor can also generate executable code, in the form of IEC 61131-3 Structured Text (ST) code. The code is represented in an XML-document and transferred into an application in the Control Builder, the Control IT development tool. The application can then be downloaded to an ABB controller.

Using OPC techniques, the execution status and variable values can be monitored on-line in the editor. Hence, it is possible to both design and execute Statecharts models directly from the editor.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
86

Recipient's notes

Security classification

Lund Institute
Of Technology
Lund University

Statecharts in ABB Control ^{IT}

Gert-Ola Carlsson

Department of Automatic Control
Lund Institute of Technology
November 2001

Table of contents

Preface

This report presents the master thesis “Statecharts in ABB Control ^{IT}. The thesis has been performed at ABB Automation Technology Products in Malmö and at the Department of Automatic Control, Lund Institute of Technology.

Acknowledgements

I would like to thank my supervisors: Karl-Erik Årzén at the Department of Automatic Control, Lund Institute of Technology, Ulf Hagberg and Niklas Sjöberg at ABB Automation Technology Products.

1. Introduction

When people started to program computer systems, programmers where often regarded as wizards. Today programming is used in every day work and natural to use for almost every person. As programming has expanded the requirement for user friendly programs have increased. Programming has to be more intuitive and performed at higher level than before. In this development however, more effort has to be taken by software developers to meet the increasing demands.

In this development, ABB Automation Technology Products has introduced the Control ^{IT} concept. This includes the Control Builder, which is a powerful programming tool mainly intended for industrial environments. The Control Builder provides the languages specified in the IEC 61131-3 standard that could be used in combination. The languages have different properties making them suitable to solve different kind of problems. The need to model different execution modes in the Control Builder has raised. This requires that a language based on state machines has to be introduced into the Control Builder. The use of Statecharts has at the same time increased for instance in UML modelling. These two facts have caused a desire to introduce Statecharts in Control ^{IT}.

This thesis constitutes the starting point in the development of Statecharts into the Control ^{IT}. Statecharts is based on state machines and provide a way to graphically modelling discrete event systems. Although Statecharts is a powerful tool it is rather easy to understand. To simplify the work, the prototype is not completely incorporated into the Control ^{IT} at this stage. A stand-alone prototype should be implemented where Statechart models could be designed. The prototype is implemented in Java. To simplify creation and structuring of graphical elements, a graphical package called JGo is used. To integrate the prototype into the Control ^{IT}, the model is executed in a Soft Controller. Furthermore, the progress of parameters during execution can be followed in the prototype directly.

1.1 Outline of the report

Chapter 2 gives a brief description of Control ^{IT}. The Control Builder, the Soft Controller and the OPC Server are products included in the Control ^{IT} which are also described.

Chapter 3 deals with the IEC 61131-3 standard and describes the included programming languages.

Chapter 4 gives a description of the definition of Statecharts. The description is based on the definition made by David Harel, the founder of Statecharts.

Chapter 5 explains the main features of JGo, the graphical software package used in the prototype.

Chapter 6 describes the Statechart Editor Prototype, which is the implemented application providing possibility to design Statechart models.

Chapter 7 describes how executable code is generated from a graphical model in the prototype.

Chapter 8 deals with the communication between the prototype and the Control ^{IT}. The problem to transfer the generated code from the prototype to the Control Builder is first dealt with. Communication with the OPC Server to get parameter values during simulation is then considered.

Chapter 9 mentions some improvements and further development of the prototype.

Chapter 10 contains a summary of this thesis.

1.2 Purpose

The major purpose of this thesis is to introduce a sixth language into the Control Builder product, the programming tool included in Control ^{IT}. The new language has to be of graphical nature and based on the concept of state machines. The purpose is also to specify such a language and implement it in a stand-alone prototype. This prototype should provide for graphical design of models using this language. It must further be possible to store the graphical representation in textual form in an XML document. The execution of a designed model should be translated into Structured Text code to be executed in Control ^{IT}. The prototype should also provide a way to transfer the code into the Control Builder for execution. Parameters should be updated directly in the graphical model during execution in the Control ^{IT}.

1.3 Scope

This thesis is meant for programmers and engineers that have an interest in modelling discrete event systems using Statecharts. The report could advantageously be read by people who have basic knowledge in state machines and how such machines are executed.

2. Control^{IT}

Overview: *This section presents the ABB Control ^{IT} concept. First the main purpose of Control ^{IT} is given. Then different products included in the Control ^{IT} are described. The Control Builder, which is the programming tool in Control ^{IT}, is first presented. Then the Soft Controller, in which the actual application is executed, is presented. The OPC Server is then briefly described. The OPC Server can communicate through the OPC Interface with controllers "online" during the execution. The MMS server, which coordinates connections of several controllers to the Control Builder, is described. Finally the Control Builder Open Interface is presented. This interface is provided by the Control ^{IT} to open up the Control Builder for external communication.*

2.1 Introduction

Control ^{IT} is an open Industrial ^{IT} control system to be used in modern industry. The concept is to integrate all business processes in a company into one single system. In this way environments like supply, production and finance could use the same system. The integrated system then provide for control in real time either locally or through Internet. The ambition is that every Industrial ^{IT} product can be integrated together with other Industrial ^{IT} products in a plug-and-produce manner to form a total Industrial ^{IT} solution.

2.2 Control Builder

Control Builder is a programming tool included in Control ^{IT}. Control Builder is fully integrated in Windows 2000 Professional and facilitates programming of several ABB controllers. Control Builder provides five programming languages according to IEC 61131-3. The included languages are: Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL), Ladder Diagram (LD) and Sequential Function Chart (SFC). Control Builder is actually represented by three different products: Control Builder Basic, Control Builder Standard and Control Builder Professional. The basic version has limited functionality while the professional version has very powerful functions, like handling of special Control Modules. Control Modules are used to package reusable control solutions and by that make it possible to program in an object oriented way.

When the Control Builder is started the Project Explorer is first shown, see Figure 2.1. In the Project Explorer the user may create and modify a project but may also configure controller software and controller hardware. At the top of the Project Explorer the title bar is shown. In the title bar the actual project name is displayed. The title bar also indicates the actual mode of operation i.e. if the Control Builder is Online or Offline. A project has a folder named Libraries, which contain predefined functions that can be used in the program. The Application folder contains the real code that is to be executed in the controller. The application is structured into several programs that are connected to separate tasks. It is possible to define special data types and function blocks that are used in the program. The controller folder specifies the actual controller where the programs should be executed. The Hardware folder inside the controller specifies almost exactly the "real" hardware, which is connected to the I/Os. The Task folder contains three tasks: Fast, Normal and Slow. The task executes the associated programs and has settings for priority and interval time. This gives freedom for the user to place important actions in a separate program. The program can then be associated to a task that is given high priority and/or short interval time.

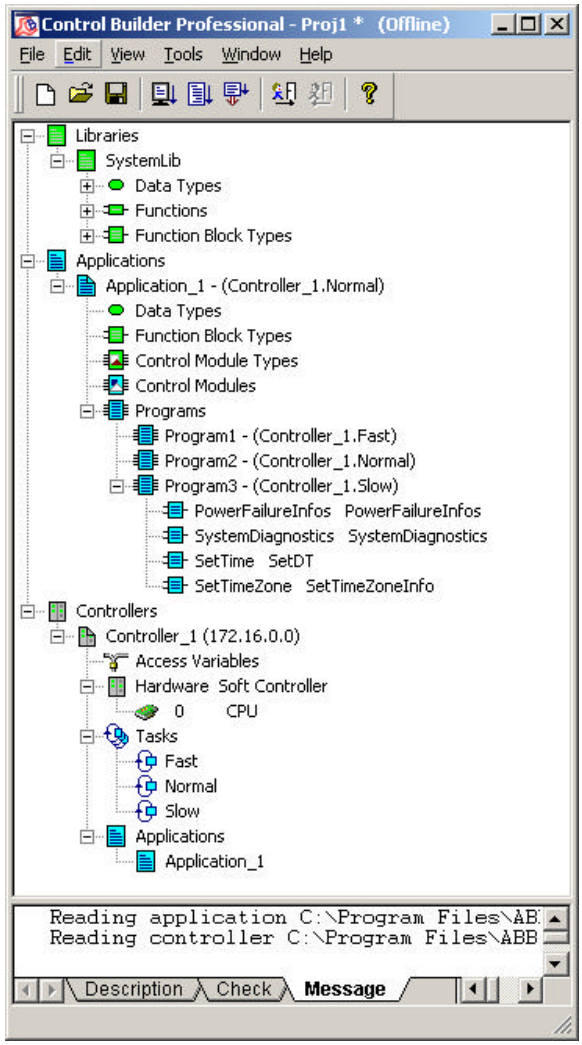


Figure 2.1 Project Explorer in the Control Builder.

By double clicking on a program item in the Project Explorer, the program editor is displayed. Figure 2.2 shows an example of such an editor. In the grid at the top of the window, parameters can be declared that are used in the program. The bottom area of the program editor is called the Code Pane. Code can be edited in the Code Pane in any of the five languages supported in the Control Builder. Several Code Panes can be stacked over each other in the pane. Selecting the associated tab at the bottom of the Program Editor shows a particular Code Pane. All Code Panes together constitute the actual program. It is not necessary to use the same programming language in all Code Panes in a program. Any of the five languages supported in the Control Builder can be combined in different Code Panes. The parameters declared at the top grid of the Program Editor are, however, common for all Code Panes in the program.

A project can either be simulated directly in the Control Builder or it can be downloaded into a special controller for execution. Controllers act as targets for applications developed in the Control Builder. When a project is ready for execution the whole application is downloaded into the controller and executed detached from the Control Builder. In simulation mode, it is not necessary to download the program into the controller. When simulation starts values of the variables are shown on-line, see Figure 2.3. A highlighted variable indicate that the value is true. During simulation, variables can be changed manually in the window.

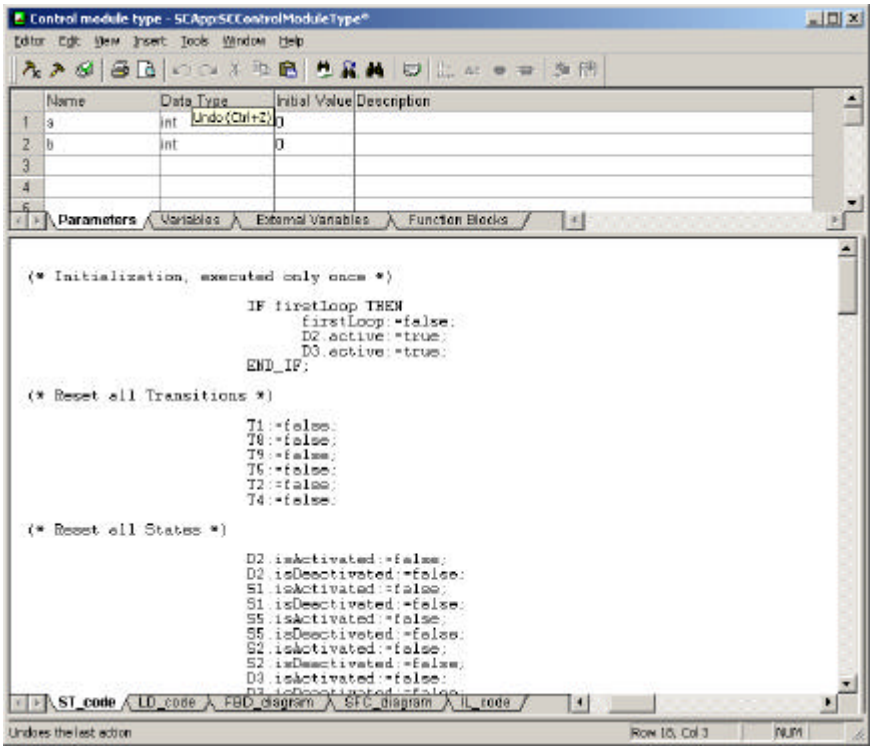


Figure 2.2 Program Editor in Control Builder.

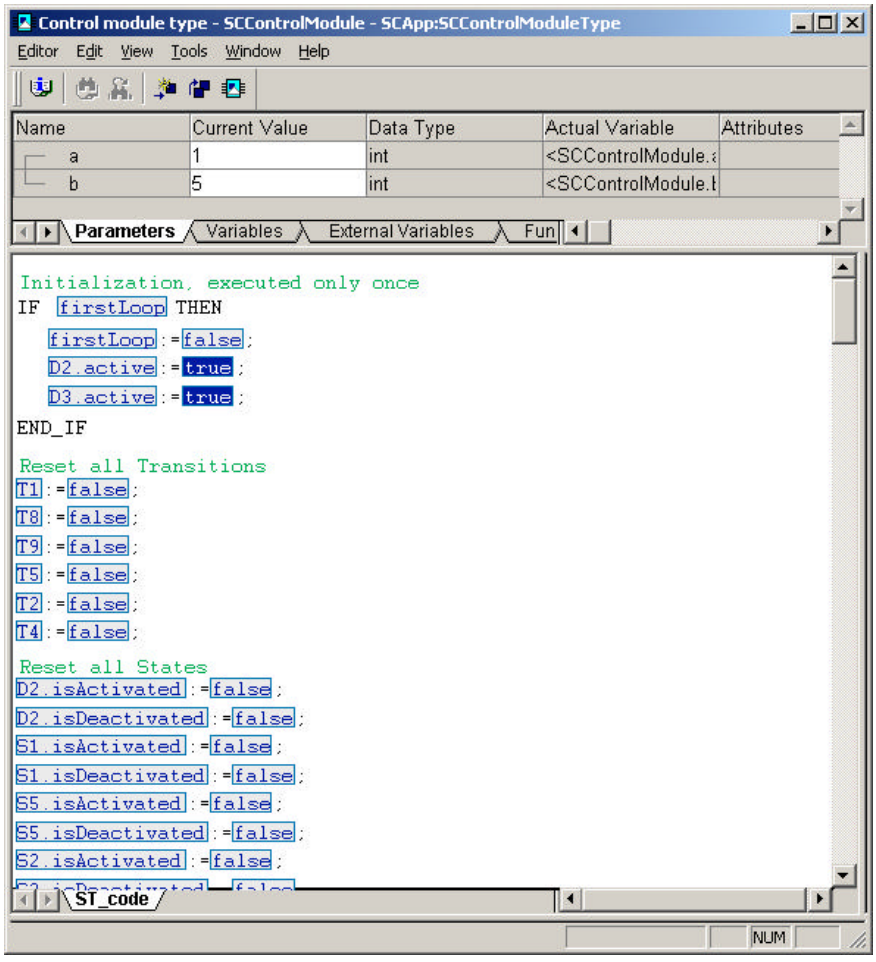


Figure 2.3 Program Editor during simulation.

2.3 Soft Controller

The Soft Controller is a special controller turning a PC into a process controller. The Soft Controller is supported by Microsoft Windows 2000. When an application in the Control Builder is created it can be downloaded into the Soft Controller for execution. The control panel for the Soft Controller is started as an ordinary application in Windows 2000, see Figure 2.4. The Soft Controller is started from the control panel by pressing the start button. Immediately after start up the Soft Controller only contains an empty application.

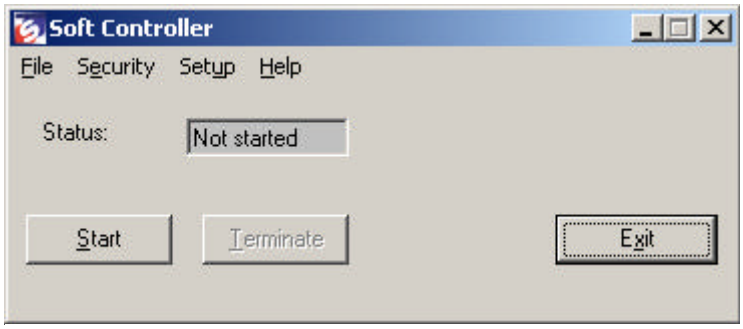


Figure 2.4 Control panel for Soft Controller.

2.4 OPC Server

OPC (OLE for Process Control) is a standard interface for communication between different data sources. The OPC server implements the OPC interface to communicate between OPC COM objects and OPC clients. COM (Component Object Model) is a technology developed by Microsoft Corporation to integrate components. All COM objects are accessed through interfaces. The OPC server gives possibility to access run-time data from an application executing in a controller. An OPC client connects to an OPC server and communicates to the OPC server through the interfaces. The client sees only the interfaces. An OPC client can connect to OPC servers provided by one or more vendors, see Figure 2.5. It is the vendor-supplied code that defines the data to which each server has access. The structure within the OPC Server consists of three main objects: the server, the group and the item. The server object provides information about the server and contains the OPC group objects. The group object has information about itself and also provides properties to organize OPC items. Data can either be read from or written to the items in the group. The client can ask for the items in a group every time it wants to update the values. The OPC client can also configure the rate at which the server should provide data changes to the client. There exist two types of groups, public and local. Multiple clients may access public groups while local groups are private for one single client. In each group the client can define one or more OPC items, see Figure 2.6. The items represent the data sources within the OPC server, but it only represents a connection to the actual source. The item caches data from the memory location where the actual value is stored. Items are not accessible as objects, but are reached through the group. Each item has three properties: Value, Quality and Time. The Value is the actual value of the specified item. The Quality tells if there is something wrong with the actual value in the Value property. The Time is a timestamp indicating when the value was collected from the memory location.

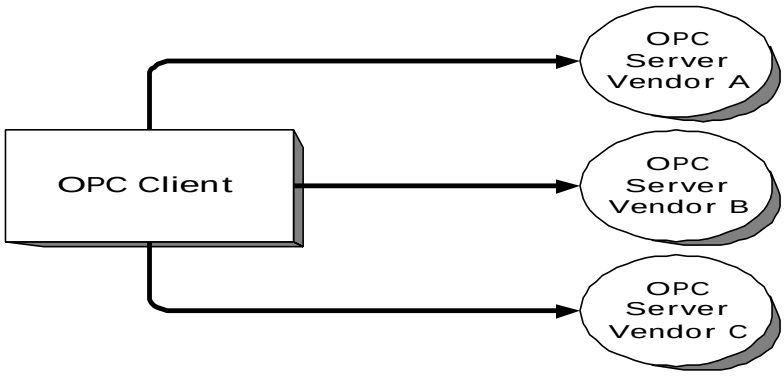


Figure 2.5 OPC Client connections.

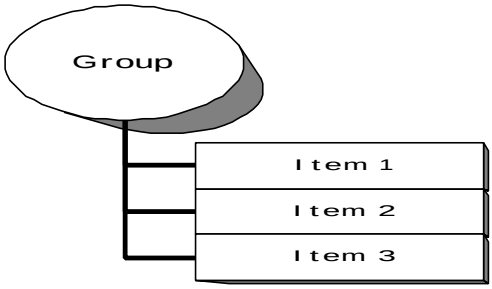


Figure 2.6 Structure of an OPC Group.

When the OPC server is started from the window starting bar the configuration panel of the OPC Server is shown, see Figure 2.7. From this panel it is possible to specify controllers to be connected to the OPC server. When the connections are established the OPC server has access to all data in the controllers.

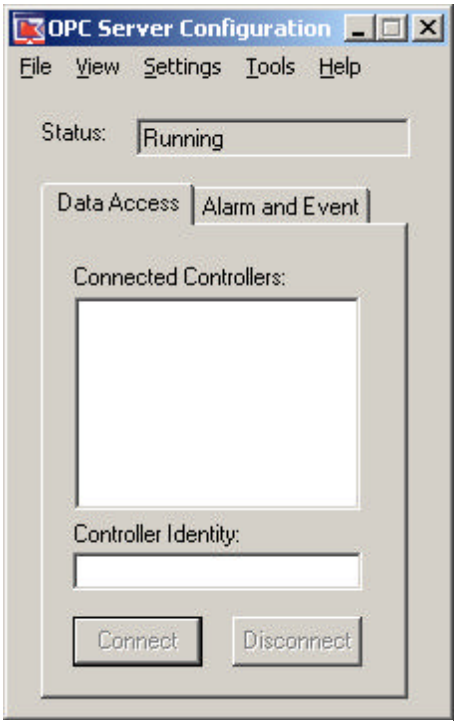


Figure 2.7 Configuration panel for OPC server.

2.5 MMS Server

The MMS (Manufacturing Message Specification) Server starts automatically when any of the previous products are started. The MMS server makes it possible to run more than one Control IT product on the same PC at the same time, see Figure 2.8. To separate the different product from each other a special identification number can identify them. The Control Builder has number 1, the Soft Controller has number 2 and the OPC server has number 22. Throughout this an OPC client can receive data from a controller by calling the OPC server. The OPC server then forwards the call to the actual controller via the MMS server that returns the value. The value is finally sent to the calling OPC client.

2.6 Open Interface

Another way to communicate with the Control Builder is by using the "Control Builder Open Interface" [Crilfe, 2001]. The Control Builder Professional exposes an open COM interface to other applications (EXEs). The interface provides a set of methods that other applications may use to create, change or delete objects in the Control Builder. Almost all functionality within the Control Builder can be manipulated using the methods in this interface. The different objects in this "Open Interface" are represented as XML documents. The reason to use XML is that it is an independent language and a widely accepted standard. Internally the Control Builder uses Microsoft's XML parser MSXML. In addition, this parser is able to validate the XML document according to its XML Schema. The XML Schema contains the grammatical rules for XML elements and attributes defined for the Control Builder.

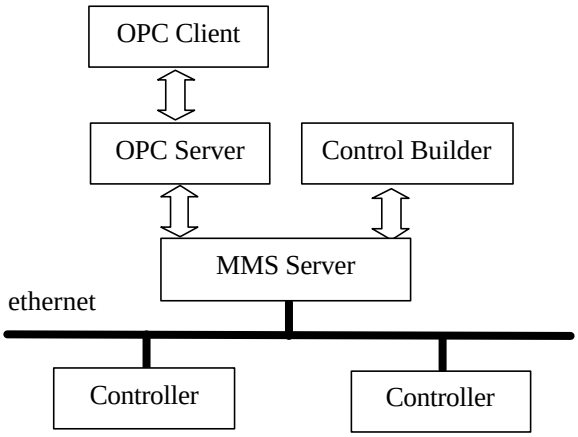


Figure 2.8 OPC server distributes data between controllers and the OPC Client.

3. IEC 61131-3

Overview: *This section gives a brief description of the IEC 61131-3 standard. Each language included in the IEC 61131-3 standard is then presented. Some advantages and drawbacks of each language are mentioned. The chapter also describes in which situations each language is most applicable. The last part gives a description of how programs and execution units are configured in the software model.*

3.1 Introduction

IEC (International Electrotechnical Commission) 61131-3 is a recent international standard intended for design of software for industrial control systems. The standard constitutes of a PLC (Programmable Logic Controller) standard emanating from a desire to improve the programming techniques for industrial control systems. The standard contains a set of graphical and textual languages that have the potential to bring huge benefits for the control system lifetime. Programming languages for industrial control systems improve in a much slower rate compared to other computerized areas. The reason is that these programs may be used in areas where failures risk human safety or give extensive economical losses. Before a new technique is accepted it must first prove that it fulfil these properties. From these primary objectives the IEC 61131-3 has evolved.

3.2 Purpose of IEC 61131-3

IEC 61131-3 was first published in 1993. Before then there was no suitable standard for programming of PLC systems. During the last years before the standard was settled different programming techniques were used to program industrial control applications. This led to that people worked with several different control systems. This resulted in inefficient use of time and money, as system development was not coordinated. These thoughts led to that a new standard for industrial control systems was required. One major objective of the IEC 61131-3 standard was to make the programs more understandable. Besides the programmer, the program should be relatively easy to follow by technicians, plant managers and process engineers. The intention is that IEC 61131-3 based software will have significantly better properties than software generated for conventional PLCs. This can be achieved by new features in the language and by using different programming languages for solving different types of industrial control problems.

