DCHARTS, A FORMALISM FOR MODELING AND SIMULATION BASED DESIGN OF REACTIVE SOFTWARE SYSTEMS

Huining Feng
http://msdl.cs.mcgill.ca/people/tfeng/

Supervisor: Professor Hans Vangheluwe

February, 2004

School of Computer Science
McGill University, Montréal, Canada

A Master's Thesis Submitted in Partial Fulfillment of Requirements for
the Master of Science Degree

Abstract

DCharts, a formalism for modeling and simulation of complex reactive software systems, is proposed and studied. The DCharts formalism is based on UML statecharts and DEVS, but provides better modularity and expressiveness. DCharts semantics is rigorously defined in both an operational way and in a denotational way. Abstract, textual, and visual syntax for DCharts are presented.

SVM, a DCharts simulator implemented in Python, is presented. It accepts textual model descriptions and simulates them. Multiple types of simulations, as well as real-time execution, are discussed in detail with examples. Model verification is supported by means of repeated simulations in SVM and rule-checking of the simulation traces with extended regular expressions.

SCC is a tool to synthesize executable code from DCharts models. It statically optimizes the models to achieve high run-time performance. Multiple target languages are supported.

Applications of the DCharts formalism are studied, by means of the the above-mentioned tools. They demonstrate how DCharts are ready for practical use.

Contents

List of Figures

List of Tables

1 INTRODUCTION

As software systems and hardware systems are becoming more and more complex nowadays, a systematic approach for the development of physical as well as software systems is needed.

As we look back at the history of software development, there have been three revolutions which greatly improved productivity and quality. The first revolution was the Fortran language. Two important concepts were introduced in Fortran: structured programming and variable names. With those concepts, programmers no longer mixed data and code in a program. They started to think in a more modular way instead of directly writing assembly code or machine code, which is hard to understand or debug for human beings. As a result, both productivity and quality of the systems were improved.

The second revolution started in the 60's with Algol and reached its peak somewhere in the 80's when the dominating languages were Pascal and C. Those languages eliminated all column-based formatting. They provided well-designed high-level control structures such as "`while`" loops and "`for`" loops. This programming was much more structured, and the use of the "`goto`" statement was widely criticized for breaking the structure (or the modularity) of the programs.

The third revolution was object-oriented programming (OOP). The OOP concept originated in the Simula language emerging in 1967. C++ matured this idea and made it practical. Programmers started to think in a more modular way and to reuse existing code to a greater extent by means of encapsulation and polymorphism. Encapsulation emphasizes the distinction between behavior and interface. Data is divided and maintained in different classes according to their semantics. These ideas help guarantee the integrity of a logical piece of software, and make it more stable. Polymorphism allows better reusability. The behavior of a whole class or part of it can be reused by means of inheritance. Overriding allows to modify part of the existing behavior of a class and fit it to a new application. These ideas greatly improve productivity.

However, as new demands arise, people have seen the limitations of OOP, or software programming in general:

- code has to be written by hand and is thus error-prone;
- it is impossible to prove the correctness of a system because of too much detailed information in the program;
- coding is too labor-consuming and once a high-level error is discovered, it is not easy to go back to the design phase of the development process.

Neither structured programming nor object-oriented programming solves the above problems, due to the fact that such problems have their roots in the high-level design instead of in the implementation. Effort has been spent on discovering systematic methods for system design. This effort leads to the research in modeling and simulation based design.

This chapter presents a general introduction to modeling and simulation. In particular, existing formalisms for this purpose, such as statecharts and DEVS (Discrete EVent Systems specification), are discussed. They are the starting point of this thesis work.

1.1 Modeling and Simulation

Modeling and simulation are enablers for principled (software) system design.

From the modeling point of view, implementation details are not interesting and are thus neglected by designers who reason at a high level of abstraction. By neglecting this information, a system can be modeled in formalisms such as statecharts and DEVS. Such formalisms are much more abstract and formal than a piece of code written in a specific programming language. Certain properties, such as reachable states or deadlocks, can be proved or disproved with the assistance of model-checking tools. In this way, most of the problems are solved in the design phase, and manual implementation is kept minimal. All these result in higher stability, better maintainability and less potential errors in the systems.