3.3 Evolution of IEC 61131-3

The IEC working group started by reviewing commonly used techniques offered by different PLC manufacturers. They then rationalized the languages to create a new set of languages. Through new features in the languages well-structured programs can be developed according to both “bottom-up” and “ top-down” concept. The languages control the data typing in a strong way. During execution there should be support for full control as some parts execute in parallel to other parts or at different times or at different rates. Sequences should have full support. Furthermore, there must be support for grouping data in special structures. It should then be possible to manipulate such structures as single entities in the application. There must also be flexibility to choose any of the included languages to solve different parts of a problem. All parts should then be executed as one compound program. The languages that finally were included in IEC 61131-3 are: Structured Text (ST), Function Block

Diagram (FBD), Ladder Diagram (LD), Instruction List (IL) and Sequential Function Chart (SFC). These languages will be presented subsequently. Example notations for some of the languages in IEC 61131-3 are shown in Figure 3.1. By providing several different languages in IEC 61131-3 the user is allowed to choose the most suitable language for the actual problem. It is potentially possible to solve the same problem using any of the languages even if the degree of difficulty to do it may vary. Some implementers have developed systems that automatically convert from one language to another, e.g. from Ladder Diagram to Structured Text. Today the programs implementing the IEC 61131-3 standard are not character based with simple graphics anymore. Instead they provide graphical interfaces with multiple windows and mouse interaction.

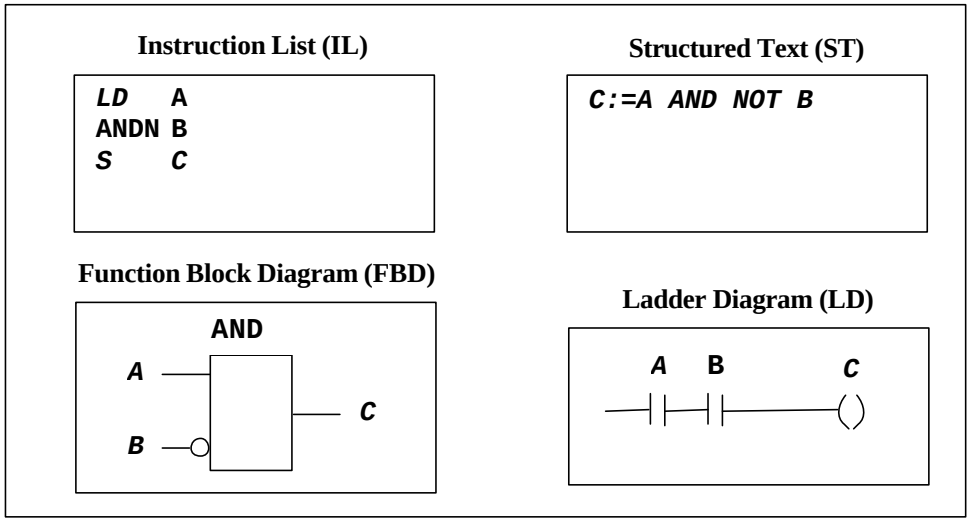


Figure 3.1 Example notations for IEC 61131-3 languages.

3.4 Function Block

One of the main ways to obtain well-structured software in IEC 61131-3 is by using function blocks. A function block consists of: an algorithm that can be executed, external input- and output-parameters and internal variables, see Figure 3.2. Function blocks can be used to implement PID controllers. The algorithm of the function block runs every time the function block is executed. The algorithm uses the current values on the external input parameters and internal variables. The algorithm produces a new set of values for the output parameters. As the function block has internal parameters that are persistent between executions, the function block is actually representing a state during execution. Function blocks are packaged so that they can be reused in the same program or in other programs. Function blocks solve a small problem and both algorithm and data are stored inside the function block. Input and output parameters used in a function block can be used in different programs as long as the types of the variables are correct. Function blocks can be compared to objects in object oriented programming but features like inheritance and polymorphism are not supported. By using function blocks, code is encapsulated and can be handled as one single entity. Function blocks also supports the concepts of information hiding where only the external parameters are possible to reach from the outside.

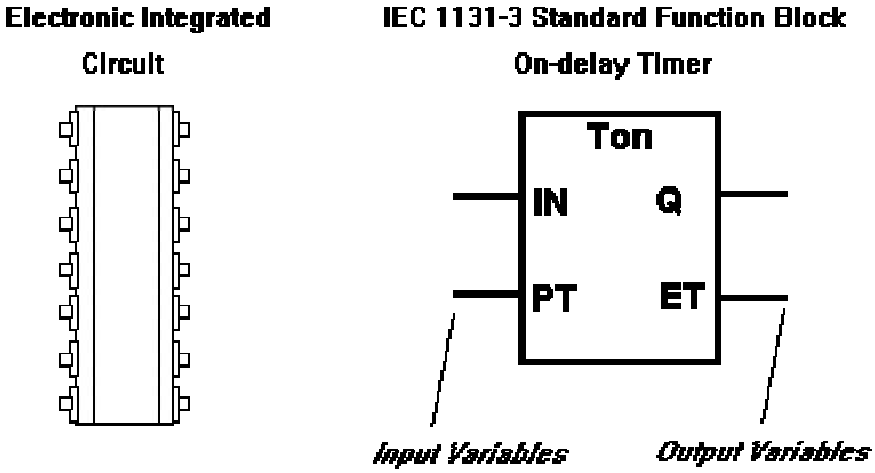


Figure 3.2 Function Blocks compared to ICs.

3.5 Structured Text (ST)

ST is a high level textual language that has been developed for industrial control applications, see Figure 3.3. ST has some similarities to Pascal. ST makes it possible for the programmer to form well-structured programs. ST can be used to express function blocks and programs and can be used in SFC to express the behaviour of steps, actions and transitions.

```
IF SPEED1 > 100.0 THEN
    Flow_Rate := 50.0 + Offset_A1;
ELSE
    Flow_Rate := 100.0;  Steam := ON;
END_IF;
```

Figure 3.3 Structured Text code.

3.6 Function Block Diagram (FBD)

FBD is a graphical language for representing signal and dataflow through function blocks, see Figure 3.4. FBD views a system in terms of the flow of signals between processing elements, like in an electronic circuit diagram. FBD can be used to express the behaviour of function blocks and programs by graphically connect them in diagrams. FBD can also be used in SFC to express the behaviour of steps and transitions. The standard allows signals to be feed back from function block outputs to inputs on previous function blocks in the signal flow.

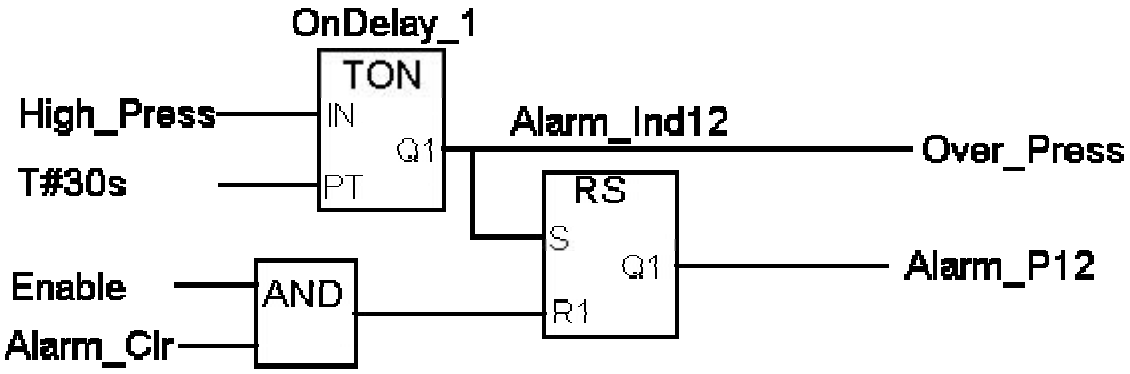


Figure 3.4 Function Block Diagram.

3.7 Ladder Diagram (LD)

LD is a graphical language that is based on ladder relay logic, see Figure 3.5. It has evolved from electrical wiring diagrams that are used to describe relay control schemas. LD can be used to express behaviour in function blocks, programs and also transitions in SFC. The LD is characterized by having a vertical power rail that supplies power through contacts along horizontal rungs. The rungs have contacts that symbolize Boolean variables. PLC ladder rungs are scanned from top to bottom during execution. LD is most suitable for expressing behaviour of programs primarily concerned with combinatorial logic. LD is well suited for small systems and requires only basic programming skills. Faults in the LD can be quickly traced. LD can be very effective in describing digital logic. On the contrary it is difficult to build well-structured programs because of the lack of procedures in the language. There is really no encapsulation of data in the language, which means that identical blocks must be copied many times instead of calling instances of the same procedure. Using LD in closed loop control or sequencing results in very large number of rungs, difficult to program and maintain. There are very limited language validation checks and most inconsistencies can only be detected during run time.

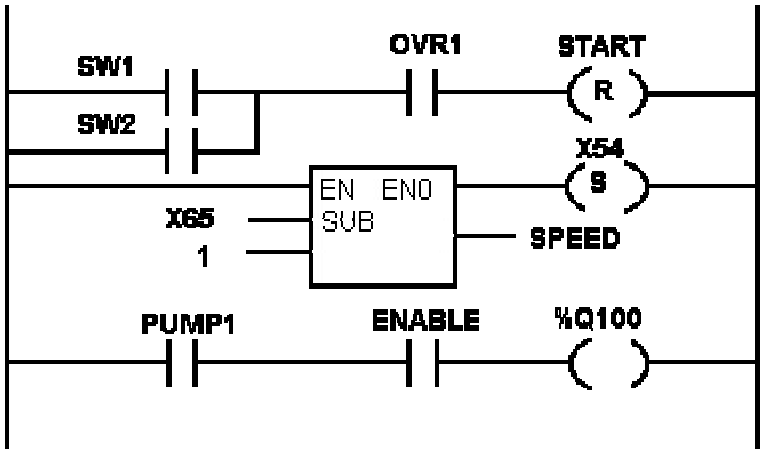


Figure 3.5 Ladder Diagram.

3.8 Instruction List (IL)

IL is a low level assembler like textual language that can be used to express the behaviour of function blocks, programs and transitions in SFC, see Figure 3.6. IL was developed because many manufacturers offered low level languages for PLC programming. The basic structure is very easy to learn. ST is suited for solving small straightforward problems. Some programmers prefer using IL instead of ST because it is easier to implement and sometimes it is possible to download the code into the PLC without intermediate compilation. It is regarded as the language to which all other IEC language can be translated. On the other hand it is very difficult to translate IL to the other IEC languages and it is much harder to follow a program written in IL than in ST. IL is well suited for optimised code performance in critical sections of a program.

```
        LD    R1
        JMPC  RESET
        LD    PRESS_1
        ST    MAX_PRESS
RESET:  LD    0
        ST    A_X43
```

Figure 3.6 Instruction List code.

3.9 Sequential Function Chart (SFC)

SFC is a graphical language for representing the sequential behaviour of control systems, see Figure 3.7. SFC is time- and event-driven. SFC is based on a state-transition formalism similar to what is used in state machines. Describing the behaviour in terms of states and transitions was originally defined in a methodology called Petri-nets [Petri, 1962]. Petri-nets are defined as a formalism for modelling discrete event systems and can be used at all stages of system development. Grafcet has evolved from Petri-nets, which is very close to SFC, has formalism for implementation of sequence control. SFC shows the states of a system and why states are changed. SFC contains series of steps shown as rectangular boxes connected by vertical lines. Each step represents a particular state of the modelled system. The transitions between states is associated with a condition which, when true, causes the preceding step to be deactivated and the step after the transition to be activated. Even if more than one transition condition is true simultaneously, only one path is selected. The flow of control is from top to bottom in sequence but branches to earlier steps can be made. This means that actions within SFC are normally performed in sequence. The nature of state machines is however that actions can be executed in arbitrary order. A step can be associated with actions executed when the step is active. A transition can be described in ST, FBD, or LD. The behaviour of each step can also be described using any of the IEC languages, i.e. ST, FBD, LD, and IL. The behaviour of an action can also be described using SFC itself. This allows complex sequential behaviour of an action to be built up hierarchically. One limit with the hierarchies in SFC is that charts residing inside a macro-step can only be entered and exited at one single point. This limit is avoided in corresponding languages based on state machines e.g. Statecharts. Sub-states residing inside a super state in Statecharts can be entered and exited in arbitrary ways. SFC defines simultaneous sequences executing in parallel. Actions can be divided into stored actions and actions executing while the step is active. Stored actions persist until they are explicitly reset independent of which step is active. The other type of actions only execute while the associated step is active. One main issue of SFC may be to describe the main changes in states occurring over time. Then the control problem can be broken down hierarchically into smaller steps and sequences.

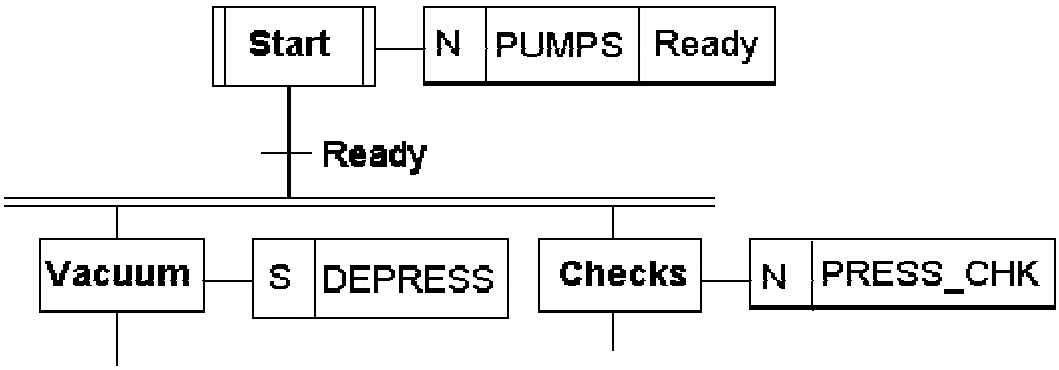


Figure 3.7 Sequential Function Chart (SFC).

3.10 IEC software model

An IEC 61131-3 program can be built from several software elements that are written in any of the five languages. A program can read or write to I/O variables and communicate with other programs. The actual execution is controlled by so called tasks. A task can be configured to control several programs either periodically or when a special triggering event occurs. During one single execution every element within the program executes once. A program cannot execute stand alone, but must be connected to a task that is executing. The standard permits variables to be declared in different software elements that can be either local or global. If the variable is global it can be used in several programs or function blocks residing in different programs.

The software for a special control problem is contained in a configuration. A configuration is defined as a language element corresponding to the programmable controller system, see Figure 3.8. The configuration has as much software as required in one PLC. A configuration is able to communicate with other IEC configurations within different PLC systems. Each configuration consists of one or more so called resources. Each resource has all needed elements to execute a program. A program cannot execute unless it is loaded into a resource. The resource reflects the structure of the PLC in which it resides. Software for a PLC with multiple processors cards may be structured with one resource for each card. To summarize, the standard allows a configuration to contain several resources and each resource is able to support more than one program. Through this it is possible to execute a number of completely independent programs in the PLC simultaneously.

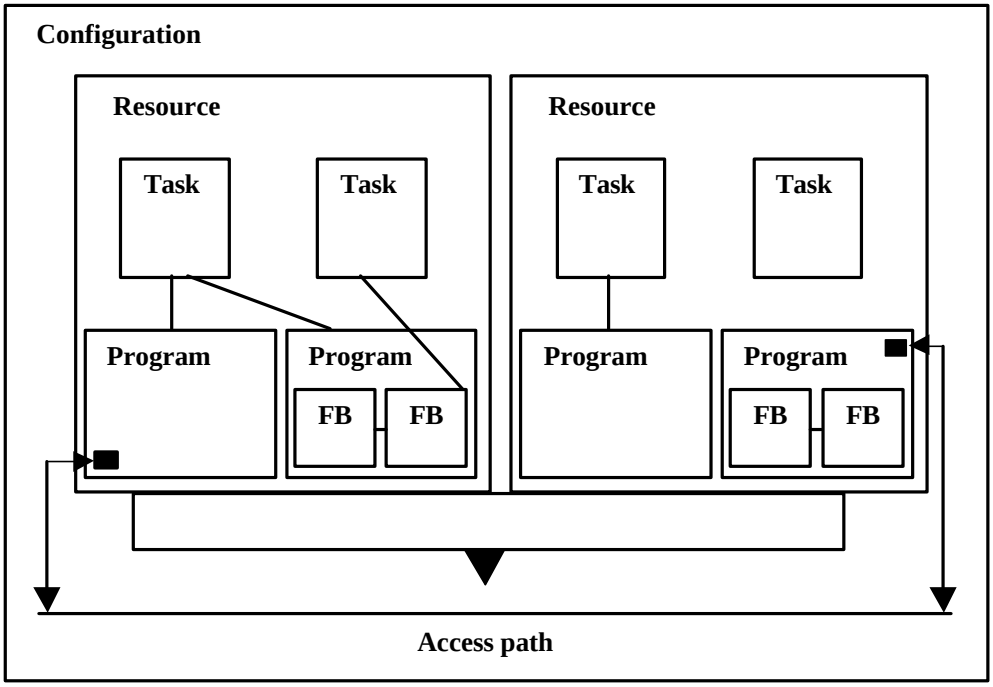


Figure 3.8 IEC software model.

An application requires at least one program but may be divided into several programs. The standard refers to programs, functions and function blocks as program organization units or POUs. POUs can be used many times in different parts of the application. Each copy of a POU is referred to as a function block instance and has its own data area but use the same algorithm. This concept encourages software reuse at higher level, which can be compared to the reuse of function blocks at the lower level. By this the IEC 61131-3 standard provide strong hierarchical design where the programs can be broken down into large function blocks representing major areas. These areas are in turn broken down to more specific function blocks dealing with more specific issues.

Variables can be passed from function blocks outputs to one or more function block inputs. At the higher program level, program inputs is passed to internal function blocks from external variables. Generally program inputs come from physical devices like sensors. Similarly outputs from programs are fed to variables associated with physical output devices. Special variables that need to be accessed from remote devices can be declared as access variables. Access variables may also be read and written by other devices on a communication system. The internal behaviour of POUs can be described using any of the five different IEC languages. Irrespectively of the language used, the variables and data types used by all POUs are described using the same common programming elements. Input and output variables of function blocks will be described in the same way irrespectively of whether the function block is written using Ladder Diagram or Function Block Diagram.

4. Statecharts

Overview: *This chapter describes the main functionality of Statecharts. First a short comparison between Statecharts and ordinary state machines is made. Then the major graphical elements within Statecharts are presented. The semantics is described i.e. how a Statechart model is executed. Finally, some examples are presented.*

4.1 Introduction

A State machine is useful for the design of complex discrete event systems. For given input values, the system will produce, in contrary to a transformational system, different output signals depending on the actual state of the system. Statecharts is based upon the conventional notion of state machines. Statecharts has not only the capability to express as much as both the well known Mealy and Moore state machines, but also additional facilities which soon will be presented. The original state machine is rather limited because it has a flat structure and is sequential in nature. Complicated reactive systems can be described in a more compact way using Statecharts, mainly because of the following properties:

- States are structured in hierarchical levels.
- Sub-states may execute in a concurrent and independent manner.
- Events are broadcasted among the whole Statechart diagram.

Another benefit for the designer, when using Statecharts is the freedom in choosing either a top-down or bottom-up approach when designing the actual Statechart diagram. The intention with the following specification is to describe the overall functionality of Statecharts, particularly in the sense of graphical notation and semantics.

4.2 Basic function

The description of Statecharts, specified in this paper, is based on the definitions made by David Harel, the founder of the language [Harel, 1986]. This is particularly important mentioning in connection with the semantics, because several different definitions exist. However, the one presented here is the one most commonly used [Harel, 1996].

The most important Statechart element is the state, which represents a mode of operation of the system. In order to enable the system to change mode of operation, transitions connected between states are used. The transition specifies which event that causes the mode of operation to change. This situation is shown below in Figure 4.1.

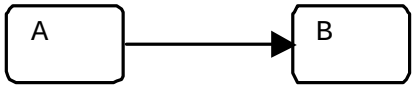


Figure 4.1 Example of states and transitions in Statecharts.

This functionality is common for almost all state machines. Additionally, in Statecharts it is possible to have hierarchies of states. A state on a higher hierarchical level is said to be a super state related to states on lower hierarchical levels, which accordingly are called sub-states. Sub-states with a common super state can either be concurrent or mutually exclusive, depending on how actions are related to each other in the system.

4.3 Elements

Below, the graphical representation of the different elements is presented. The notations follow the specification stated by Harel in his definition of the Statechart language.

4.3.1 States

A state describes a mode of operation of the modelled system. A state can be active or inactive depending on the actual mode of operation of the system. States are graphically described with a rectangle with rounded corners, as shown in Figure 4.2. States may perform actions and activities.

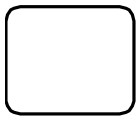


Figure 4.2 Graphical representation of a state.

Actions constitutes of events that are carried out momentarily and have no prolongation in time. This is of course only valid in the ideal case and is not possible to accomplish in a real application. Activities consist of events that take some time to execute, in contrast to actions. States can also be associated with static reactions (SR). SR can be seen as an ordinary action but with the difference that the SR is not coupled to any transition. On the contrary SRs are actually not performed if any transition takes place. The label shown in Figure 4.3 contains properties that are characteristic for states. In this figure `name` is the actual name of the state. Actions following the keyword `entry` are performed when the state is activated. Similarly, actions following the `exit` keyword are performed when the state is deactivated. Activities that should be executed periodically when a state is active are placed after the keyword `throughout`.

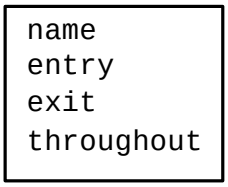


Figure 4.3 Label associated with a state.

4.3.2 Super states

A super state is a state which contains other states called sub-states. Super states can exist in multiple hierarchical levels. A super state is said to be a parent to its contained sub-states whereas each sub-state is called a child of its super state. Graphically a sub-state is shown as a state inside another state as in Figure 4.4. If a sub-state is active, then all of its super states are active at the same time. If a super state is active then one of its children must also be active, assuming a legal decomposition.

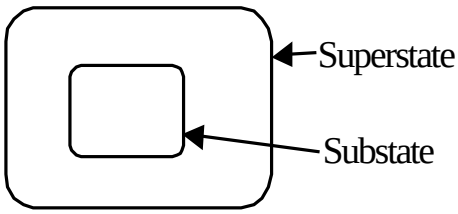


Figure 4.4 Super states and sub-states in Statecharts.

4.3.3 Concurrent (AND) states

Sub-states with a common super state that are active at the same time are said to be concurrent. That is, if any of the concurrent states is exited by the system, then all of the other concurrent states, having the same super state, are also left at the same time. Concurrent states are independent of each other in the sense that the outcome in each state does not depend on which sub-state of the other concurrent states is active. But still, events in concurrent states can depend on actions in other states. Graphically a concurrent super state is shown as an ordinary state but with a dashed line dividing the state into concurrent sub-states, see Figure 4.5.

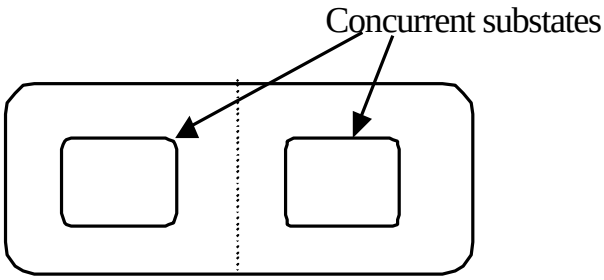


Figure 4.5 Concurrent states.

4.3.4 Exclusive (XOR) States

States located inside the same super state, related as XOR-states, have the property that only one of them can be active at the same time. Mutual exclusive states are shown as ordinary states inside the same super state, as shown in Figure 4.6.

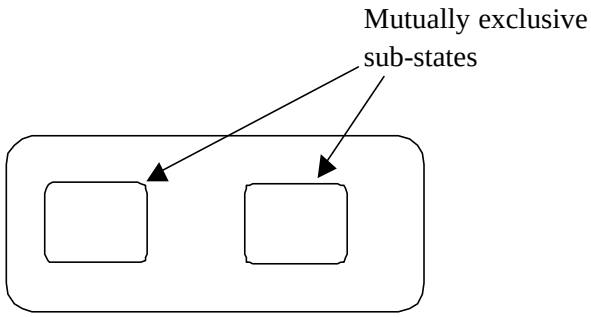


Figure 4.6 Mutually exclusive states.

4.3.5 Condition Connections

Connections have some similarities compared with states. Transitions can be drawn from and to a connection like an ordinary state. The difference is that a transition cannot stay in a connection but must pass through and reach a state before the step is finished. An exception from this is the terminating connection, which is mentioned later on. A condition connection is intended to help the designer to keep the Statechart diagram clean. The connection is represented graphically with a circle with the capital letter C inside, located in a state, as depicted in Figure 4.7. Instead of describing a complicated expression on the transitions in the Statechart diagram, it is possible to connect the transition to a Condition Connection and then specify the description separate from the diagram.

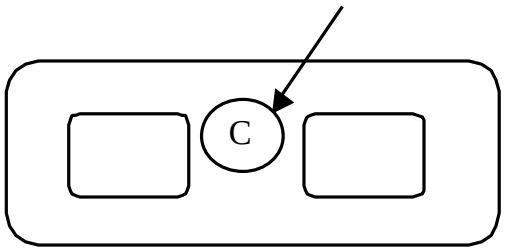


Figure 4.7 Condition connector in Statecharts.

4.3.6 Terminating Connections

A terminating connection is a connection that denotes the termination of the Statechart evaluation. It is graphically denoted as a circle with the capital letter T inside, located inside the actual state. This situation is shown in Figure 4.8. Notice that a transition to a termination connection will end the entire execution, not only the concurrent sub-state in which the system enters a terminating connection.

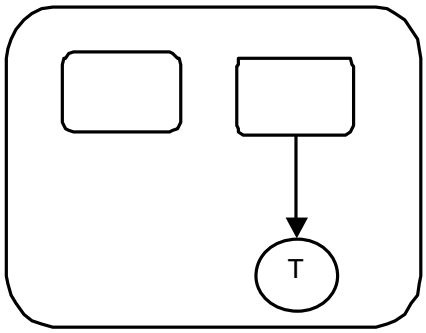


Figure 4.8 Terminating connector.

4.3.7 History Connection

When a super state associated with a history connection is activated, the most recently active sub-state within the super state will be activated. If the super state is activated for the first time, the default transition, which will be mentioned in the subsequent text, will be taken instead. Placing a circle with the capital letter H inside graphically indicates a super state with this property, see Figure 4.9. History is only relevant at the level in which the history symbol appears. To indicate that the history is relevant for all descending levels, an asterisk is marked as a superscript after the letter H.

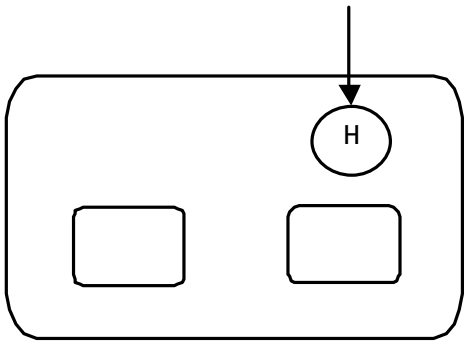


Figure 4.9 History connector.

4.3.8 Transitions

A transition represents the dynamics of a system. Transitions connect states or connections. A compound transition consists of one or several transitions, but must start and end in a state to be legal. The only exception from that rule is that the transition also could end in a terminating connection. A transition is associated with the following label:

```
event[condition]/action
```

Event is the external or internal event, which triggers the transition. The transition is taken if the *condition*, or guard, which is a Boolean expression, is evaluated to true. If and only if the transition is taken, the action after the slash will be generated.

4.3.9 Default transitions

When a super state becomes active it might be possible that it contains several states, as described previously. In that situation the question is then how to define which one of the sub-states should be activated. The solution to that problem is to define a default transition. If no activated transition has a target in any of the sub-states, then the default transition is taken. The graphical notation of a default transition is shown in Figure 4.10.

Figure 4.10 Default transition.

4.3.10 Inner transitions

By making use of an inner transition it is possible to deactivate and activate the same state in the same step. The situation is shown in Figure 4.11. Suppose event d occurs when any of the sub-states inside super state S is active. This causes the inner transition to be fired. Exit-actions associated to the previous active sub-state inside S will be executed as well as the optional transition-action. Finally sub-state B is activated and its enter-actions will execute. Note that super state S is not exited and its exit-actions are consequently never executed. The advantage of using an inner transition is when several sub-states triggers on the same event and reacts in the same way. It is in that case possible to replace all individual transitions with one single inner transition.

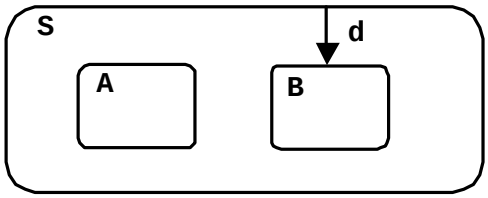


Figure 4.11 Inner transition.

4.3.11 Events

The event denotes a happening that occurs instantaneously in time and is only relevant in one single point in time. In Statecharts, events are used to trigger transitions. The events defined in Statecharts are summarized in Table A.1 in Appendix 1.

4.3.12 Conditions

A condition is a Boolean expression that might have the values true or false. The actual value is only relevant in the time interval between two execution steps and is forgotten later on. Conditions can be composed of several single expressions to form a compound expression. The conditions defined in Statecharts are summarized in Table A.2 in Appendix 1.

4.3.13 Actions

An action is an instantaneous operation, which is either a consequence of a transition or is performed when a state is activated or deactivated. Within a single transition, several actions can be listed. All these actions are, however, performed instantaneously at the same point in time, at least in the ideal case. The actions defined in Statecharts are summarized in a Table A.3 in Appendix 1.

4.4 Semantics

To be able to define the actual behaviour of Statecharts, the semantics from the conventional state machine alone cannot solely be used to get a full description. Additional rules must be adapted to the existing semantics to conform to the additional properties contained in Statecharts. Officially there is no definite semantics. The one described here is the one which is described by the author of the original Statecharts [Harel, 1996]. In the original state machine it was sufficient to know the functional behaviour. This is not enough when dealing with hierarchical and concurrent modelling.

4.4.1 Scope

When a transition takes place from a source to a target, it will probably cross different hierarchical levels in the Statechart diagram. It is then of great importance to define which of the states that are activated, deactivated or unaffected by this happening. For that reason the concept of scope of a transition is introduced. The scope of a transition is the lowest XOR-state, in the hierarchical level, containing all sources and targets involved in the transition. When the transition is taken, all active sub-states within the scope of the transition will potentially be deactivated. All target states of the transition and their successive ancestor super states, up to the scope, will be activated. The scope itself need not to be activated, because it already is active. Consequently the associated actions when entering and leaving states will be performed as a consequence of activating and deactivating states.

4.4.2 Determinism

When executing a step it may happen that several transitions are enabled at the same time. If any two of those enabled transitions have some common source states, the transitions are said to be in conflict with each other. Giving the transition with the scope on highest hierarchical level, priority over the other conflicting transitions, will give a deterministic behaviour in most cases, but not always. If two conflicting transition, with the same scope, are fireable in the same step, Harel gives no deterministic solution. The implementer of Statecharts must invent his own rules to get models with completely deterministic behaviour.

4.4.3 Race Conditions

A transition may perform several actions at the same instant in time ideally. In reality, actions are of course performed in sequence. This combined with the fact that several transitions might possibly be taken in the same step, can lead to a so-called race condition. A race condition happens if the value of an element is changed several times in the same step. Again it is up to the designer of the model to prevent race conditions to arise.

4.4.4 Execution of a step

Systems modelled in Statecharts consider changing state and executing actions at special time instants, called steps. The question is if a change of state really should happen. That is dependent on which of the states that is active and the present values of variables, at the time instant when the step is executed. The execution step determines the sequence of dynamic changes of the system. Or in other words, at the time instant of the execution step, the state and input is read off and a new status and output is generated. The obtained status continues until the next execution step. The steps with their intermediate status are illustrated in Figure 4.12, where time is supposed to increase from left to right.

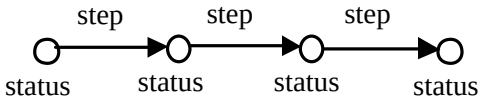
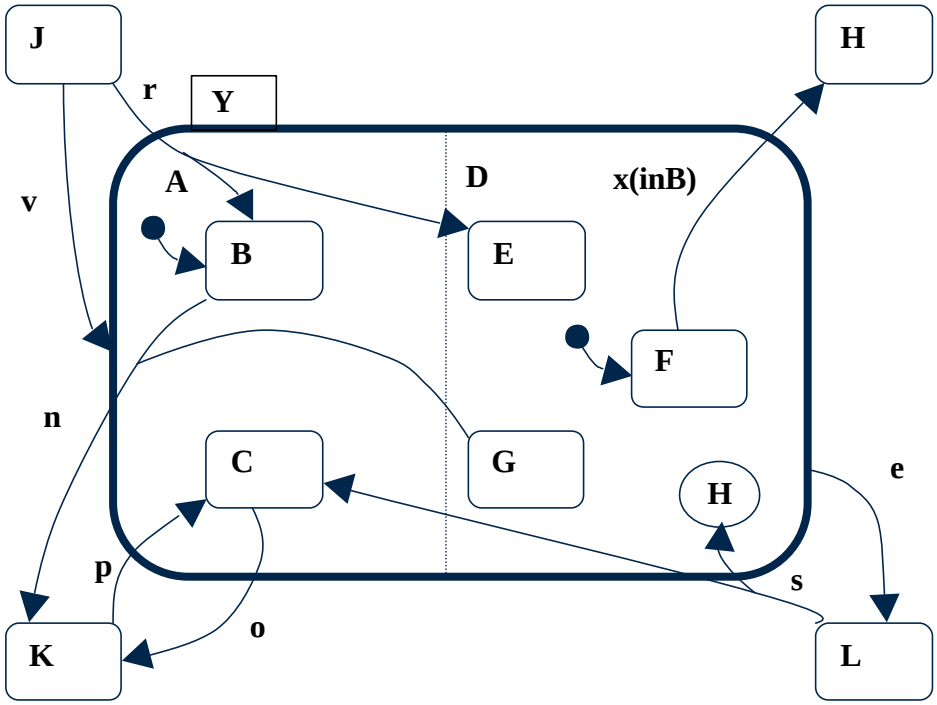


Figure 4.12 Execution of steps.

The fundamental question under consideration is whether changes coming up during the actual step should be sensed in that step or in the next. In this paper the latter approach is chosen, which in fact also was chosen by the founder of Statecharts. Within a given step, say n, the system evaluates generated events, identifies values of parameters and conditions, before execution of step n. When actually executing the step, actions are performed in parallel, that is, the result from one action does not influence another subsequent action, because all actions get their inputs when the step is started. Every change in step n is not sensed until the start of step n+1. For example an event generated by an action in step n is not sensed before step n+1. Note that generated events only live for one step and is not noticed in future steps.

4.5 Example-Statecharts



Figur 4.13 Example of a Statechart model.

Figure 4.13 show a simple example consisting of a Statechart model. Internal transitions in super state Y have been omitted for simplicity. Now a description of what may happen in different situations and under given assumptions follows.

- The split transition from state J illustrates that upon a transition `r`, J is assumed to be active before the transition, the system is entering state (B, E).
- A `p`-event when K is active will cause the system to enter (C, F). C explicitly because of the transition and F because of the default transition.
- An `s` event at L causes a transition to C and the most recently active state in D i.e. either of E, F or G.
- An `n` when the system residing in (B, G) causes an entrance into K.

- An alternative to the last possibility is to replace one of the arrows with a condition as in the x-transition. An x-event in (F, B) causes a transition to H.
- An o transition in C causes the system entering K independent of which of the sub-states in D were active before the event.
- The most general exit from (A, D) is the unconditional transition e entering L.

4.6 Example - Execution of a step

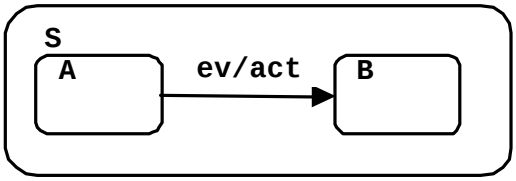


Figure 4.14 Execution of a step.

Figure 4.14 shows a Statechart-model where the system resides in state A initially and event *ev* has been generated in the preceding step. The procedure when executing the step is as follows:

- The transition is activated since state A is active and event *ev* is generated. The transition will be taken.
- The events *exited (A)* and *entered (B)* is generated and will be sensed in the next step.
- Conditions *in (A)* and *in (B)* are evaluated to false and true respectively and are likewise sensed in the next step.
- Actions associated with the exit of state A are generated.
- The action act associated with the transition is generated.
- Actions associated with the entrance of state B are generated.
- All static reactions in super state S are generated if the triggering event is fulfilled. The static reactions are active because state S is active before the step and is not left during the execution of the step.
- All activities associated with the keyword *throughout* in state A are deactivated and those corresponding to state B are activated.

5. JGo

Overview: *In this chapter the JGo package will briefly be presented. A description of the facilities in JGo to group graphical objects together is given. The special structure among JGoObjects residing in JGo is explained. The special features for connecting JGoObjects together with links are also mentioned.*

5.1 Introduction

JGo (version 4.0) from Northwood's Software Corporation is a class package built on the Java 2 (1.2 or 1.3) platform and Swing [Northwoods Software Corporation, 2001]. JGo has a corresponding package conforming to the Microsoft .Net platform called Go.Net. Go.Net has almost identical functionality as JGo even if the implementation of Go.Net use common Microsoft conventions. JGo supports several graphical objects that can be grouped together in a hierarchical structure and visualized in a special view. The mouse can be used to conveniently manipulate graphical objects in JGo. These manipulations are performed in a drag-and-drop fashion. Objects are contained in special documents that are further structured in several layers. This master thesis is especially concerned about connecting nodes by links, which is supported by JGo through special JGoPorts and JGoLinks.

5.2 JGoObject

JGoObjects are the fundamental elements in JGo. Every object visualized in a view originates from such a JGoObject. Each JGoObject has a size and a location in the document that it is included in. JGoObjects have also a special bounding rectangle where different spot locations are positioned. The different spot locations can be referred to when objects are positioned in the document or when they are located relative to other objects in the document. An example of a graphical JGoObject is shown in Figure 5.1. The figure also shows the selection handles of the JGoObject that facilitate resizing of the object by the mouse. It is possible to copy a JGoObject. When the object is copied, a special copyObject method in the object is called. To get a copy where all properties of the original object are copied, the copy method of the original object must be overridden. The copy method must explicitly set all properties in the new object that should be copied from the original one.

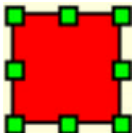


Figure 5.1 JGoObject with selection handles.

5.3 JGoDocument

In JGo, JGoObjects are stored in special documents called JGoDocuments. The document represents a group of JGoObjects to be viewed in a JGoView, see Figure 5.2. This document has a list of layers where all contained graphical elements are included. The layers are ordered in sequence in the list. When the document is created it contains only one layer but the implementer can add arbitrarily many layers. To be able to include an object in a layer, the object must be a subclass of JGoObject, the common ancestor class of all graphical elements in JGo. All elements included in a document can be

visited by iterating the list. It is possible to add an object directly into the document instead of to the layer, if the layer concept is not of interest. Layers and documents have the same methods for manipulating included objects. Special events can be generated from the documents. As in all Java applications it is possible for other objects to register themselves as listeners of these document events. It is possible to listen for events like: a new JGoObject has been inserted, removed or changed.

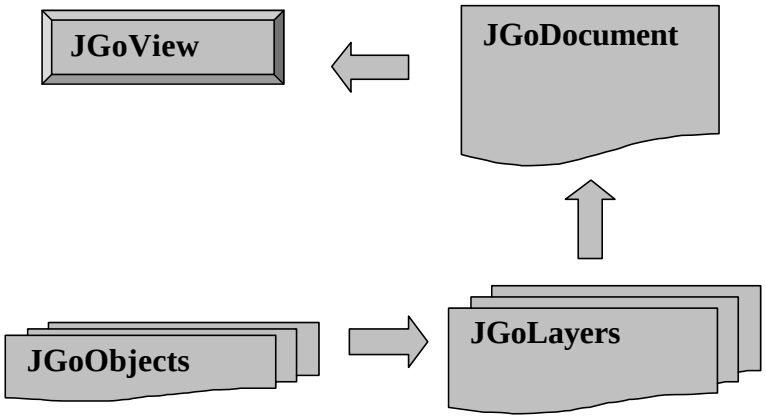


Figure 5.2 Structure of JGoObjects in JGo.

5.4 JGoLayer

It is possible to add JGoObjects directly into the document. But to make better structure among objects in a graph, objects are usually first added into layers that in turn are included in the document. By this, it is possible to separate JGoObjects in a JGoDocument. Layers are ordered in sequence in the document. JGoObject can be visualized in front of other objects by locating them in a layer coming after the layer containing the other objects. When a JGoObject is created it is possible to add it at the beginning or at the end of the layer depending on whether the object is supposed to be visualized in front of or behind other objects in the layer. Layers have special properties that can be used to manipulate all objects contained in the actual layer at the same time. All elements in a layer can for instance be chosen to be visualized or not by setting the visibility property for the actual layer.

5.5 JGoView

To actually show the elements included in a document JGoViews are used. JGoViews provide a window that visualizes the elements in a document and have facilities for the user to interact by clicking on the objects in the view. Because the JGoView is derived from the canvas class in Java, it can be provided with scrollbars. JGoView also supports zooming. The objects contained in the view can be zoomed by changing the scale property of the view. It is also possible to make printouts of all graphical objects in the view by calling the corresponding print method in the view. It is further possible to present the same document in two different views, even if the normal usage is that each document belongs to a single view. Furthermore, JGoView provide facilities like cut-and-paste of JGoObjects like any Microsoft based product. Pressing the left mouse button on the selection handle on the objects boundary, and dragging the handle of the object to the desired size can change the size of an object in the view. By using a special selection class, the view can keep track of objects that gains and looses selection at the same time as selection handles appear accordingly on the screen. JGoObjects can be dragged and copied between different views using custom drag-and-drop behaviour. Objects are copied or moved by the use of serialization; objects that do not support serialization cannot be copied by this facility. JGoView supports generation of different events, in the

same way as JGoObjects can listen to JGoDocument events. When objects in the view are inserted, changed, removed or clicked on by the mouse, events are fired to all registered listeners.

5.6 JGoArea

JGoObjects can be structured in two different ways. Layers are used to structure JGoObjects in the view whereas the JGoArea group JGoObjects as one single object. By using JGoAreas it is possible to add several JGoObjects into a single area. The area can then be manipulated as one single object. Since a JGoArea also is a JGoObject, several JGoAreas can in turn be grouped together to constitute another compound object. This grouping can be performed up to arbitrary levels. Individual objects within a JGoArea do not have individual bounding rectangles. Instead the compound area has one bounding rectangle surrounding all objects included in the area.

5.7 JGoPort

JGoPorts are used as a connection point for JGoLinks. The port has a special list in which each link that is connected to the port is included. From a port it is possible to iterate through all connected links to get to all ports at the other end of the link. The graphical representation of the port may appear in different ways. The port may be invisible, look like an ellipse, or may have an appearance as any optional graphical object. In Figure 5.3 two JGoObjects are shown. Their JGoPorts constitute of the black spots in the centre of the ellipses. The JGoPorts make it possible to connect the objects together. Each port can be individually configured to either be a valid source port, a valid destination port or both, for a potential link. It is even possible to specify that a particular pair of ports cannot have a link connection at all. In some cases it may be convenient to connect a link to different points on the port depending on if the port is a source- or destination-port. These are some of the properties that can be configured in the JGoPort class.

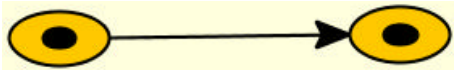


Figure 5.3 Connection between two JGoObjects.

5.8 JGoLink

A JGoLink connects two JGoPorts. It is potentially possible to connect a link between one and the same port. Although that is an unusual occurrence it is really implemented in the actual application. When the link is created it must be told which source- and destination-port it should connect. As the link knows to which source- and destination port it connects it is possible to iterate from port to port through intermediate links in the whole graph. The graphical representation of a link can in simple cases just be a straight line connecting to the ports. The link path may also consist of alternating horizontal and vertical paths by specifying a special orthogonal property in the link class. It is possible for the implementer to define special tailor-made appearances of the JGoLink, by overriding the method for calculating the stroke of the links. Additionally it is possible to add arrowheads to the links to achieve directed graphs. This is actually done in the Statechart Editor Prototype.

5.9 User Interaction

Clicking with the mouse button on a JGoPort and dragging the mouse to the desired destination port can create a JGoLink. When clicking on a port, the port checks to see if the actual port is a valid

source for a link. If so, a temporary port and a link connecting the source port and the temporary port are created. While the user drag around the mouse, the temporary port and the link follows the mouse. The view checks at the same time if there is any valid destination port within the so-called gravity distance from the actual mouse location. The gravity distance is really a property of the destination port. If the mouse is released in this dragging mode the temporary port automatically snaps to the location of the first valid destination port within the port gravity distance. If there is no valid destination port within the gravity distance when the mouse is released, the link and the temporary port is removed and no connection is created. When a connected JGoArea with its included port is dragged around in the view, the link automatically follows the node. This facility is provided by the JGo package and need not be explicitly implementation by the user.

It is further possible to insert text in a view by using JGoText objects, which as all objects in JGo are derived from the JGoObject class. The user can edit these text objects during run-time just by single clicking on the objects with the mouse.

JGo also provide a possibility to implement undo and redo actions. This facility acts as a rescuer when the user wants to reverse unintended changes. These actions are not used in the Statechart Editor Prototype but are briefly mentioned anyway. Swing supports redo and undo in Java but to support it in JGo some additional implementations must be made to get full performance. The support in JGo is valid for JGoDocuments and the objects contained in the documents. The undo managing slows down the editing because every change must be followed by a corresponding action from the undo manager. Therefore it is worth to consider if the undo facility is really needed when high performance is demanded. Performance needs not to be considered if the application contains fewer than about hundred of JGoObjects.