Some of the modeling formalisms provide a rigorous way to specify interfaces (e.g., class diagrams), interaction protocols (e.g., activity diagrams and sequence diagrams) and behavior (e.g., statecharts and DEVS). The combined use of them protects the internal structure of components (modular parts of a system), while still revealing enough information to the outside world to ensure reusability. For example, in the UML (Unified Modeling Language) 2.0, the interface of a component is defined by a class diagram. The associations between classes reflect the relations between them. Sequence diagrams are used to illustrate the interaction between different components (of the same type or of different types). They influence each other at run-time by means of messages. Finally, the full internal behavior of these components is specified with statecharts, which give an executable semantics to them. With this semantics, simulation as well as code synthesis of the system becomes possible.

Specifying the complete behavior with a formalism is analogous with implementation. However, this approach differs from the traditional implementation phase in that by specifying a system in a formal way, the designers are able to fully predict its run-time behavior and prove its correctness. The models can be simulated in appropriate environments. Code can be automatically synthesized with code generators. Those tools save human labor and greatly increase productivity.

1.1.1 Models and Meta-models

To specify a system as a model in a formalism, the two important parts of the formalism must be well-understood by all the designers and whoever wants to reuse parts of the system: the syntax and the semantics. The *syntax* enforces certain rules on every model designed in the formalism, while the *semantics* defines the concrete meaning of every model that conforms to the syntax. If the formalism is executable, its semantics provides the basis for simulation and execution; if the formalism is non-executable, its semantics helps ensure a unique interpretation of a model among multiple designers and users.

The formalism, if it is considered as a model itself, can be explicitly modeled with another formalism. In this case, the formalism to be modeled is called a *meta-model*. There are many benefits to meta-modeling. One of those is that the syntax of the modeled formalism can be very concisely and explicitly defined. For example, the syntax of FSA (Finite State Automata) can be easily modeled by an ER (Entity-Relationship) diagram as shown in Figure 1.1. This syntactic definition is much more rigorous than a definition in a natural language. A parser can be built from it, which automatically checks whether a model is an FSA.

Another benefit is that a user can easily design his/her own formalism that best fits a specific application area. With a tool capable of generating a modeling environment from a meta-model, such as AToM^3 (A Tool for Multi-formalism and Meta-Modeling) developed in the MSDL (Modeling, Simulation and Design Lab) of McGill University [1] [2], the designer gets a domain-specific modeling environment. The environment is then used to solve problems specific to the application domain. Model designers can thus make full use of their knowledge in that domain. [3]

1.1.2 The Process of Modeling and Simulation Based Design

Modeling and simulation based system development requires a series of steps of transformation. In each step a model is transformed into another one. The new model is usually in another formalism at a lower level of abstraction. This takes the original design along the way from a very abstract model (derived from user