6. Statechart Editor Prototype

Overview: *This chapter describes the Statechart Editor Prototype. First an overall description of the user interface is given. The actual implementation of Statecharts in this prototype is then examined. The notion of hierarchies in Statecharts is also described. Different ways of presenting the model in the view are then discussed. Facilities used from the JGo package in the graphical interface is also presented. Then the textual XML representation of a graphical model is described. The last part of this section deals with the design of the prototype implementation.*

6.1 Introduction

The Statechart Editor Prototype is a standalone application with respect to the graphical design of Statechart diagrams. When the designer wants to execute a Statechart model, the prototype is no longer a separate application but works in combination with the Control Builder environment. The designed graphical model is actually executed in the Control ^{IT} and the progress of parameters and variables can be followed there. It is however possible to follow this progress also in the prototype. All parameter- and variable values are scanned at regular time intervals from the executing controller and updated in the prototype. Hence, all needed functionality to edit and execute a Statechart model can be controlled directly from this Statechart Editor Prototype.

6.2 Overall Description

The Statechart Editor Prototype is implemented in the Java 2 platform, version: 1.3.1. The JGo package is used to provide facilities for connecting graphical objects together. The prototype is a graphical editor facilitating design of Statechart models, see Figure 6.1. The editor consists of four different parts: the editing area, the toolbox, the table for parameter declaration and the main menu bar. The editing area is a JGoView in which it is possible to create Statechart models using states and transitions. The toolbox or the palette is also a JGoView but it is not possible to edit in this area. The palette instead acts as a toolbox, providing states and default states to be easily dragged by the mouse into the editing area. In the parameter table, parameters used in the code blocks associated with the states and transitions can be declared. At the top of the prototype is the menu bar, providing different facilities like saving a model in a file.

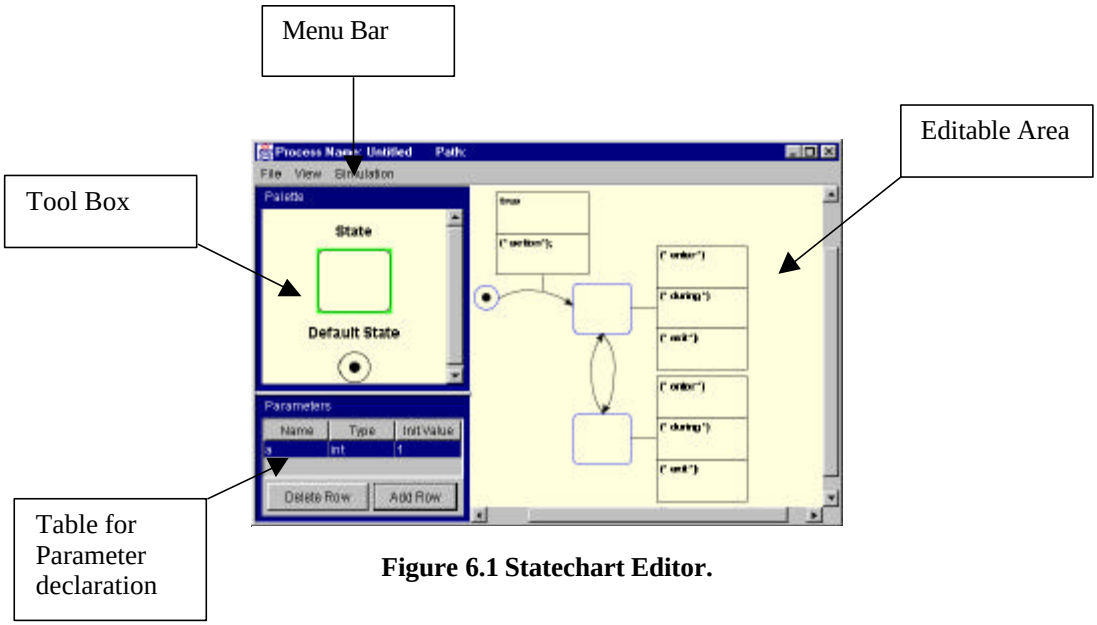


Figure 6.1 Statechart Editor.

When the Statechart Editor is started, the editable area is empty. The user is intended to drag objects from the toolbox into the editing area. The objects can be connected with transitions by pressing the left mouse button on a source object. If the mouse is dragged to the destination object and the mouse button is released, a transition is automatically created. The transition is connected between the source and the destination state. The transition is provided with an arrow at one end, indicating in what direction the transition can be taken.

Programmable code can be associated to each state in three different code blocks. These code blocks are executed when the state is active, activated or deactivated, respectively. Each transition is associated with two code blocks. The first code block constitutes a condition or trigger. If the condition evaluates to true the transition fires and deactivates the source state of the transition, while the destination state is activated. The second code block associated with the transition is an action block that executes if the transition is taken i.e. when the condition is true.

When the Statechart Editor is exited, the model existing in the editing area is discarded. If the designer wants to use an edited model later, the model can be stored in a file. The graphical model is represented as an XML-document when stored, see Figure 6.2. Through this it is possible to understand how the model looks like directly from the document, because it is represented as fully understandable text in the stored file. The stored XML-document can, at a later occasion, be restored into the graphical editor again for further modification.

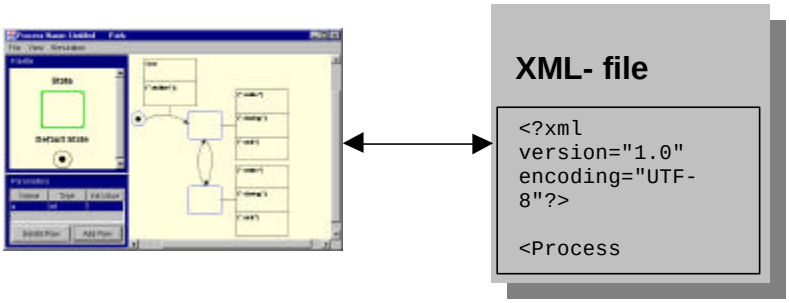


Figure 6.2 Graphical representation is translated into an XML-document.

When the designer eventually wants to simulate the graphical model, the execution of the graphical model is first translated into Structured Text code. The generated code is then transferred into the Control Builder environment where the model can be executed. To simulate the graphical model, it must first be loaded into the editing area. It is not possible to simulate a model directly from the stored XML-file describing the graphical representation.

6.3 Statecharts in the prototype

The Statechart Editor provides basically three different graphical elements. The elements are the State, the DefaultState and the Transition. Sometimes States are called StateNodes and DefaultStates are called DefaultNodes. Transitions are sometimes abbreviated as links. The StateNode and the DefaultNode are provided in the palette but not the transition. The reason is that the transitions are automatically created when the mouse is dragged between two nodes in the view. According to the section describing the Statechart language, there exist additional elements in the language. In this prototype it is only possible to implement the most common functionality within Statecharts. The history property is for instance not implemented. Neither is the terminating states used to stop the execution. However, there is nothing preventing from including additional facilities, starting from the current prototype.

6.3.1 Notation – StateNode

The StateNode is shown in Figure 6.3.

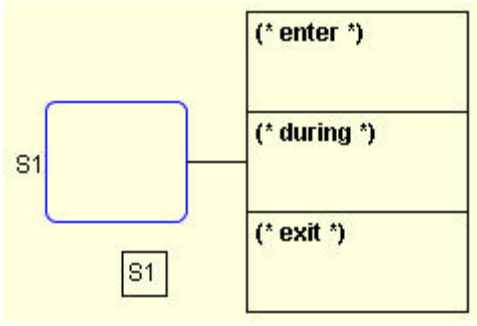


Figure 6.3 StateNode and associated action blocks.

The actual StateNode only consists of the rectangle with rounded corners. Each StateNode has a name. To the left of the StateNode the name label is displayed if it is set as visible in the StateNode property dialog box. When StateNodes are inserted into the editing area they get default names. The default name starts with the letter “S” followed by a number. The first inserted StateNode get the name “S1”, the next StateNode get the name ” S2” and so on. The designer can change the name by single clicking on the default name with the left mouse button. Arbitrary names can be used as long as each StateNode has a unique name. The label is also showed as an ordinary tool tip when the mouse pointer is located inside the StateNode area. The three rectangles connected to the right of the StateNode contain the different action blocks associated to the StateNode. Structured Text code can be edited into each of the associated code blocks. The editing mode is invoked by single clicking inside any of the rectangular areas. The code must follow the syntactical rules according to the Structured Text language, in the same way as if the code was edited in an ordinary application. Each line of code must for instance end with a semicolon. Code contained in the uppermost rectangle is executed when the StateNode is activated during execution. The code in the middle rectangle is executed while the state is active and the code in the bottom rectangle is executed when the state is deactivated. When a StateNode is inserted, neither the action blocks nor the name of the StateNode are shown. Just double

clicking on the StateNode with the left mouse button makes the action blocks visible to the right of the StateNode, as is shown above. The action blocks can be hidden again by another double click. The properties of a StateNodes can be edited by first clicking with the right mouse button on the StateNode. A popup menu is then displayed, see Figure 6.4.

Figure 6.4 Popup menu of StateNode.

From this popup menu it is possible to choose whether the action blocks and/or the state label should be shown or not. If the Edit State item is selected, a dialog box pops up, see Figure 6.5.

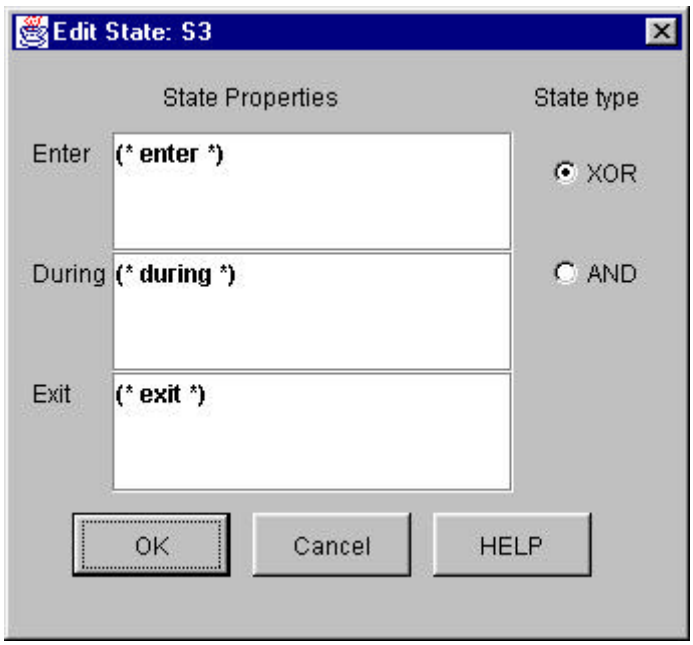


Figure 6.5 StateNode property dialog.

This dialog box provides the possibility to edit the action blocks. These are of course the same action blocks that were shown in Figure 6.3. The only difference is that the code blocks in the dialog box are provided with scroll bars, so that arbitrary much code can be edited. From the StateNode dialog box it is also possible to choose whether the StateNode should be of AND- or XOR type. If the StateNode is chosen to be of AND type, all potentially contained sub-states are activated simultaneously when the StateNode is activated. If the StateNode is of XOR type, the default case, just one of the topmost sub-states is activated when the StateNode is activated.

6.3.2 Notation - DefaultNode

When several StateNodes exist in a model, the question is which one of them that should be activated when the execution starts. The DefaultNode, the second object in the toolbox, solve that problem. When the execution starts, the system always resides in the DefaultNodes. StateNodes connected to a transition from such a DefaultNode are activated after the first step after start-up. To be correct there is actually a small difference in how StateNodes connected to DefaultNodes are activated depending on

whether the StateNode is located in the outermost hierarchical level or not. This will be discussed later on. In some implementations the StateNode connected to a DefaultNode is activated immediately at start-up, while in others it is activated after the first step. The definition of Statecharts is vague concerning that matter. In this prototype the graphical representation of a DefaultNode is shown in Figure 6.6. The DefaultNode has the same default naming conventions as StateNodes, except that the default name of DefaultNodes starts with the letter “D”.

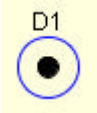


Figure 6.6 DefaultNode.

6.3.3 Notation - Transition

A transition indicates a change of state within a system. A transition does not exist on its own, but only when connected between two states, see Figure 6.7.

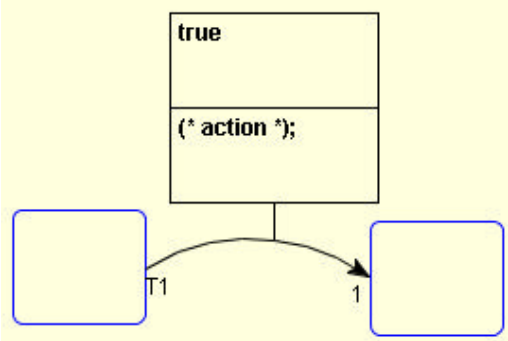


Figure 6.7 Transition connecting two states.

The transition starts in a source node and ends in a destination node. It is possible to select one of the connected nodes and drag it to any desired location in the editing area. The connected transition will automatically follow the moved node, still being connected between the two nodes. Each transition is associated with two code blocks. The topmost code block is a condition or trigger, consisting of an expression in Structured Text code. Note that a condition code block does not end with a semicolon as the action code block does. The condition code is equivalent to the condition following an ordinary if statement in Structured Text. A transition fires when the source node is active and when the condition in the condition code block evaluates to true. When a transition fires, the actions in the bottom code block of the transition are executed. At the same time, the source node is deactivated and the destination node is activated. The name of the transition is shown at the starting end of the transition. The number at the other of the transition, next to the arrow, indicates a priority that the transition has. This priority is only considered in cases when conflicting transitions are fireable in the same step. The transition with the greatest priority number is the only transition taken. Priorities are not included in the original Statechart definition. Using priorities was considered the only way to achieve completely deterministic behaviour during execution. The properties of Transitions can be set in the same way as for the StateNode, see Figure 6.8. The transition has only the two mentioned code blocks. The priority of the transition can be set to any desired value. The first inserted link has the name T1 and priority 1 by default. The next link has the name T2 and priority 2 and so on.

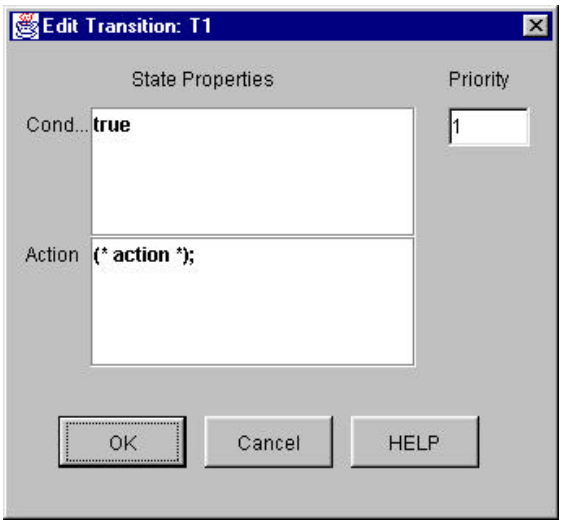


Figure 6.8 Transition property dialog.

6.3.4 Hierarchies

All basic types of graphical elements, included in this prototype, have now been presented. It may seem that the width of expressiveness could not be overwhelming just by using these few symbols. The strength of Statecharts is that hierarchies of StateNodes can be built. In this prototype it is possible to insert StateNodes and DefaultNodes as sub-states into any StateNode. The StateNode in which the nodes are inserted is then called a super state, while the inserted nodes are called sub-states. New nodes can in turn be inserted into the newly inserted sub-states. StateNodes can consequently have both a super state and sub-states. The hierarchy of nodes can generally be made up to arbitrary depth. However, in this prototype the hierarchical levels are limited to three. However, the number of levels can easily be extended if desired.

The StateNode is the only object that can contain other objects i.e. form a super state. Each graphical object references the immediate containing super state. The reference is set to null if the object has no super state. Similarly, each StateNode has a vector where all potential sub-state objects are included. Each time an object is inserted into the editing area, the entire hierarchy of objects are revised. The updateHierarchy method in the SCEditor, performs this issue. The updateHierarchy method checks for each object in the editing area if the object is contained in any other StateNode. By contained means that the graphical symbol of an object is completely contained inside some StateNode symbol. If so, the super state attribute of the sub-state object is set to the super state. Correspondingly, the sub-state is added into the sub-state vector of the containing super state object.

A super state and its sub-states can be related in two different ways. If the super state is of XOR type, only one of its contained sub-states is active when the super state is active. The graphical representation of this XOR relationship, as it looks like in the prototype, is shown in Figure 6.9.

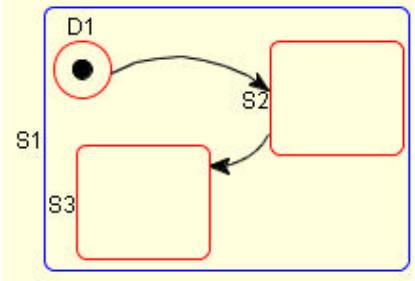


Figure 6.9 Super state of XOR-type with sub-states.

The StateNode (S1) is super state to the two included StateNodes (S2 and S3) as well as to the DefaultNode (D1). When S1 is activated, S2 will also be activated because it is connected to a DefaultNode. There must exist exactly one DefaultNode inside a non-empty state of XOR type. On the other hand, there may not be any DefaultNode inside an otherwise empty StateNode. There is a small distinction between DefaultNodes located inside super states compared with the outermost DefaultNode. The outermost DefaultNode has theoretically the root as super state but this connection is never used during modelling. The outermost DefaultNode is deactivated after the first step, while DefaultNodes located inside "ordinary" super states (not the root) are always active. This distinction is only interesting during execution. If the DefaultNode on the outermost hierarchical level has not been deactivated after the first step, the connected default transition would fire in each step, which is not desirable.

The other type of relation between super states and sub-states is when the super state is of AND-type. The graphical representation of this AND relation in the prototype, is shown in Figure 6.10.

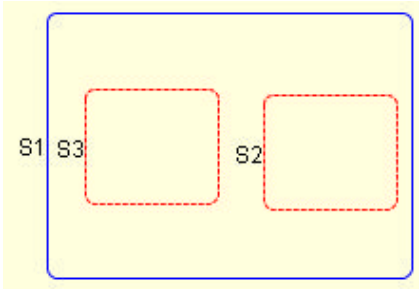


Figure 6.10 Super state of AND-type with sub-states.

The dashed sub-states in Figure 6.10 (S2 and S3) indicate that their super state is of AND type. In addition there exist no DefaultNode inside the super state, as was mandatory when the XOR super state was described. The explanation to that is that there is no need for any DefaultNode, because all included sub-states are activated anyway. Thus, both S2 and S3 are activated when S1 is activated. The link between S2 and S3 was also taken away, compared to Figure 6.9.

6.3.5 Views

When a model requires just a few nodes and the complexity is low, the graphical Statechart model would be rather easy to overview. However, when the system is complex, the graphical model tends to be blurred and hard to survey. Another benefit with Statecharts compared to basic state diagrams is that the inherent hierarchical structure within Statecharts can be utilized. By the use of Statecharts, it is possible to design models either according to the bottom-up or the top-down concept. This concept is based on the fact that all hierarchical levels within a model are not interesting to study all the time when modelling. The prototype facilitates the user to choose the most suitable abstraction level dynamically during model development. In other words, it is possible to just display objects down to a user defined hierarchical level. Figure 6.11 shows the items included in the View menu in the main menu bar.

Figure 6.11 View Menu.

By clicking on the Abstraction Level-Up item, objects currently located in the lowest hierarchical level are temporarily hidden. Combining this abstraction with the zoom function, makes it possible to study the most interesting abstraction level for the actual moment.

Another item in the View menu is the Overview item. By selecting this a new window displaying the entire editing area is shown, see Figure 6.12. Inside this Overview a rectangle is displayed. This rectangle indicates the part of the total editable area that is actually shown in the editable window. By using the mouse it is possible to drag around this rectangle while the interior of the rectangle is shown in the editing area in the “real” view. This overview window can be a valuable tool when the model is large.

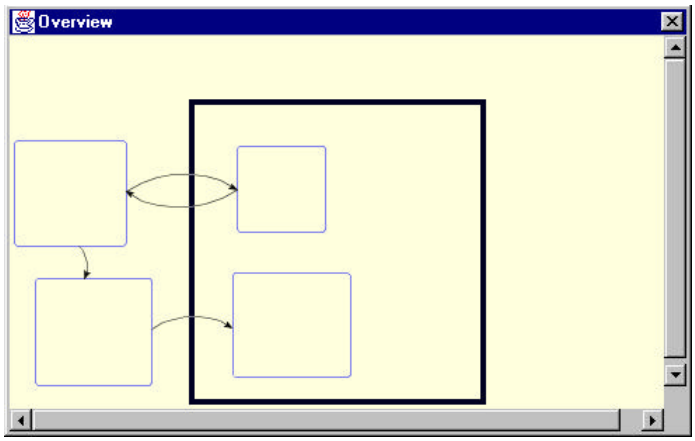


Figure 6.12 Overview window.

6.3.6 Execution

When the designer wants to execute a model, the first requirement is that the model is located in the editing area of the prototype. The model cannot be executed in the prototype itself. The model must first be translated into Structured Text code that then is transferred into the Control Builder for execution in a Soft Controller. The code generation as well as the code transfer are managed from the prototype, by clicking on the Download and Simulate menu choice in the Simulate menu, see Figure 6.13.

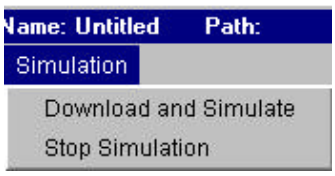


Figure 6.13 Simulate menu.

When the simulation starts, the execution in the controller could be followed directly in the prototype. Values of parameters, located in the table, are updated as the execution in the controller progresses. States are highlighted directly in the editing view in real time. Links that are fired are also highlighted, making it easier for the user to track the behaviour of the system. The execution can be stopped at any time by clicking on the Stop Simulation menu choice in the simulate menu. Chapter 8 describes more about the connection between the prototype and the controller during execution.

It is also possible to only generate executable Structured Text without execution by selecting the Save ST-Code menu choice in the file menu, shown in Figure 6.14. The executable Structured Text code is

then generated from the model residing in the editing area of the prototype. The generated code can then be stored in a file.

Figure 6.14 File menu.

6.4 JGo

The JGo package is extensively used in the prototype. When objects are created, connected and viewed in the different areas, JGo is used. When looking at the Statechart editor in Figure 6.1, all graphics is based on JGo and Swing. The editable area is an editable JGoView. The view display the objects residing in a single JGoDocument. This document is in turn divided into three layers. A layer coming after another layer in sequence in the document appears in front of that layer in the view. Elements are then added to the layers instead of the document. All elements in a specific layer are then visualized in front of or behind elements in other layers. Initially a JGoDocument has only one layer but arbitrary many layers can be added to the document. In this prototype three layers are used. The intention is merely to demonstrate the possibilities, not to give support for huge applications. The layers in the document are used to structure the graphical objects residing in different hierarchical levels. The first layer contains elements located at the highest hierarchical level in the Statechart model and the next layer contains objects located next to the highest hierarchical level. The structuring in different layers is performed at the same time as the objects are inserted into the editing area, in the updateHierarchy method in the SCEditor class. When the structure has been obtained it is easy to make objects in particular hierarchical levels invisible, by one single method call in the actual layer object. The toolbox, or the palette is a JGoView. In this view it is not possible to perform any editing.

The StateNode and DefaultNode are both two kinds of JGoAreas. In both areas one JGoPort is included to make it possible to connect the node to links. In StateNodes in contrast to DefaultNodes the JGoPort is not visible, yet it is located in the centre of the StateNode. Although the port is invisible it is possible to start and end links at them simply by clicking on them and drag to a destination. One difference between StateNodes and DefaultNodes is that the StateNode may act as both source- and destination point for a potential link, while the DefaultNode just can act as source. Such properties are easy configured in the respective JGoPort. Another difference is that the DefaultNode only may have one single link, while the StateNode can connect arbitrary many links in both directions.

In JGo it is possible to configure the visual layout of the links to be either automatic or user defined. The automatic layout makes the link to be drawn orthogonal i.e. alternating horizontally and vertically in the view. In this prototype the layout has been chosen to be user defined. This means that the layout of the links is user defined from the prototype point of view. From the user point of view the layout is automatic. Neither the connection point of the link at a specific node nor the actual shape of the link can be changed without moving the connected nodes. The links have additionally been modified from

the default JGoLink class. To begin with, an arrowhead is created at one end of the link, to make the graph directed. This is easily achieved by a simple method call on the actual link object.

The standard link is just represented by a straight line. To connect two nodes by one link in both directions, the links would overlap if they where displayed as straight lines. It seemed necessary to separate them in some way. By making the links a little rounded it would be possible to separate two links connected in both directions. That issue was not so easy to implement but it was actually done by overriding the method that calculates the actual stroke between two nodes. In that method several points are inserted in a specific way forming the link as a smooth curve from the source to the destination. In the original definition of Statecharts links could start in the same StateNode as it ends in. JGo does not generally provide this possibility, if the link layout is user defined. Hence this functionality is explicitly implemented in the stroke calculation method, included in the link object.

6.5 XML-representation

The graphical representation in the Statechart Editor can be stored in a file using the save menu item shown in figure 6.14. A file that has been stored in this way can be reloaded again into the editor for further modification, or perhaps for execution in the Control Builder. When the model is stored, all graphical objects and their properties are first translated into readable text in a special XML-format. Since this application allow saving in XML-format only, the filenames must end with ".xml"; otherwise the whole saving operation is rejected. A special java API for XML-processing (JAXP) is used, which enables applications to easily parse and transform XML-documents.

When the graphical model should be stored, a new instance of a DocumentBuilderFactory is created, see sample code below.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.newDocument();

Element process = (Element) document.createElement(processTag);
        process.setAttribute("name",getName());
        ...
document.appendChild(process);
```

Then a new XML-document is created from a special document builder. All these facilities are provided by the JAXP package. It is then possible to create tags. The first tag in this document is the process tag that will contain attributes characteristic for the entire graphical model. The process tag constitutes the root tag of the XML-document. Attributes can then easily be included into the created tag. When all attributes are inserted into the process tag, the tag together with its included attributes are finally appended to the XML-document. The result of the above operation gives the XML-file shown below:

```
<Process name="Untitled" location="C:\Gert-Ola\ABB\Statecharts\model1.xml"
stateCounter="2" linkCounter="3" defaultNodeCounter="1" x="0" y="0">
</Process>
```

Each graphical object is then added to the document as a separate tag. All objects are added as children of the Process tag, the root tag. As an example the process of appending a StateNode looks like this:

```
Element state = (Element)document.createElement(stateTag);

state.setAttribute("id",node.getId());
...
state.setAttribute("exitActionText",node.getExitActionString());