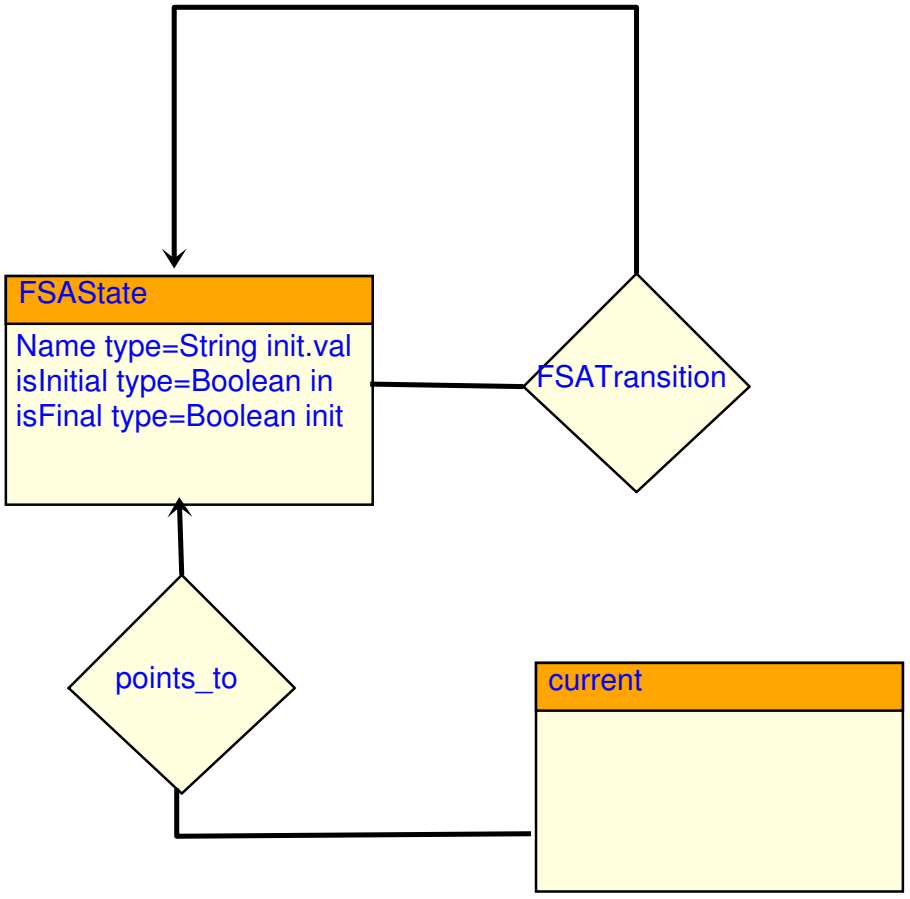

Figure 1.1: The Finite State Automata syntax in an Entity-Relationship diagram

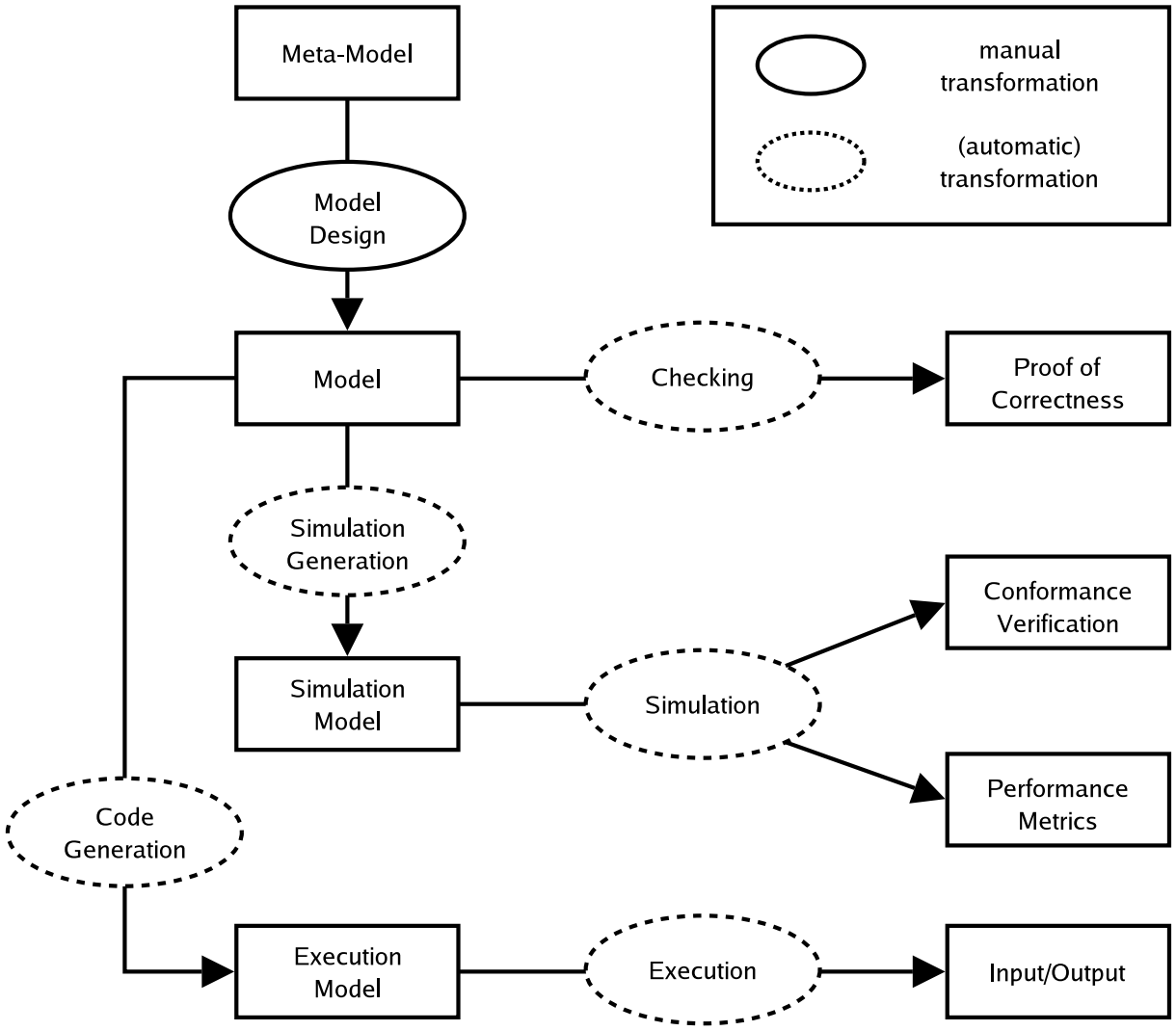

Figure 1.2: Modeling and simulation based design process

requirements) to the simulation model and eventually, application code.

Figure 1.2 illustrates the system development process and the different transformations involved in it. Usually, a user (the designer of a system) starts from an existing formalism, and defines his/her own model based on the syntax and semantics of that formalism. However, in case no existing formalism is suitable to specify the system, the user may define a meta-model with a meta-modeling tool, such as AToM^3, and further design the model with the formalism defined by that meta-model.

After the “model design” phase, which requires manual work of the designer, automatic transformations can be done to obtain different models for various purposes. Tools can be used to check the correctness of the model. Those tools actually generate *checking models*, which give the designer such information as reachable states, deadlocks and reactiveness to every possible event in every state. The designer has to modify the model if potential errors are found in the checking. This is much easier than the debugging in the traditional development process, where code is written and debugged manually.

Tools that generate *simulation models* enable simulation of the model. Simulating a model requires detailed information about the model execution. If this information is not given in the original model, it must be added at the time when the simulation models are generated. Executable formalisms, such as statecharts and DEVS, allow to fully specify this information in the models. Interpreters of such formalisms are thus