process.appendChild(state);
```

Using the createElement method as before creates the state tag. Then attributes are created associated to the newly created tag. The state tag is now appended to the process tag instead of to the document. This makes the state to be a child of the process tag. All other StateNodes residing in the model are appended to the document in this way. Then DefaultNodes and Transitions are appended to the document together with their properties in the same way as StateNodes. The final properties that have to be stored in the document are the parameters declared in the parameter table.

The stored properties of each object type are shown in the following tables. An example of how an entire XML-document looks like can be studied in Appendix 2.

Property	Description
Process	Enclosing tag for process properties
location	Path to the file containing this XML-document
stateCounter	Keeps track of the number of StateNodes
linkCounter	Keeps track of the number of SCLink
defaultNodeCounter	Keeps track of the number of DefaultNodes

Table 6.1 Properties of the Process Tag in the XML-document.

Property	Description
StateNode	Enclosing tag for StateNode properties
id	The name of the StateNode
type	The StateNode type, XOR or AND
superState	The super state of the StateNode
x	x-coordinate of the top left corner of the StateNode in the view
y	Corresponding y-coordinate
width	Width of the StateNode rectangle in the view
height	Height of the StateNode rectangle in the view
enterActionText	Code block executed when the StateNode is activated
duringActionText	Code block executed when the StateNode is active
exitActionText	Code block executed when the StateNode is deactivated

Table 6.2 Properties of the StateNode Tag in the XML-document.

Property	Description
DefaultNode	Enclosing tag for DefaultNode properties
id	The name of the DefaultNode
x	x-coordinate of the top left corner of the DefaultNode in the view
y	Corresponding y-coordinate

Table 6.3 Properties of the DefaultNode Tag in the XML-document.

Property	Description
Link	Enclosing tag for SCLink properties
id	The name of the SCLink
from	Source node of the SCLink
to	Destination node of the SCLink
conditionText	Condition code block
actionText	Action code block
priority	Explicit assigned priority

Table 6.4 Properties of the SCLink Tag in the XML-document

Property	Description
Parameter	Enclosing tag for parameter properties
name	The name of the parameter
type	The type of the parameter
initValue	The value of the parameter

Table 6.5 Properties of the Parameter Tag in the XML-document.

6.6 Design of the prototype

The major class in the Statechart Editor Prototype is the SCEditor, which controls the execution of the program. The user interface is created and initiated to react on user actions. The SCEditor class is also responsible for the code generation from a given model. Another important class is SCDocument, which holds all elements included in the designed model. To make created elements visible they are shown in an SCView, which in addition reacts upon mouse clicks on its included objects.

6.6.1 Overview

The block diagram in Figure 6.15 shows the relationship between different classes in the prototype. Each block in the diagram represents several classes and every two classes within the same block have more or less a close connection.

The main block in the figure is the SCEditor block. It is responsible for the user interface and the interaction between the user and the prototype. The SCEditor block is also responsible for the communication with the Control ^{IT}. SCTree is the data structure also found inside the SCEditor block. Links and Connectors are inserted into the SCTree when the prototype is generating code. The tree is parsed from top to bottom while code is generated and implicit priority is achieved among states and links.

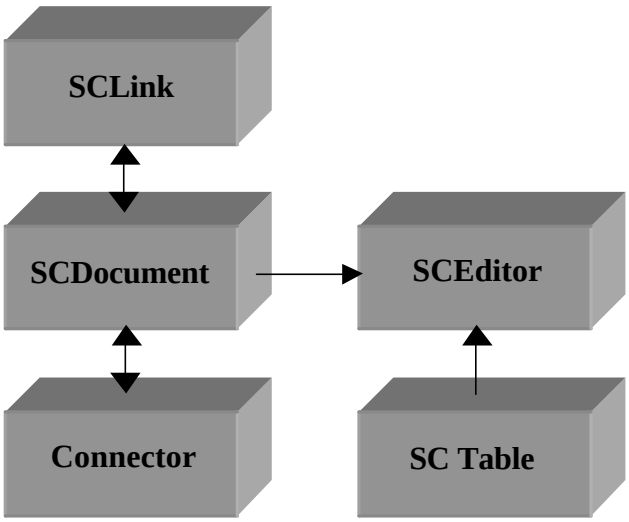


Figure 6.15 Block diagram of the overall system design.

The SCDocument holds all elements in the designed model and can translate all elements and their properties into XML-code to be saved in a file. SCLink is a directed graph connecting two connectors and represents the transition between two states of the model. A Connector is a common name for StateNodes and DefaultNodes in this prototype, but other connectors exist in Statechart to. Common for all connectors is that they facilitate a connection point for one or more links. SCTable enables the designer to declare parameters that are used in the code blocks.

6.6.2 Class Diagrams

6.6.2.1 SCEditor

The SCEditor block in Figure 6.15 consists principally of the classes shown in Figure 6.16. Directly used classes from the Java standard edition and the JGo package are filled with grey colour. Program execution is controlled from the SCEditor class, which is the most important and extensive class. SCPalette is the graphical toolbox, from which states may be dragged into the SCView. SCView is the editing area where the actual model is built. SCView maintains typical cut and paste behaviour when states and links are edited. Parameters used in the design must be specified in the SCTable, which together with the SCPalette are inserted in a standard JSplitPane for visualization. The SCEditor consists further of a JMenuBar at the top of the window maintaining customary actions like saving and loading models from file. Popup menus pops up when an object is right-clicked in the SCView. The actual action to execute when a specific item in the different menus are clicked on, is handled by the AppAction class that knows which action to call. Connectors and links also have separate dialog windows appearing when a specific element is selected in the SCView. The designer may set properties of the chosen element in these dialog boxes. A StringBuffer contains the text created during the code generation that subsequently is transferred into the Control Builder environment.

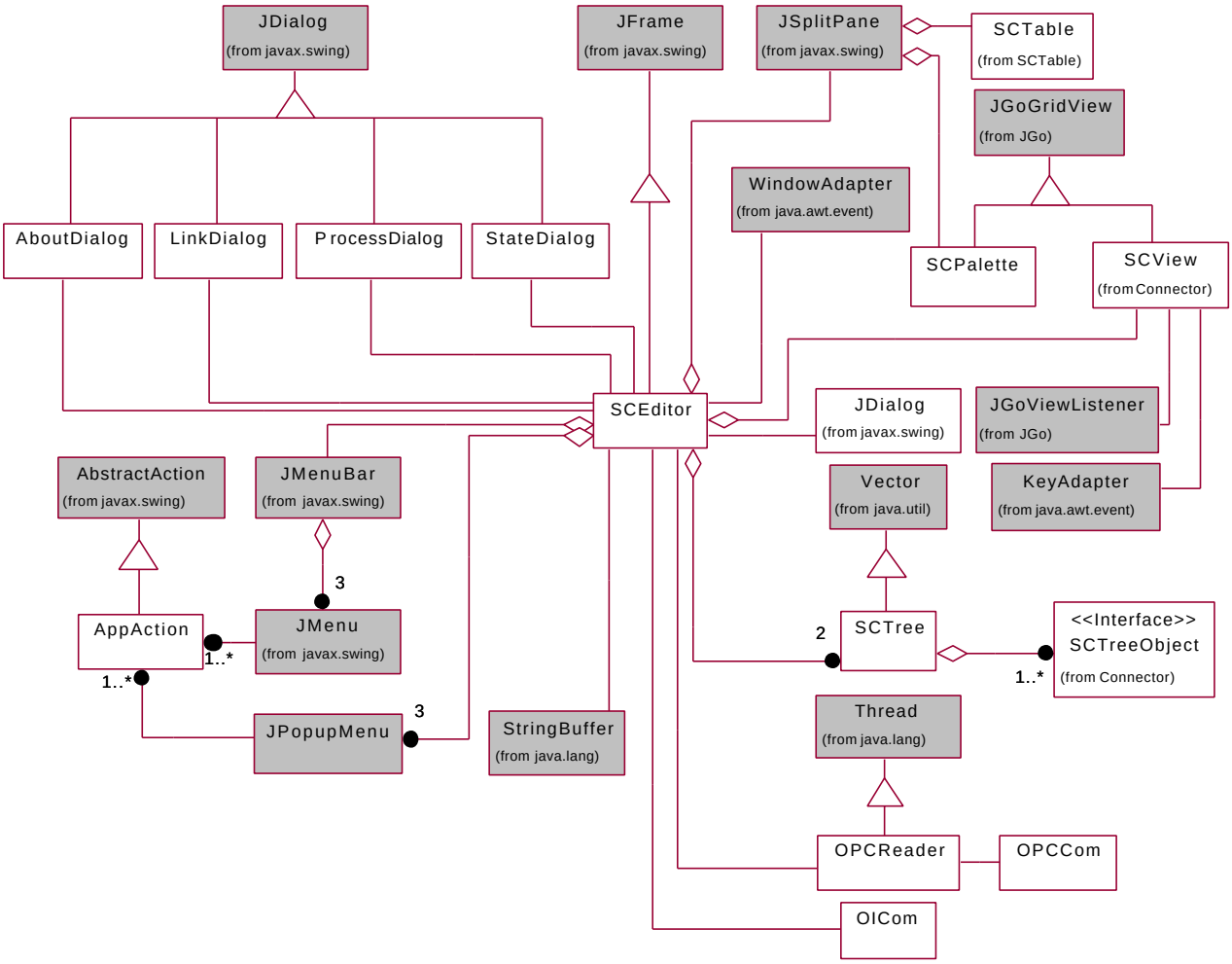


Figure 6.16 Class diagram showing the SCEditor block.

OICom is a class that transfers the generated code into the Control Builder by using so-called native method calls through the Open Interface. OPCCom is the corresponding class to OICom that get current values of parameters from the controller via the OPC interface during execution. OPCReader is actually a thread that executes on its own. The OPCReader is started when the execution starts and gets values at equidistant time intervals. The issue of communicating with the Control Builder is further discussed in chapter 8.

6.6.2.2 Connector

The classes in the Connector block are shown in figure 6.17. The abstract connector class is a JGoArea, which implements the concept of grouping objects together to form one single object. A Connector also implement some interfaces defined to obtain good structure among different classes during code generation. DefaultNode and StateNode are then specialized Connector classes. The StateNode is associated to an ActionBlockArea, which in turn consists of the three different code blocks editable by the user.

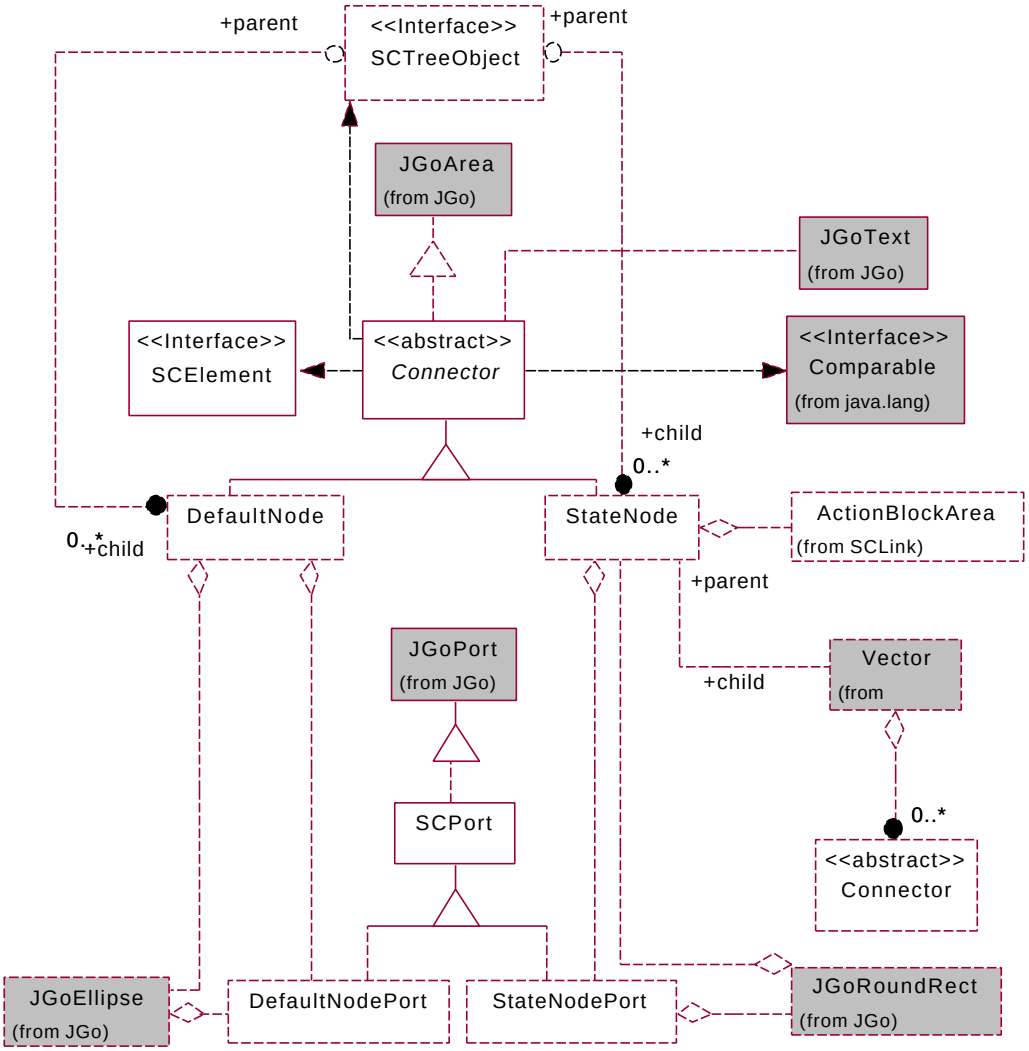


Figure 6.17 Class diagram showing the Connector block.

6.6.2.3 SCLink

Figure 6.18 shows the SCLink block. An SCLink is a graphical stroke connecting to ports. The link has an arrow at one end making the link act as a directed graph connecting two nodes that not necessarily must be different. Associated with the link is an Action Block consisting of two code blocks editable by the user. The first code block is the condition code block and the second code block is the action code block. Two JGoTexts are also associated with the link, representing the actual name and priority of the link respectively.

The CTransition class consists of a vector of one or more SCLinks. SCLinks that should be taken as a compound transition are grouped together in a CTransition. Four vectors are also associated together with the CTransition. The vectors consists of: the links included in the CTransition, states entered when the transition is taken, states leaved when the transition is taken and the children at lower hierarchy in the SCTree. Note that CTransition instances are not created until the designer generates code from a graphical model.

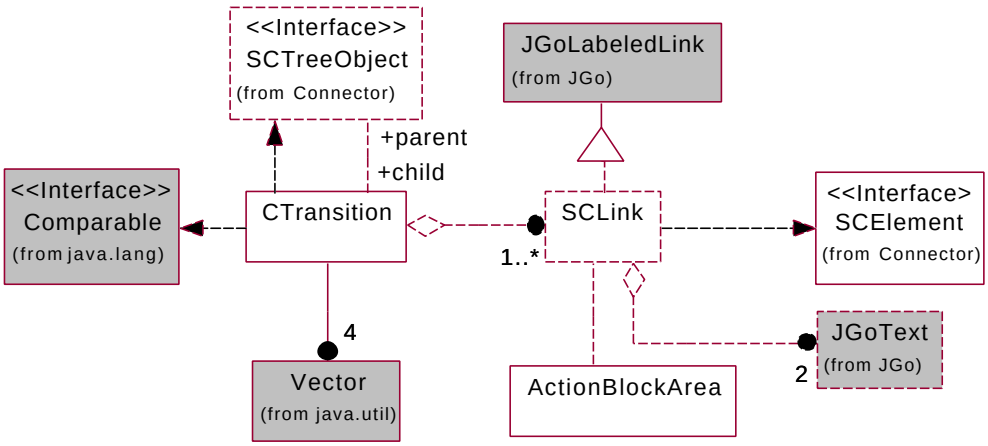


Figure 6.18 Class diagram showing the SCLink block.

6.6.2.4 SCDocument

An SCDocument represents a group of graphical elements that can be viewed inside a JGoView, see Figure 6.19. The SCDocument consists of an ordered list of elements that can be iterated in sequential order. To improve the structure among elements the document is divided into different layers. Elements on different hierarchical levels are located in different layers in the model. This gives freedom for the designer to choose the most suitable abstraction level to consider, by hiding layers out of interest for the moment. For convenience a special identifier class is associated with each layer. The identifier contains a unique number and a special colour for the graphical elements located in the specific layer. The DocumentBuilderFactory supports creation of the XML-document that corresponds to the graphical representation of the elements included in the SCDocument. To be able to also put declared parameters in the document, SCDocument asks SCTable to do so in a supplied XML-document.

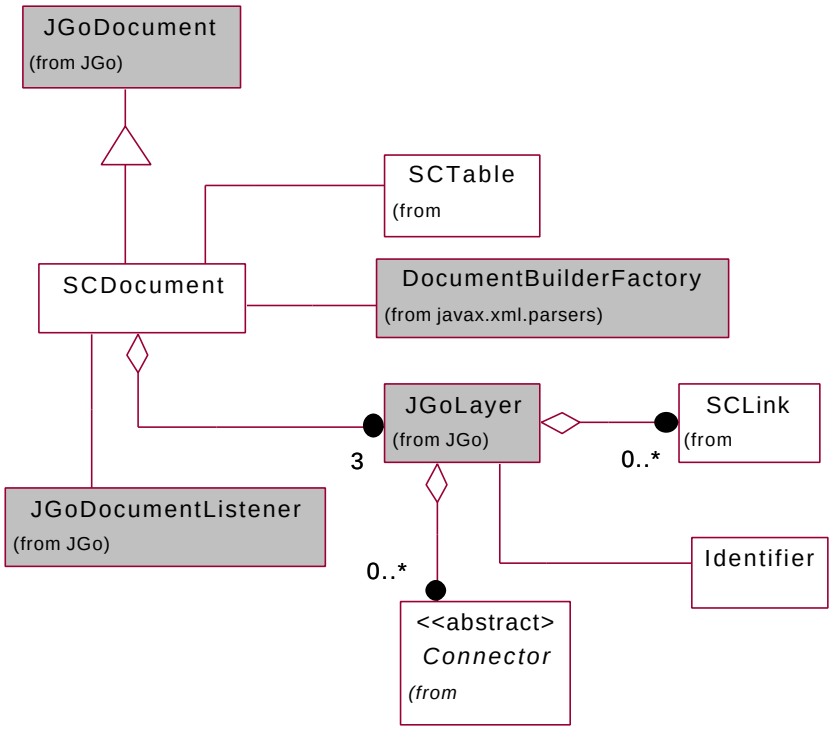


Figure 6.19 Class diagram showing the SCDocument block.

6.6.2.5 Overview

This code block is of less importance for the functionality of the prototype. However, the classes may be convenient to use when the diagram is large and hard to survey. The main class is the Overview that creates a new view showing the entire diagram in small scale, see Figure 6.20. Inside the view an OverviewRectangle is created showing the part of the chart that is shown in the ordinary view. This OverviewRectangle may be moved around using the mouse and the interior of the rectangle is accordingly shown in the ordinary window. The Overview is visualized using a JDialog frame provided in the standard Java package.

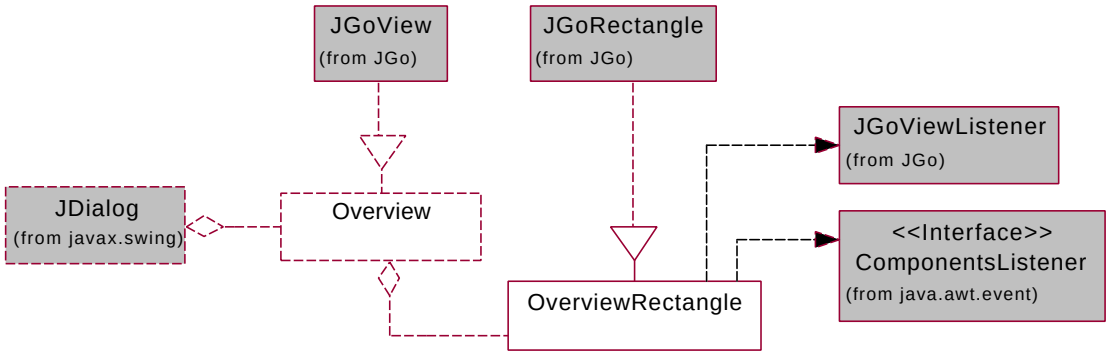


Figure 6.20 Class diagram showing the Overview code block.

6.6.2.6 SCTable

The SCTable is simply an editable table that permits the designer to declare arbitrarily many parameters used in the different code blocks, see Figure 6.21. These parameters are sent to the Control Builder together with the generated code from a designed model.

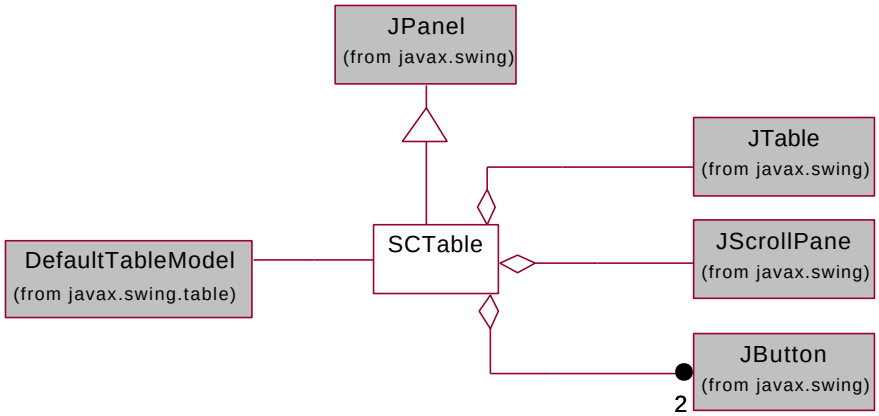


Figure 6.21 Class diagram showing the SCTable code block.

7. Code Generation

Overview: *This chapter deals with the question of how the Statechart diagrams should be executed. First some background information about the semantic concept of Statecharts will be explained. Then a brief description of the data structures needed to perform the actual code generation is given. The actual code generation, which consists of several consecutive steps, are presented next. Finally the theoretical description is applied to an example model to better explain the code generation procedure.*

7.1 Background

A Statechart model executes according to the corresponding semantical rules. The execution must be translated into some language included in the Control Builder. The reason to that is that the execution is performed in the Control ^{IT} and not directly in the prototype.

As there are no official semantics of Statecharts, it is free to propose any semantics that fit for the actual application. Yet, the semantics defined by David Harel, the founder of Statecharts, is regarded as almost a standard and is the semantics most often used. For that reason this definition of the semantics is used in this application. Some parts of the semantics are however ambiguous and may not give deterministic execution. In those ambiguous cases specific semantic rules have been developed.

As described in the chapter about Statecharts, the values of parameters and variables are evaluated at particular time instants, called steps, see Figure 7.1. Between these time instances the status is kept constant until the next step is evaluated. Parameters are represented by the parameters declared in the table in the prototype. Variables are used to represent the active property of each StateNode in the model. At each step, the values of these parameters and variables are evaluated, which in turn gives a new status. Each change of such a parameter or variable within a step is not sensed until the beginning of the next step. This property counteracts the risk for race conditions.

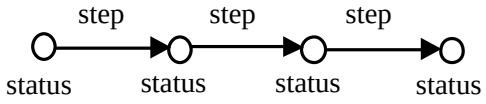


Figure 7.1 Execution of a Statechart model.

7.2 The Concept of Compound Transitions

When a transition fires it causes some states to be activated and others to be deactivated. If an activated state is a super state of XOR type, then the default transition residing inside this super state is also taken. The destination state of this default transition may in turn be a super state. Then the default transition inside this super state is taken as well. This reaction continues until the destination state does not contain any states. Suppose for example that the model shown in Figure 7.2 is executed. At the start the default transition T1 is taken. This causes transition T2 to be taken, since super state S1 is activated and its default transition is fired. This series of reactions occur each time the first transition in the series is fired. By collecting all transitions that fire in such a series of reactions into a Compound Transition (CT) instead, the whole CT can be fired as one sole entity. If the first transition in the CT

then fires, all other transitions included in the CT fire as well. All actions associated with each individual transition will be executed as if the transitions had been fired separately. During the actual code generation it is the Compound Transitions that are considered, not the individual transitions.

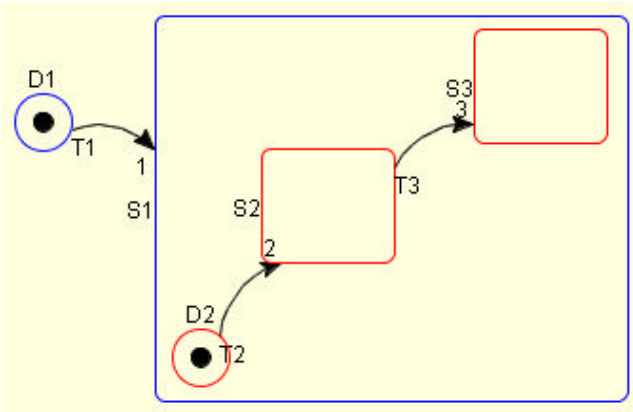


Figure 7.2 Model causing creation of a Compound Transition.

7.3 Data Structures

Before the code generation, some structure among compound transitions, StateNodes and DefaultNodes must be acquired. Two types of tree structures are created. A special class called SCTree is used to handle both these structures in the application. When the structures have been built up, the actual code generation is very simple to carry out. There is a special property about this SCTree class that needs some additional explanation. The SCTree class consists of a vector in which the included objects are inserted. Each object, called SCTreeObject, in the SCTree, has a parent attribute and a vector of children. These two attributes refer to the super state and the children of the actual SCTreeObject respectively. Thus, objects in the SCTree may either be stored directly in the vector or in a hierarchical tree structure. This provides a way to store more than one root in the SCTree.

7.3.1 Node Tree

The first tree structure is called NodeTree and contains StateNodes and DefaultNodes included in the model. An extremely simple NodeTree is shown in Figure 7.3, which is based on the model in Figure 7.2 above. The node at the highest level in the tree is called the root and represents the whole editing area. The root node acts as a super state to the nodes on the highest hierarchical level in the model. The root is created automatically by the prototype previous to the code generation. The NodeTree preserves the hierarchical relation between super states and sub-states according to the graphical model. The advantage of a hierarchical structure of the nodes, is that the descendants of a special node can easily be found, just by iterating down through the tree.

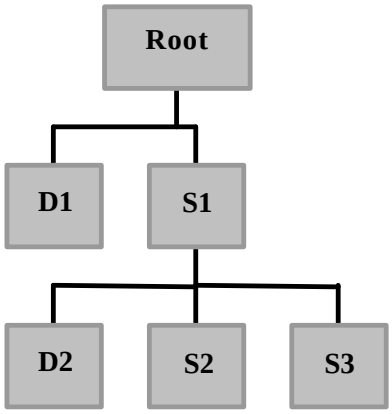


Figure 7.3 Node Tree based on figure 7.2.

7.3.2 CT Tree

The second structure is called CT Tree and contains the compound transitions. The CTs are also ordered in a hierarchical structure in the tree. The order of the CTs in the tree is based upon the priority of the first link included in the CT. Figure 7.4 shows the CT Tree based on the model in Figure 7.2. CT0 consist of link T1 and T2 while CT1 only consist of link T3. The grouping is based upon the rules discussed in the previous section about compound transitions. CT0 and CT1 reside in the same hierarchical level in the tree, at the root in this case, because there is no conflict between these two CTs. The priority is only considered if two CTs could give rise to a conflict. As mentioned before, two conflicting transitions have a common source state that is deactivated independently of which of the CTs that is taken. CT0 and CT1 could, according to the CT Tree, be taken in the same step if their corresponding source states are active. However, according to Figure 7.2 there is no possibility that D1, the source state of CT0, and S2, the source state of CT1 could trigger their connected links simultaneously. Therefore CT0 and CT1 are never taken at the same time even if the CT Tree does not prevent this. The tree of CTs is parsed from top to bottom during the actual code generation. The order among CTs in the CT Tree is hence maintained in the generated code. Due to the implicit priority among CTransitions in the code, the first fireable CTransition that is found should be taken.

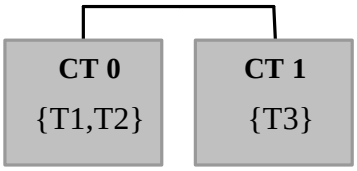


Figure 7.4 CT Tree based on figure 7.2.

Studying Figure 7.5 gives better insight about how the priority among compound transitions works. Suppose CT1 consists of transition T1, CT2 of T2 and CT3 of T3, respectively. Each pair of CTs in the model is in conflict. The reason is that each transition deactivates state S2 when fired. T3 has higher scope than both T1 and T2, thus CT3 is located at the top in the CT Tree. T1 and T2 have the same scope value and their explicit priorities must be considered. The explicit priority of T1 is 1 while T2 has priority 2. Hence, CT2 is located on top of CT1 in the CT Tree. The appearance of the CT Tree after this structuring is shown in Figure 7.6.

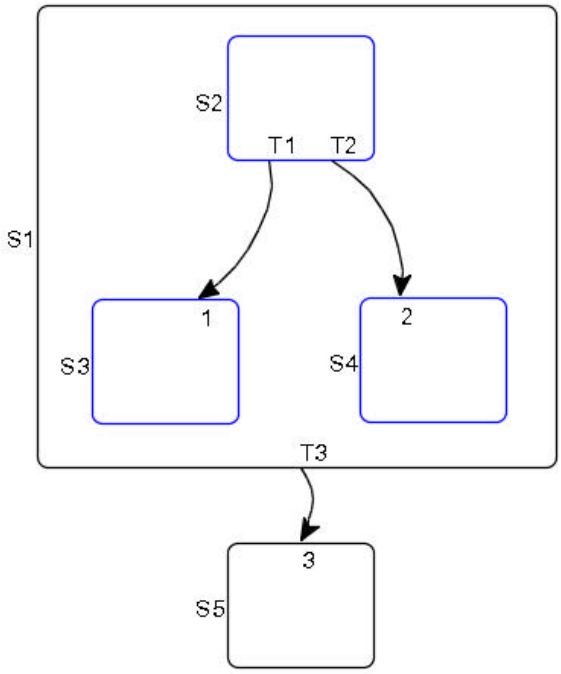


Figure 7.5 Example model showing priorities among compound transitions

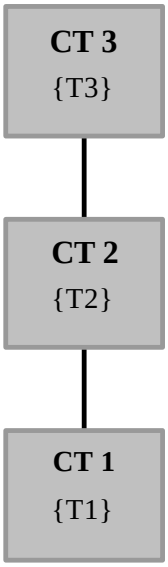


Figure 7.6 CT Tree for the model in Figure 7.5.

7.4 Code Generation

When the designer wants to create executable code from a graphical model, several steps have to be taken before the actual code could be generated. The procedure is shown in Figure 7.7.

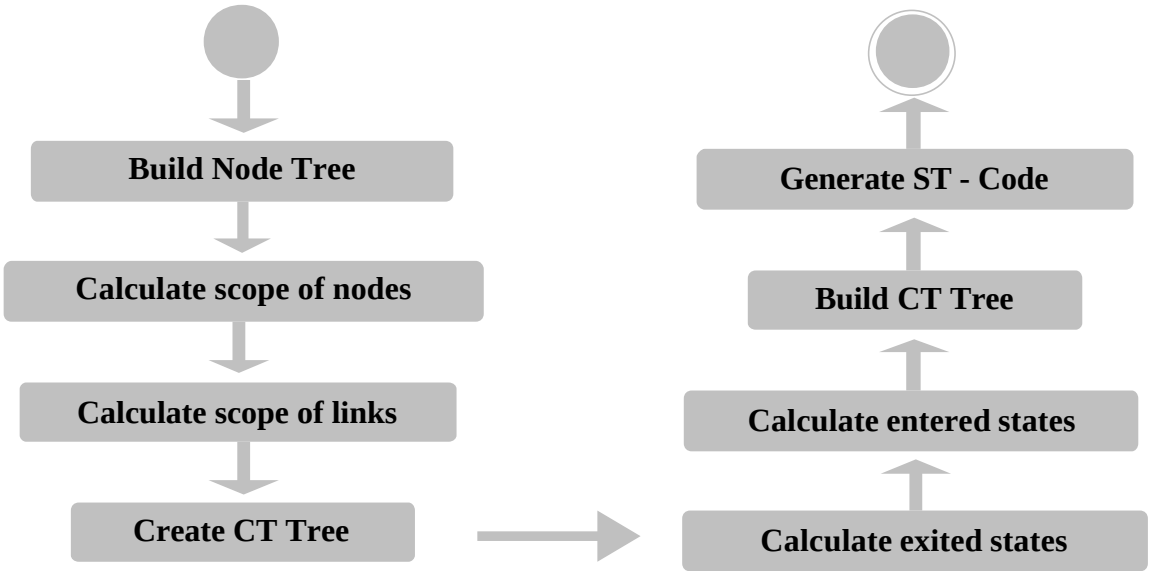


Figure 7.7 Steps taken to generate executable ST-Code.

7.4.1 Build Node Tree

The procedure begins with the creation of the NodeTree where all nodes in the actual model will be inserted. First of all, the root node is created, which is the super state of all elements included in the highest level in the editing area.

The elements are then inserted into the NodeTree by visiting the layers in the document in sequential order. When inserting the nodes residing in the first layer, they are explicitly set as children of the root. When nodes from the other layers are inserted into the tree, no parent-child relationship needs to be set. These relations have already been set when the nodes were inserted into the editing area.

7.4.2 Calculate Scope of nodes

The scope is defined by each individual node itself. In order to make it easier to compare the hierarchical level of different nodes in the implementation, an integer value represents the actual scope level. The scope of a node is simply calculated as the number of the layer in which the element resides. The identifier associated with the layer gives the number. Note that the root has the value zero of its scope. A node on a high hierarchical level in the model has consequently a small value of its scope. In Figure 7.2, S1 and D1 have the value one for their scope while D2, S2 and S3 have the value two.

7.4.3 Calculate Scope of Links

More effort is required to calculate the scope of links compared to the corresponding calculations for nodes. The scope of a link is defined as the scope of the common ancestor XOR-state of the source- and target node at the lowest hierarchical level. The scope of link T1 in Figure 7.8 is, according to this rule, state S1.

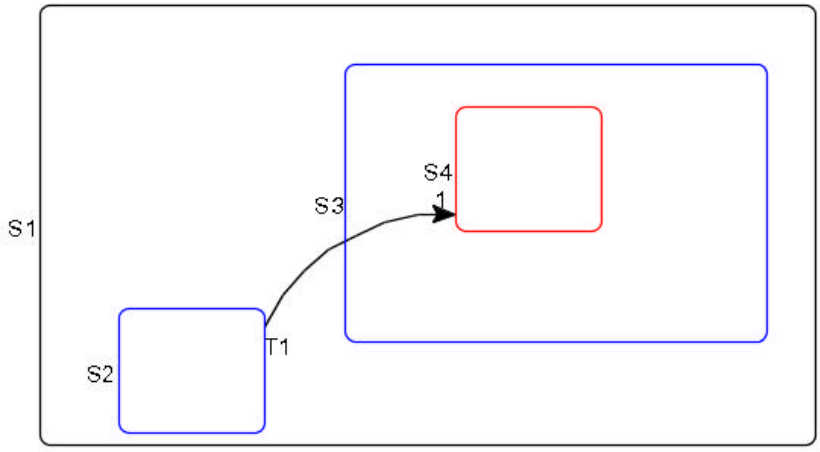


Figure 7.8 Example showing the calculations to get the scope of a link.

The scope of all links included in the model is calculated in this way. The source node of a link can either be a StateNode or a DefaultNode. The destination node is always a StateNode.

When calculating the scope of a link, the source node of the link is first investigated. In case the source is a DefaultNode it cannot possibly be the common ancestor. In fact a DefaultNode cannot contain any states at all. Thus the super state of the DefaultNode is fetched.

The super states of the source- and the destination node are successively iterated upwards in the hierarchy. This iteration is preceded until a common super state is found. The found ancestor super state may be of either AND- or XOR type, but according to the definition of the scope, the super state must be of XOR-type. Therefore the NodeTree is further iterated upwards in the hierarchy, until the first state of XOR-type is found. This node, which must be a StateNode, represents the scope of the actual link. The scope of the link is finally given the same value as the scope of the found ancestor state. The scope of a link is always found. This is because the root node is always of XOR type and is a common ancestor of all nodes in the model. In Figure 7.2, T1 has the scope value zero because the scope is the root StateNode. The scope of T2 and T3 is StateNode S1, which gives the scope value one.

The scope of link T1 in Figure 7.8 is calculated in the following way: The source of T1 is S2 and is first looked at. The parent of S2 is not fetched because S2 could potentially be the scope of T1, even if it is not in this case. The destination node of T1 is S4. Because source and destination node of T1 is different the scope is not found at this stage. The super state of S4 is fetched because S4 is located on lower hierarchical level than S2. S3 is super state to S4. S3 and S2 have the same scope. Therefore it does not matter which of their parents that are first fetched. Suppose S3s parent are first fetched. The parent of S3 is S1. Now S2 is located on lower hierarchical level than S1 and its super state is called for. S1 is super state to S2. A common super state is found. Since S1 is of XOR-type the scope of T1 is S1. The scope value of T1 is set to the scope value previously calculated for S1

7.4.4 Create the CT Tree

To simplify the decision of which links to fire, these are collected into compound transitions, as mentioned before. At this stage the CT Tree is created and the compound transitions are defined from the model. The CTs are then just inserted into the vector in the CT Tree. The tree structure is achieved at a later stage.

7.4.5 Calculate States Exited by Compound Transitions

Active source states of a CT are deactivated when the transition fires during execution. To minimize the calculations during execution, the states that potentially could be deactivated when a CT is taken are calculated in advance. Note that it is just the active states among the potential exited states that really are deactivated. This is important to know because it is only their associated actions that are executed.

When these deactivated states of the CTs are calculated it is sufficient to consider the first link in each CT. The reason for this is that all other links have DefaultNodes as sources, thus are never exited. This is not true for the DefaultNode located in the first layer that really is exited after the first step during execution, never to be activated again.

When calculating the deactivated states of a CT, the source node of the first included link is found and successive parents of this source connector are obtained. This iteration is preceded until the actual StateNode, which is the scope of the link is found. To be precise the procedure is stopped one step before this scope-node is reached, thus excluding the scope from being deactivated when the transition fires. The iteration from the source of the actual link upwards in the hierarchy is conveniently performed in the NodeTree.

When the scope-node of the CT is found, all descendants of the node next to the scope-node may potentially be exited when the transition fires. All these descendants are therefore added into a special vector in the CT. In Figure 7.2 the exited states of CT0 {T1,T2} is D1 because D1 is next to the scope of link T1. Exited states of CT1 {T3} are S2 because S2 is the state next to the scope (S1) of T3.

7.4.6 Calculate States Entered by Compound Transitions

Fired CTs also cause states to be activated during execution. These are also precalculated. Differently from the previous calculations of exited states, each link in a particular CT must be considered when the calculations are made.

For each link in a CT, its destination node, which is a StateNode, is investigated. The destination node is added to the vector of entered states of the CT. If this StateNode is of AND-type, all of its sub-states are also added to the vector of entered states. Remember that all nodes inside a super state of AND type are activated at the same time as their super state. The procedure of adding states to the vector of entered nodes propagates further downwards in the hierarchy until no more AND states are found.

If the scope of the link is on a higher hierarchical level than the parent of the originating destination node, this parent is also added to the entered vector and possibly its children in case this parent node is of AND-type. This procedure of getting successive parents are proceeded until the parent of the considered StateNode is on the same hierarchical level as the SCLink, i.e. it is the scope of the link.

The procedure is shown on a simple example: Suppose that the potential vector of entered states of a compound transition should be calculated. The CT consists of T6 and T3 in Figure 7.9. T6 is the first link in the CT and is considered first. The destination node of T6 is S3, which is immediately added to the vector representing entered nodes. Since S3 is of XOR type there are no more states to be added on lower hierarchical levels from link T6. But since T6 has scope zero and the parent of S3, which is S2, has scope one, S2 is also added to the vector of entered nodes. No more states are added to the vector when T6 is considered. The next link in the CT is T3. S4 is the destination node of T3 and is added to the vector. The scope of T3 is 2 and the parent of S4 has also scope 2. Thus no more nodes are added to the vector. As no more links are included in the CT, all states that are entered when this CT fires are S2, S3 and S4.

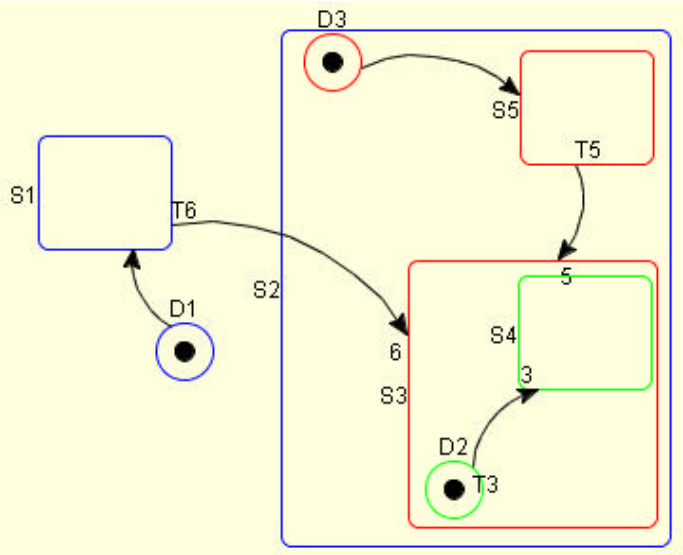


Figure 7.9 Model demonstrating which state to enter when the CT consisting of T6 and T3 fires.

7.4.7 Build the CT Tree

Compound transitions have presently been included in a vector in the CT Tree but have not been ordered in any specific way. CTransitions are now ordered in a hierarchical structure. The obtained structure forms the order that elements are visited during the subsequent code generation. The following ordering is based upon two priorities, called prio1 and prio2. Both priorities are derived from the starting link of the CTransition. Prio1 is the scope of this first link, where the link on highest hierarchical level has the highest priority. The designer may assign prio2 during modelling. By default the most recently inserted link has greatest priority with respect to prio2.

First the vector of CTs in the CT Tree is sorted with respect to prio1 and prio2. Prio2 is only considered in case of a tie of prio1. Each CTransition in the tree is iterated through, starting with the one with the highest priority, called CT1 in the sequel. CT1 is compared to each subsequent CT in the vector regarding possible conflict. CT1 is first compared to the nearest CT in the vector i.e. the one with second highest priority e.g. CT2.

Two CTs are in conflict if they have any common node in the vector of possible exited states. If CT1 and CT2 are not in conflict, CT1 is compared to the next CT in the vector. If CT1 and CT2 on the other hand are in conflict, CT2 is compared to the possible children of CT1. If CT1 has no children, then CT2 is added as child to CT1 and removed from the vector. If CT1 has children, CT2 is successively compared to them and checked for conflict. This procedure of successively searching for children continues until no more children exist or until no conflict is found on the actual level. CT2 is then inserted as child of that child of CT1 that CT2 was latest in conflict with. CT2 is then removed from the vector in the CT Tree. CT2 is from now on only reachable by iterating downwards from CT1.

When CT1 has been compared to all CTs, the next CT in the vector is compared to all subsequent CTs in the vector in the same way. When all CTs in the vector have been checked the CT Tree is structured. It may happen that the CT Tree consists of more than one root, if the CTs on the highest level are not in conflict. From this follows that more than one CT may potentially fire in the same step.

The procedure is now exemplified starting from the model shown in Figure 7.5. First each CT in the CT Tree is ordered according to prio1 and prio2. Note that a lower value of prio1 gives higher priority

while a lower value of prio2 gives lower priority. The appearance of the CT Tree after sorting is shown in Figure 7.10.

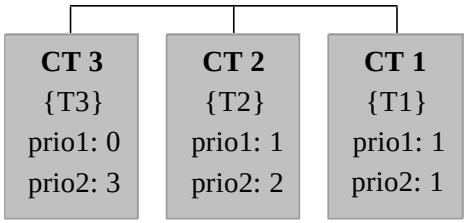


Figure 7.10 CT Tree based on the model shown in Figure 7.5 after sorting.

The first pair that is investigated is CT3 and CT2. CT3 and CT2 are in conflict, thus CT2 will be a descendant of CT3. CT3 has presently no descendants that CT2 in addition should be compared with. CT2 is consequently added directly below CT3 in the CT Tree and removed from the vector. The appearance of the CT Tree now looks like Figure 7.11.

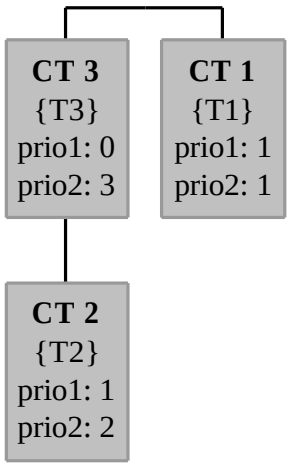


Figure 7.11 CT Tree when CT 3 and CT 2 have been compared.

CT1 is now compared with CT3 and they are found to be in conflict. Because CT3 has descendants CT1 is in turn compared with CT2. CT1 and CT2 are in conflict and CT1 is inserted below CT2 in the CT Tree. The final appearance of the CT Tree is shown in Figure 7.12.

7.4.8 Generate ST Code

When the CT Tree is structured the actual code generation is straightforward. It is done by simply iterating the CT Tree from the root(s) downwards in the hierarchy. Each CT on the way down is checked whether it is fireable or not.

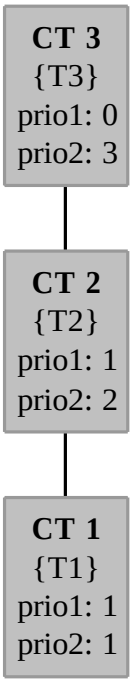


Figure 7.12 Appearance of the CT Tree when structured.

7.5 Example

A simple demonstration example is now presented to better show how the actual code generation procedure works. The model that the example is based upon is shown in Figure 7.13.

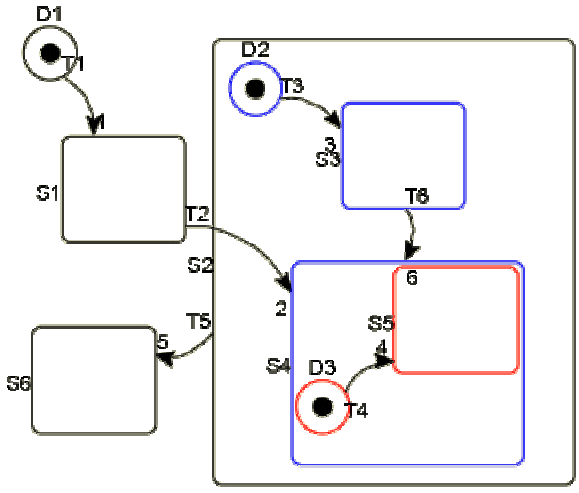


Figure 7.13 Statechart model considered in the demonstration example.

First the NodeTree is created. The root node is placed in the top of the tree and the rest of the hierarchy follows directly from the implicit hierarchy in the Statechart model, see Figure 7.14.

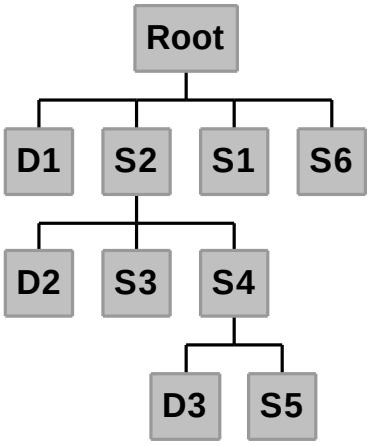


Figure 7.14 Node Tree created from the model in figure 7.13.

Then the scope of the nodes in the Node Tree is calculated, starting with the root-node that is assigned to the value zero of its scope. Nodes at lower hierarchical levels get lower scope values. The scope values of each node is presented in Table 7.1.

Node	Scope
Root	0
D1	1
S1	1
S6	1
S2	1
D2	2
S3	2
S4	2
D3	3
S5	3

Table 7.1 Scope of nodes of the model in Figure 7.13.

Then the scope of each transition is calculated according to the previously mentioned procedure. The scope of T1 is for example calculated by looking at its source and destination node, which are D1 and S1. The super state of those nodes is the root node, which of course is a common parent to both. From that follows that T1 is assigned to the same scope value as the root i.e. the value 0. The scope of all transitions in the model are shown in Table 7.2.

Link	Scope
T1	0
T2	0
T3	1
T4	2
T5	0
T6	1

Table 7.2 Scope of links in figure 7.13.

Now are the compound transitions defined and inserted in a new CT Tree. The CTs are inserted in the vector in the CT Tree in arbitrary order, see Figure 7.15.

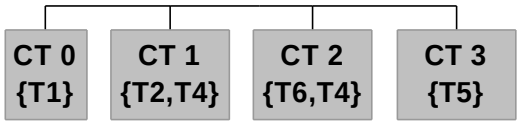


Figure 7.15 Compound transitions defined from the model shown in Figure 7.13.

The next step in the code generation procedure involves the calculation of potentially exited states of the compound transitions. Only the first link in each CT needs to be considered during this calculation. The potential exited states of the CTs presented in the model in Figure 7.13 are shown in Table 7.3.

CT	Exited States
CT 0 {T1}	D1
CT 1 {T2,T4}	S1
CT 2 {T6,T4}	S3
CT 3 {T5}	S2,S3,S4,S5

Table 7.3 Potentially exited states of the CTs.

Then entered states of the CTs are calculated. In this calculation each link in the CT must be looked at. The found states are not only potentially states to be entered but are certainly entered when the associated CT fires. The states entered when each CT fires are presented in Table 7.4. When CT1 fires S4, S2 and S5 are all activated. State S4 and S2 are due to link T2 while S5 is a consequence of link T4.

CT	Entered States
CT 0 {T1}	S1
CT 1 {T2,T4}	S4,S2,S5
CT 2 {T6,T4}	S4,S5
CT 3 {T5}	S6

Table 7.4 Shows which states to activate when the associated CT fires.

Now it is time to structure the CTs previously inserted into the CT Tree. First the CTs are sorted in the vector according to prio1 and then according to prio2. Remember that prio1 was the implicit scope value of the first link in the CT. A lower value of this scope gives higher priority. The second priority, prio2, is the explicit priority, possible to set for each individual link during the modelling. Higher value of prio2 gives higher priority. The respective priorities of the CTs are shown in Table 7.5.

CT	Priority
CT 0 {T1}	0, 1
CT 1 {T2,T4}	0, 2
CT 2 {T6,T4}	1, 6
CT 3 {T5}	0, 5

Table 7.5 Priorities of each CT of the model in figure 7.7.

When the CTs are ordered according to the priorities given in Table 7.5 the vector of CTs in the CT Tree looks like Figure 7.16.

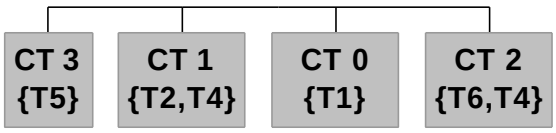


Figure 7.16 Vector in CT Tree after reordering of the CTs according to their respective priorities.

Now each pair of CTs in the vector are compared regarding possible conflict, starting from the left with CT3 and CT1. The only pair of CTs in the vector found in conflict is CT2 and CT3. This depends on that both CTs deactivate state S3 when they are fired, see Table 7.3. Because CT3 has higher priority than CT2, CT2 is placed as child of CT3 in the CT Tree. The final structure among CTs in the CT Tree is shown in Figure 7.17.

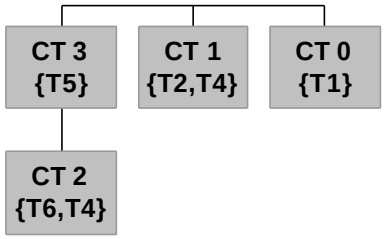


Figure 7.17 Appearance of the CT Tree when structured.

Now most of the work of the code generation is already done. The remaining part is the actual code generation. This will be presented subsequently. All the presented code will be parsed in each step. First some initialisation is performed. To carry out this initialisation just one single time, a special Boolean flag is used. In the actual example the initialisation code looks like this:

```
(* Initialisation, executed only once *)

IF firstLoop THEN
     firstLoop:=false;
     D1.active:=true;
     D2.active:=true;
     D3.active:=true;
END_IF;
```

Then all transitions, which are represented as Boolean variables, are reset:

```
(* Reset all Transitions *)

T1:=false;
T2:=false;
T5:=false;
T6:=false;
T3:=false;
T4:=false;
```

All states, which are represented as special data structures are reset:

```
(* Reset all States *)

S1.isActivated:=false;
S1.isDeactivated:=false;
     ...
```

Then the CTs are checked to see if they should be fired or not. This is performed by iterating through the CT Tree from top to bottom:

```
(* Check if CT3 is fireable *)
IF S2.active & true THEN
     T5:=true;
ELSE

(* Check if CT2 is fireable *)
     IF S3.active & true THEN
          T6:=true;
     END_IF;
END_IF;

(* Check if CT1 is fireable *)
(* Check if CT0 is fireable *)
```

In each IF statement both the source state and the triggering condition must be evaluated to true, to make the link fireable.

The transitions to fire have been decided and each state in the exit-vector of fireable transitions are deactivated:

```
(* Deactivate states *)

IF T5 THEN
     IF S2.active THEN
          S2.active:=false;
          S2.isDeactivated:=true;
     END_IF;
     ...
END_IF;

IF T6 THEN ... END_IF;

IF T2 THEN ... END_IF;

IF T1 THEN ... END_IF;
```

All states that at the previous stage have been marked as deactivated execute their exit actions:

```
(* Execute exitActions *)

IF S1.isDeactivated THEN
     (* exit *)
END_IF;
...
```

All transition actions of fireable transitions are executed:

```
(* Execute transitionActions *)

IF T1 THEN
    (* action *);
END_IF;
...
```

All states in the entered vector of all fireable CTs are marked as activated:

```
(* Activate states *)

IF T5 THEN
    IF NOT S6.active THEN
        S6.active:=true;
        S6.isActivated:=true;
    END_IF;
END_IF;
...
```

Enter actions of these marked states are executed:

```
(* Execute enterActions *)

IF S1.isActivated THEN
    (* enter *)
END_IF;
...
```

Finally the during actions of all still active states are executed:

```
(* Execute duringActions *)

IF S1.active THEN
    (* during *)
END_IF;
...
```

All code that is generated from the model in Figure 7.13 has now been presented. The code is inserted in a special XML document to conform to the Control Builder. The actual code is inserted as a ST code block defining a new ControlModuleType in the Control Builder. The XML document looks like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ControlModuleType Name="SCCMType" Protected="0" Hidden="0”
     Scope="public” InteractionWindow="" AspectObject="1”
     xmlns="urn:CBOpenIFSchema2_0">
     <CMParameters> </CMParameters>
     <CMVariables>
          <CMVariable Name="S1" Type="StateNode”
               Attribute="retain" InitialValue= "" />
     </CMVariables>
     <CodeBlocks>
          <STCodeBlock Name="Code">
               <ST_Code><![CDATA[ Generated Code]]>
               </ST_Code>
          </STCodeBlock>
     </CodeBlocks>
</ControlModuleType>
```

The document is structured according to the rules governed by the Open Interface specification. The XML document represents a ControlModuleType, which will be instantiated later in the Control Builder. This document also declares the parameters specified in the table in the prototype. Each parameter is declared inside a CMParameter tag. Then all nodes and transitions are declared as variables. These are declared inside CMVariable tags. The previously generated code is inserted as a large code block indicated by the *Generated Code* expression in a special code block.

8. Communication with the Control Builder

Overview: *In this section the communication with the Control Builder is described. First the problem of transferring generated code into an application in the Control Builder is discussed. This has been done in two different ways. The first approach transfers the code in an indirect way by first storing the code in a file. The second approach transfers the code directly into the Control Builder. The next problem is to read the values of parameters and variables during execution. Communication through an OPC Server performs this. Finally some experiences gained during execution are presented.*

8.1 Introduction

The aim is to execute the graphical representation in the Control ^{IT} environment. The generated code must therefore be transferred into the Control Builder. Most of the functionality in the Control Builder can be manipulated through the Control Builder Open Interface. For that reason it seemed as if using this should be the easiest if not the only way, to perform the code transfer.

The Open Interface can be used to create projects in the Control Builder in offline mode. When the actual execution has been started, Open Interface provides limited functionality. It is not possible to get current values of variables through Open Interface in online mode. To get run-time values from the Control Builder, another approach must be used. If the project in the Control Builder is downloaded into a Soft Controller, online values can be fetched. This is done by connecting an OPC Server to the Soft Controller and reading the values from the OPC Server during execution.
These two subjects will be discussed in the following text.

8.2 Code Transfer To Control Builder

At this stage the designer has created a graphical Statechart model in the prototype. It is now desired to execute the model in the Control ^{IT} environment. For that purpose, the execution of the graphical model is first translated into Structured Text code. This code is then transferred into to the Control Builder to be executed as an ordinary application. In this section two approaches to transfer the generated code into the Control Builder are discussed.

8.2.1 First Approach – Code Transfer through Test Client

To interface to the Control Builder in a easy way, the Open Interface provides a special test application, called Test Client. The Test Client takes an XML-document as argument, which it parses. This XML document consists of directions or “test vectors” i.e. method calls to be forwarded through the Open Interface. The method calls are applied to the Control Builder and can manipulate most of its functionality, e.g. create new projects.

To transfer the generated code into the Control Builder the code must first be stored in a file, see Figure 8.1. The code can be saved directly from the prototype. Then the Test Client is called with the XML-document of method calls given as argument. Before executing the Test Client, the Control Builder must first be started in offline mode i.e. not in simulate mode.

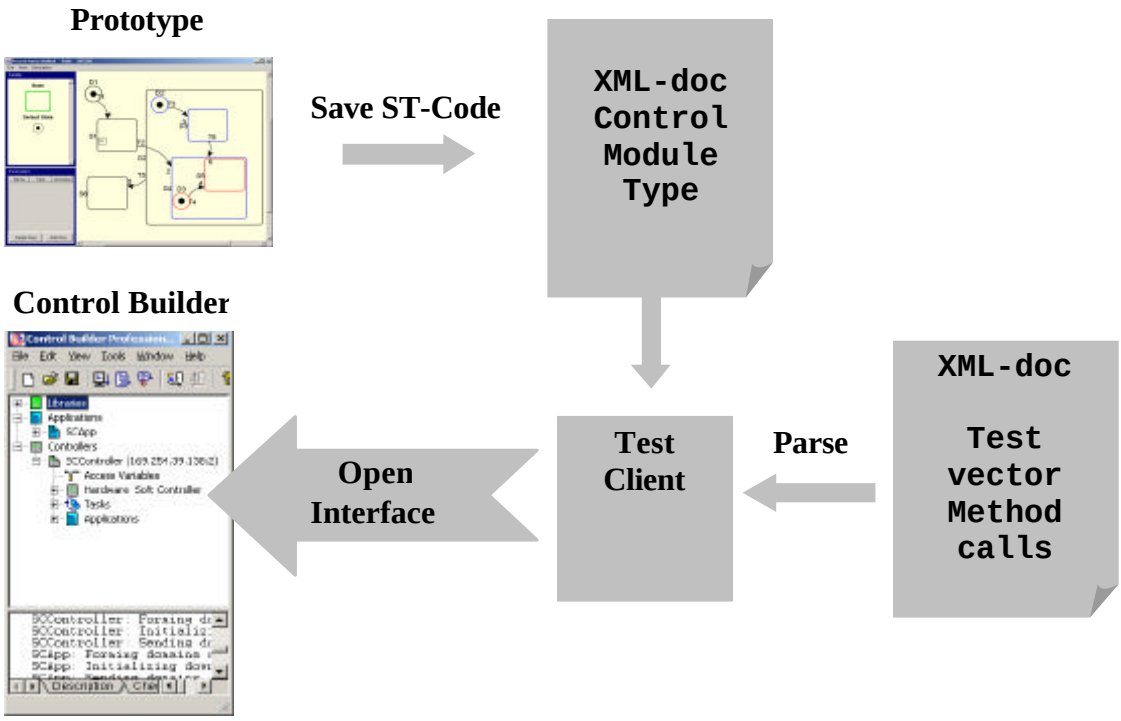


Figure 8.1 Transfer of XML-document between prototype and Control Builder using Test Client.

The method calls that the Test Client parses through in the XML-document will now be discussed.

The first method call in the XML-document is:

```
<MethodCall Name="CloseProject" ExpectedResult="Ok">
</MethodCall>
```

This method closes the existing projects that potentially exist in the Control Builder.

The second method call creates a new project in the Control Builder:

```
<!--Create project ...-->
<MethodCall Name="NewProject" ExpectedResult="Ok" >
<Parameter Value="SCProject"/>
</MethodCall>
```

The single argument constitutes the name of the project.

Calling the following method creates a new application:

```
<MethodCall Name="NewApplication">
<Parameter Value="SCApp"/>
</MethodCall>
```

The parameter gives the name of the application in the Control Builder.
The order in which the methods are executed could not be altered. A project must really exist in the Control Builder before applications could be added to it.

In the next method call, the document that contains the generated code is transferred into the Control Builder.

```
<!—-Add generated code -->
<MethodCall Name="NewControlModuleType"
ExpectedResult="Ok">
<Parameter Value="SCControlModuleType"/>
<Parameter Value="SCApp"/>
<Parameter Value="SCTest.xml"/>
</MethodCall>
```

The most important argument that this method need is the file containing the definition of the actual ControlModuleType. This file is named *SCTest.xml*.

Earlier it was mentioned that the nodes in the graphical model were represented as a new data type in the Control Builder. This is now taken care of by first creating a library to contain the data type in:

```
<MethodCall Name="NewLibrary" ExpectedResult="Ok">
<Parameter Value="MyNewLibrary"/>
<Parameter Value="SCProject"/>
```

After that the actual data type is created and inserted into the newly created library:

```
<MethodCall Name="NewDataType" ExpectedResult="Ok">
<Parameter Value="StateNode"/>
<Parameter Value="MyNewLibrary"/>
<Parameter Value="AutoSCTestStateNode.xml"/>
</MethodCall>
```

The definition of the StateNode data type is contained in the *AutoSCTestStateNode.xml* file. This file declares the StateNode data type holding the three Boolean attributes *isActivated*, *isDeactivated* and *active*. These attributes are used during execution to indicate the actual state of the StateNode.

At this stage, no instance of the new ControlModuleType has yet been created. This is performed by the following method call:

```
<MethodCall Name="NewControlModule" ExpectedResult="Ok">
<Parameter Value="SCControlModule"/>
<Parameter Value="SCControlModuleType"/>
<Parameter Value="SCApp"/>
</MethodCall>
```

As mentioned before, an application cannot execute on its own. To be able to execute the application, a controller must be created and added to the project. The controller is created by:

```
<MethodCall Name="NewController" ExpectedResult="Ok">
<Parameter Value="SCController"/>
<Parameter Value="Soft Controller"/>
<Parameter Value="SCProject"/>
</MethodCall>
```

The created controller should conform to the actual controller into which the application should be downloaded. In this project the application will be downloaded to a Soft Controller for execution.

To actually connect this controller to the real controller, the system identity must be set. This is done by the following call:

```
<MethodCall Name="SetSystemIdentity" ExpectedResult="Ok">
<Parameter Value="SCController"/>
<Parameter Value="169.254.39.138:2"/>
</MethodCall>
```

The second parameter is the id number of the local computer in which the actual Soft Controller resides.

Calling the following method then specifies the hardware unit inside the controller:

```
<MethodCall Name="NewHardwareUnit" ExpectedResult="Ok">
<Parameter Value="SCController.0"/>
<Parameter Value="CPU"/>
</MethodCall>
```

Then the application and the created controller must be connected, which is done by this method call:

```
<MethodCall Name="SetConnectedApplications"
ExpectedResult="Ok">
<Parameter Value="SCController"/>
<Parameter Value="AutoSCtest_1.xml"/>
</MethodCall>
```

The second parameter is an XML file that contains the name of the application that should be connected.

The final thing to do before the simulation could be started is to connect the control module to a task in the controller. This connection is performed by the call:

```
<MethodCall Name="SetTaskConnection"
ExpectedResult="Ok">
<Parameter Value="SCApp.SCControlModule"/>
<Parameter Value="SCController.Normal"/>
</MethodCall>
```

The appearance of the explorer window in the Control Builder when the Test Client has been executed is shown in Figure 8.2.

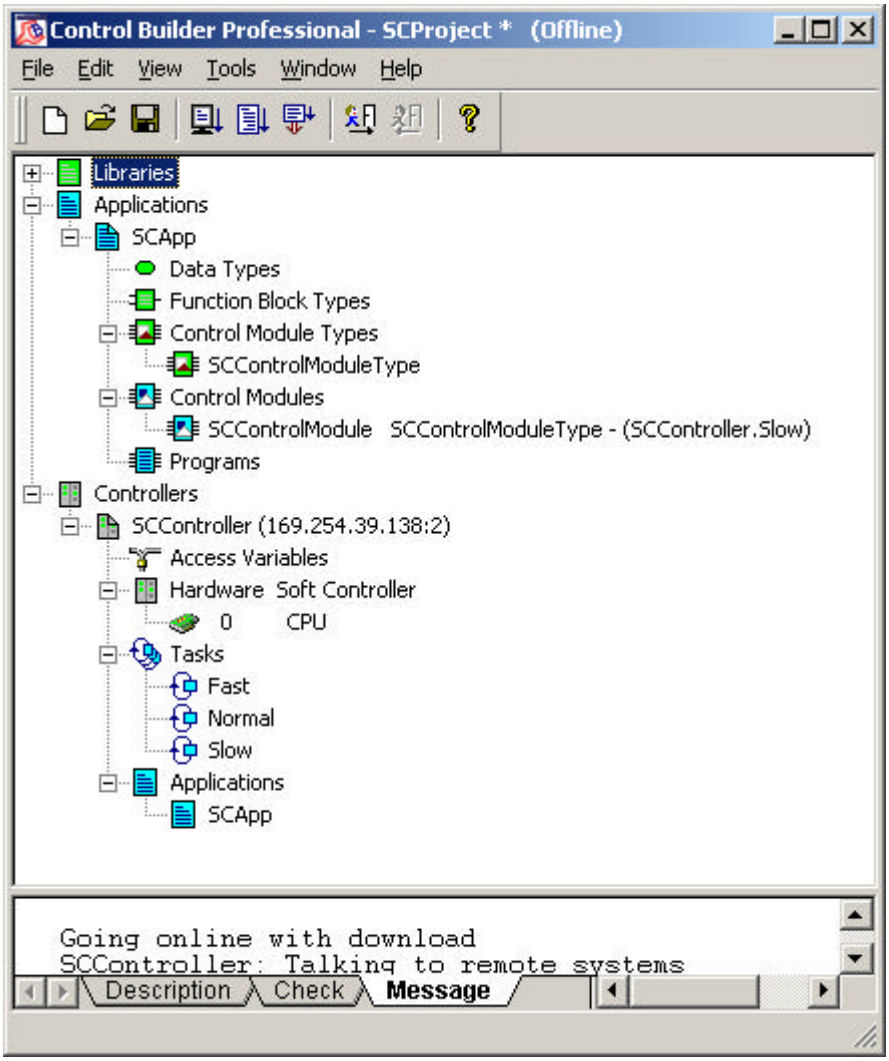


Figure 8.2 Project explorer in the Control Builder.

The simulation of the application in the Control Builder is started by a *Simulate* method call in the XML document. It is possible to follow the simulation in the Program Editor, where actual values of variables and parameters are shown during execution, see Figure 8.3.

By using the Test Client it is possible to transfer the generated code into the prototype and simulate it in the CB. The only source of irritation is that it not possible to transfer the generated code directly from the prototype into the Control Builder. The prototype must first store the generated code in a file. It is then necessary to explicitly call the Test Client to get the file into the Control Builder for execution. This inconvenience led to a desire to directly transfer the code from the prototype into the Control Builder and start the simulation. This is solved by the second approach.

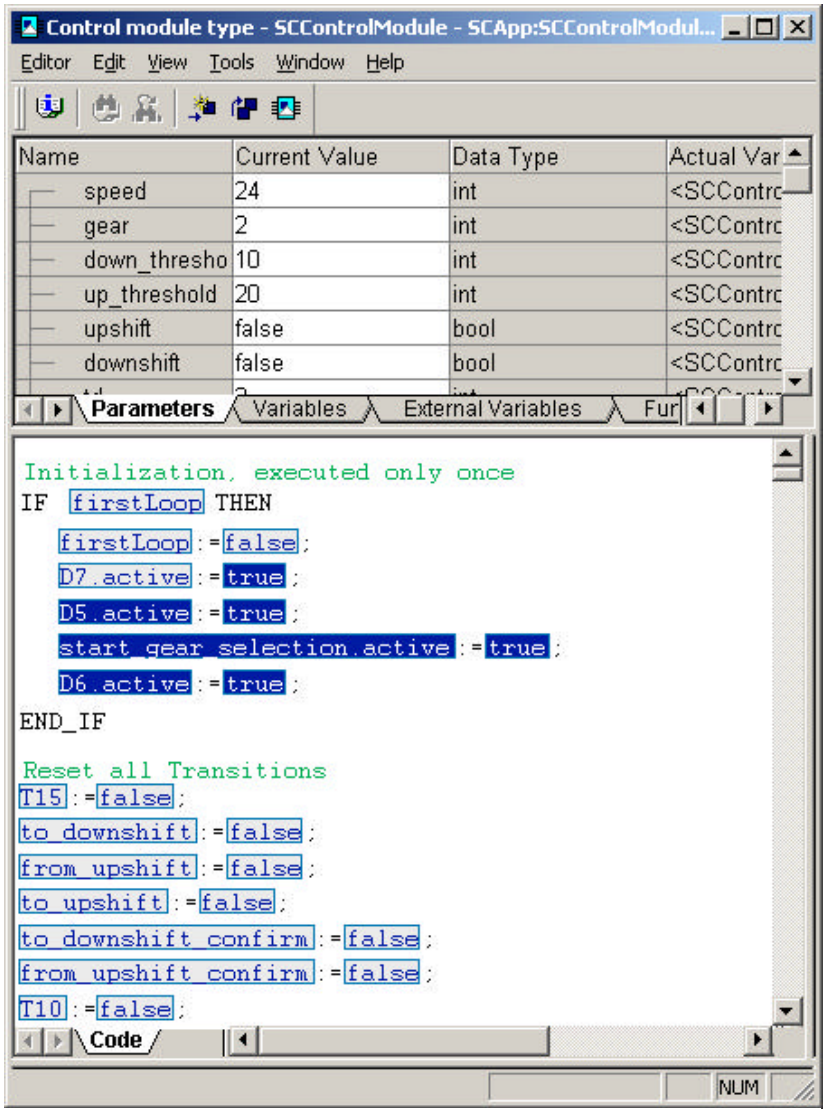


Figure 8.3 Program editor during simulation.

8.2.2 Second Approach – Code Transfer through Java Native Interface (JNI)

It is not possible to get in contact with the Control Builder directly from the prototype in a simple way. The reason is that the Open Interface exposed from the Control Builder really is a COM interface. COM (Component Object Model) was defined by the Microsoft Corporation to integrate products in order to get language independence. It was investigated how to communicate with COM objects directly from Java. However, this resulted in complicated or non-standard solutions. One of the first attempts was to use Microsoft Java, which supports integration of Java and COM. That approach ended in problems and was rejected. Another possibility that arose during the investigations was to use the Java Native Interface (JNI) API in Java. This API is included in the standard Java package and facilitates integrating Java programs with programs written in other languages, see Figure 8.4. The procedure of creating this native interface starts on the Java side. The methods that should be applied to the Control Builder are not implemented but just declared in the Java-classes. These native methods are in addition marked with the *native* keyword.

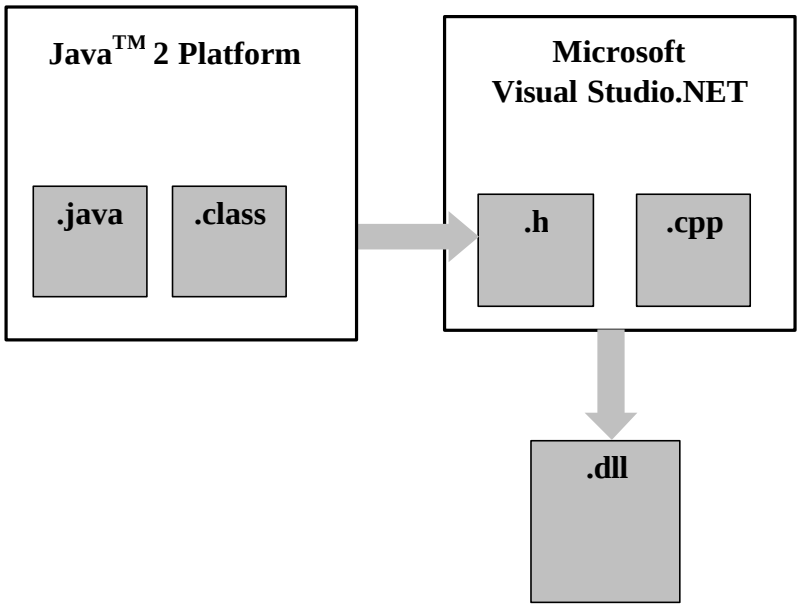


Figure 8.4 Block diagram showing how native methods could be implemented using the Java Native Interface. The *.java*-file contains the native method declarations. The *.h*-file is a header file generated from the Java-side conforming to the C++ envionment. The native methods are implemented in a C++ application in a *.cpp*-file. The *.dll*-file is a Dynamic Loadable Library generated from the native class to be imported into the Java Virtual Machine during execution.

The method for creating a project in the Control Builder from the Java side looks like this:

```
//create a new project in CB
public native void newProject(String projectName);
```

Notice the *native* keyword indicating that the real implementation of the method is provided from a native class outside the JVM (Java Virtual Machine). When all native methods have been declared the Java-file is compiled. By using a special Java tool it is possible to create a special header file from the actual Java class. This header file declares how the method declaration and arguments would look like in a C++ environment. This header file could then be implemented in a C++ application. The implemented method look like this on the native side:

```
JNIEXPORT void JNICALL Java_OICom_newProject
(JNIEnv *env, jobject, jstring projName) {

          //implementation
}
```

If a dll-file (dynamic loadable library) is created from this implementation, it could be loaded into the JVM (Java Virtual Machine) during run-time. The library could be loaded into the JVM in the following way:

```
static {
            System.loadLibrary("OICommunicator");
    }
```

OICommunicator is the dll-file created from the native side.

Using this JNI interface, the generated code could be transferred directly into the Control Builder through Open Interface without the intermediate storage in a file, see Figure 8.5. The methods provided by the Open Interface are easily imported into the C++ application and may consequently perform direct methods calls to the Control Builder. The *Main.exe* file of the Control Builder is simply included in an import statement in the native class. The sequence of method calls to the Control Builder is exactly the same as was presented in the previous section when the Test Client was used.

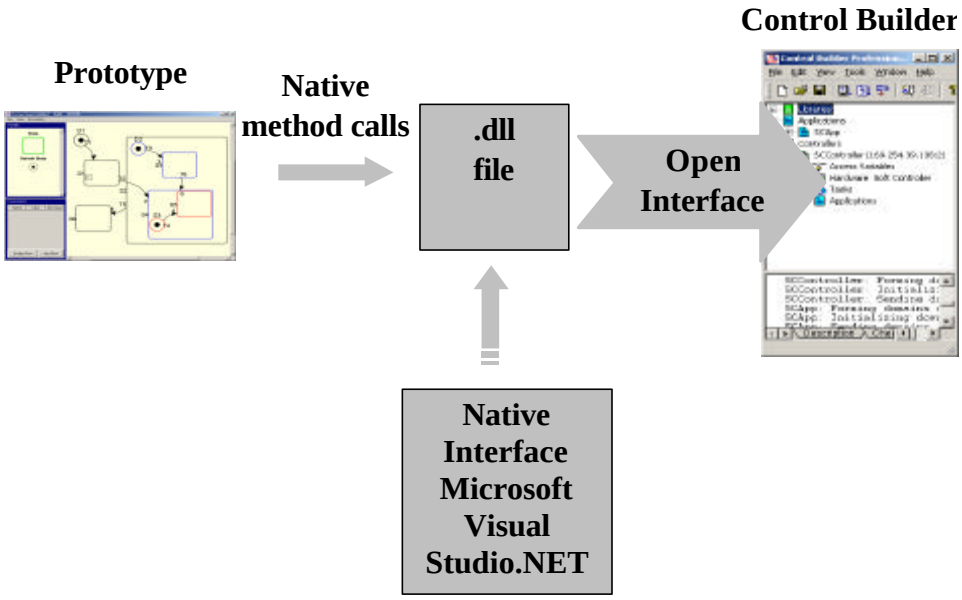


Figure 8.5 Transfer of code using Java Native Interface.

8.3 Update variables through the OPC Server

Executing a graphical Statechart model in the Control Builder was the first stage in the process of integrating the prototype into the Control ^{IT}. The direction in which data can flow is mainly a one-way street. This is due to the properties of the Open Interface, providing rather limited support during the actual simulation of an application. To open up the Control Builder during simulation, another communication approach must be considered. A natural way to do this is by using the OPC Interface. The Control ^{IT} provides an OPC Server that implements the OPC interface. The OPC Server can in turn be connected to several controllers, via the MMS Server, see Figure 8.6. A MMS Server coordinates several Control ^{IT} applications running simultaneously, see section 2.5.

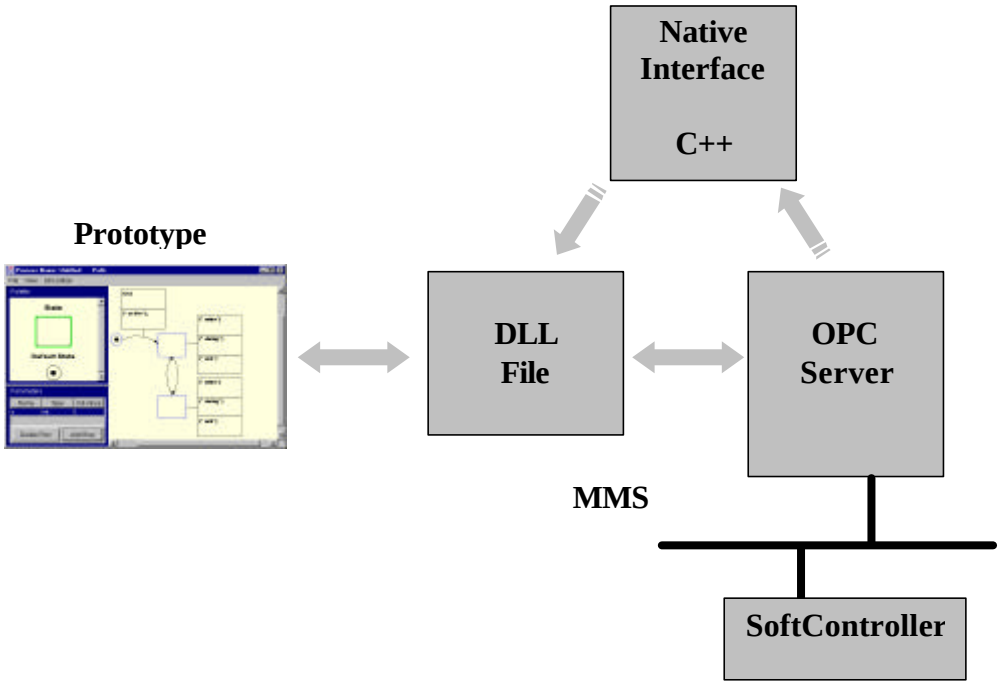


Figure 8.6 Variable values are collected from the Soft Controller directly from the prototype during execution. A MMS Server provides for the connection between the Soft Controller and the OPC Server.

A controller is a stand-alone unit in the Control ^{IT} that executes an application developed in the Control Builder. In the actual project it is possible to get current values of variables and parameters during execution by creating an OPC Client that communicates with the OPC Server. However, some additional modifications must be made, compared to the case when a project was directly simulated in the Control Builder. An OPC Server has to be started and a controller must be running. A so-called Soft Controller is used as a controller in the actual case. The Soft Controller simply runs in the local PC like the other applications. The project in the Control Builder must be downloaded into the Soft Controller and executed. By this, the application is executed in the Soft Controller instead of in the Control Builder itself. This looks more like a real world application where a project runs inside a real hardware unit, coupled to a separate processor. To get in contact with the OPC Client from the prototype, a native interface must be created as in the case when the generated code was transferred into the Control Builder.

An overview of the entire communication between the prototype and the Control ^{IT} is shown in Figure 8.7. The OPC Reader is a special thread, created on the Java side, whose only task is to communicate with the OPC Server. When the actual model should be executed, the code is transferred into a new application in the Control Builder and downloaded into the Soft Controller. Then the OPC Reader is started. The OPC Reader creates a group of items in the OPC Server. The items constitute those parameters and variables that are wanted to update during the execution. When the items are added to the group in the OPC Server, the OPC Reader starts to read all values at equidistant time intervals. The OPC Reader then updates the values in the prototype. When the OPC Reader has been started the other classes in the prototype do not have to think about the updating at all.

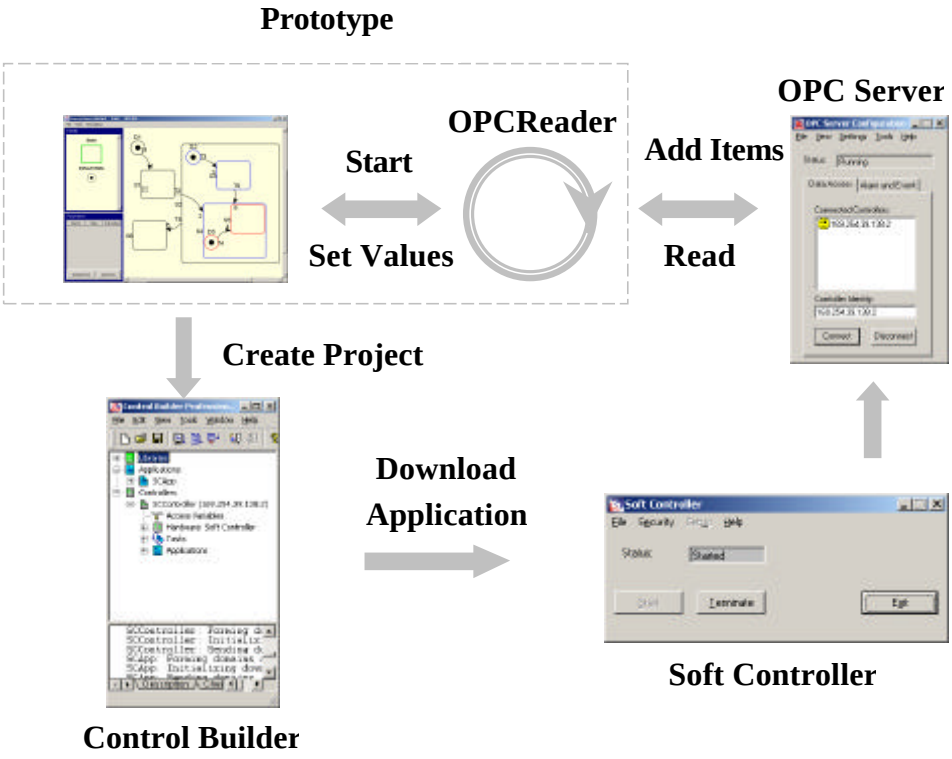


Figure 8.7 Overview of the communication between the prototype and the Control IT products.

8.4 Experiences during test

When an application is downloaded into the Soft Controller it takes some time before the OPC Server is updated with the new application. This can be noticed when the OPC Reader add items to the OPC Server. In some occasions the OPC Server rejects the adding because is does not recognize the parameters. However, after a few attempts the OPC Server accepts and adds the items correctly.

One problem with the execution is that the Soft Controller starts running immediately when the application is downloaded from the Control Builder. Since it takes some time before the updating is started in the editor, the execution cannot be followed from the beginning. Using a Boolean flag could easily prevent this drawback. The flag could be manually or automatically set when the update facility has been started.

Another property that would be desirable is to set the update rate. There are actually two rates to set when changing this property. First the update rate in the associated task must be set. This rate controls the actual time intervals between consecutive steps in the evaluation of the model. The other rate is the rate between updates in the prototype. This rate is possible to set in the OPC Reader class, which provides a method where this property can be manipulated.

9. Improvements and Future Development

9.1 Improvements

- The graphical representation is not validated for correctness regarding the notation rules of Statecharts. It is for instance never checked that a super state of XOR type contains exactly one single DefaultNode. It is not checked that a super state of AND type contains no DefaultNodes.

- The prototype assumes that code residing inside code blocks is correct Structured Text code. The code is never checked for correctness before sent to the Control Builder. If code blocks contain any errors the Control Builder detects these and execution will never start.

- When an XML document containing a model representation is loaded into the prototype no validation check is performed. The prototype assumes that the document has been stored by the prototype itself and detects no errors if the document is invalid.

- When the prototype is exited the model residing in the editable area is discarded without warning. The user is never asked to save the model. Therefore, the user must be very careful not to unintentionally discard a model during program exit.

- The prototype provides no undo or redo facilities.

9.2 Future Development

- The Statechart Editor Prototype implements only the main functionality defined in Statecharts. The notion of History, which is one of the special properties of Statecharts, has not been implemented at all.

- Terminating States are important to implement to stop execution in the model.

- There exist special Condition Connectors in Statecharts that could be used to represent conditions for transitions. Complicated expression can be stated in such Condition Connectors instead of in transitions, thus keeping the model clean.

- Actions could be scheduled for execution at specific moments in time.

- In the present prototype three hierarchical levels can be used. Additional levels could be added without major problems.

- The generated code is inserted into the XML document as an entire code block. The generated code needs to be structured in separate tags in the XML document to simplify fault-detection.

- As the Control Builder is developed using Microsoft Visual Studio .Net it would be preferable to implement the Statechart editor on this platform. Using Go.Net instead of JGo could perform this.

10. Summary and conclusions

Statecharts is a powerful tool when modelling discrete event systems. Statecharts can be used to represent different execution modes and mode changes of a system. The hierarchical structure of Statecharts could be used to model either according to Top-Down or Bottom-Up concept. Although a powerful tool, Statecharts is rather easy to use. Therefore Statecharts is suitable to use also for less experienced programmers. The execution of a Statechart model could be translated into Structured Text code and executed in the Control ^{IT}. It is also possible to read actual values of parameters and variables from a model executing in a Soft Controller. The progress of these values could be followed directly in the graphical model during execution. The flow of control is much easier to study directly in the Statechart model than in the executable code.

11. References

ABB Automation Products AB (1999): *Control IT, Beginner's Handbook Version 2.0*

Alhir, Sinan Si (1998): *UML In a Nutshell*; O'Reilly & Associates, Inc

Crilfe, Anders (2001): *Implementation Proposal Open Interface;* ABB Automation Products AB

Harel, D (1986): *STATECHARTS: A Visual Formalism For Complex Systems,* Department of Applied Mathematics, The Weizman Institute of Science, Rehovot, Israel

Harel, D; Naamad, A (1996): *The STATEMATE Semantics of Statecharts*, ACM Trans. Soft. Eng.

Harold, Elliote Rusty; W.Scott Means (2001): *XML In a Nutshell;* O'Reilly & Associates, Inc

Lewis, R.W (1996): *Programming industrial control systems using IEC 1131-3*

Mathworks Inc (1997): *StateFlow For use with Simulink;* Users Guide

Mathworks Inc (1998): *Stateflow online help*

Northwoods Software Corporation (2001): *JGo™ Graphical Object Editor Classes, User Guide*

OPC Foundation (2000): *Data Access Custom Interface Standard*

Petri, C.A (1962) : *Kommunikation mit Automaten.* PhD thesis, Schriften des IIM Nr.2, Bonn

Rumbaugh, J; M. Blaha; W. Premerlani; F. Eddy; W. Lorensen (1991): *Object Oriented Modelling and Design;* Prentice Hall, Inc

Årzén, K.E. (2000): *Real-Time Control Systems, Lund*

12. APPENDIX

12.1 APPENDIX 1 - Language Elements

Event Elements

Events	Triggers when:
en (S)	State S is activated
ex (S)	State S is deactivated
st (A)	Activity A is started
sp (A)	Activity A is stopped
ch (V)	The value of a parameter is changed
tr (C)	The value of condition C set to true
fs (C)	The value of condition C set to false

Table A.1 Event elements, defined in Statecharts.

Condition Elements

Condition	True when:
in (S)	System resides in state S
ac (A)	Activity A is active
not C	Condition C is false
C1 and C2	Condition C1 and C2 is true
C1 or C2	Condition C1 and/or C2 is true

Table A.2 Condition elements, defined in Statecharts.

Action Elements

Action	Performs:
E	Event E is generated
tr! (C)	Condition C is set to true
fs! (C)	Condition C is set to false
V:=EXP	Assign the value of EXP to parameter V
st! (A)	Activate activity A
sp! (A)	Deactivate activity A
hc! (S)	Erase the history of state S
dc! (S*)	Erase the history of state S and all descending sub states.

Table A.3 Action elements, defined in Statecharts.

12.2 APPENDIX 2 – XML-representaion of a graphical model

```
<?xml version="1.0" encoding="UTF-8"?>

<Process name="Untitled" location="C:\Gert-
Ola\ABB\Statecharts\model1.xml" stateCounter="2"
linkCounter="3" defaultNodeCounter="1" x="0" y="0">

<StateNode id="S1" type="0" superstate="" x="91" y="182" width="70"
height="60"
enterActionText="(* enter *);" duringActionText="(* during *);"
exitActionText="(* exit *);" />

<StateNode id="S2" type="0" superstate="" x="264" y="184" width="70"
height="60"
enterActionText="(* enter *);" duringActionText="(* during *);"
exitActionText="(* exit *);" />

<DefaultNode id="D1" x="83" y="318" />

<Link id="T1" from="D1" to="S1" conditionText="true" actionText="(*
action *);" priority="1" />

<Link id="T2" from="S1" to="S2" conditionText="true" actionText="(*
action *);" priority="2" />

<Link id="T3" from="S2" to="S1" conditionText="true" actionText="(*
action *);" priority="3" />

<Parameter Name="a" Type="int" InitValue="1" />

</Process>
```