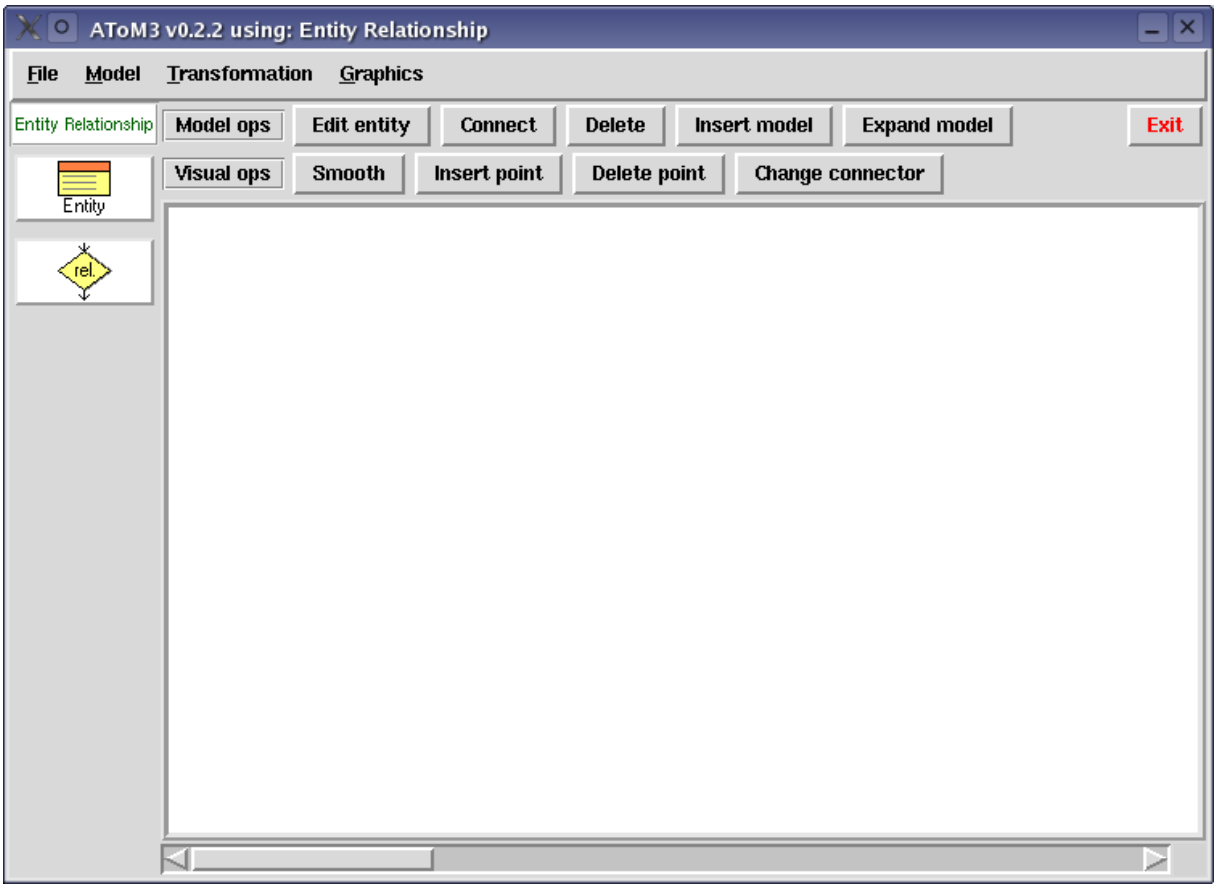

Figure 1.3: AToM^3 meta-modeling environment with the Entity-Relationship diagrams meta-model loaded

able to simulate the original designs. Simulation can be used to increase confidence in the correctness of the model. Commonly, simulation is used to calculate performance metrics. These can be used to tune model parameters to satisfy system performance requirements.

Another kind of tools take the original models as input and generate executable code. The code is optimized and efficient, but usually platform-dependent. Its execution does not require the support from an underlying environment, as simulation does. The purpose of this code generation is to maximize performance and to release the well-developed system to the end-users, who are not interested in the model design.

1.1.3 Modeling and Meta-modeling in AToM^3

AToM^3 [1] [2] is a tool for modeling, meta-modeling and simulation. It is developed by Prof. Hans Vangheluwe at the MSDL (Modeling, Simulation and Design Lab) of McGill University in Canada in close collaboration with Prof. Juan de Lara at the Autonomous University of Madrid. It allows building and dynamically loading meta-models in its graphical environment. When a meta-model is loaded, the graphical environment is modified according to the allowed entities of the formalism. The user can design models according to the syntax of the formalism. Transformations between models of different formalisms are handled with graph grammars, a powerful formalism to specify transformations in a graphical form. With the support of a simulation engine that implements the semantics of the loaded formalism, AToM^3 can also be used as a simulation environment.

Figure 1.3 shows the main window of AToM^3 with the Entity-Relationship diagrams meta-model loaded in it. The left panel of AToM^3 shows only the buttons of allowed entities. In this case, “entity” and “relation” are two different kinds of entities in an Entity-Relationship diagram.

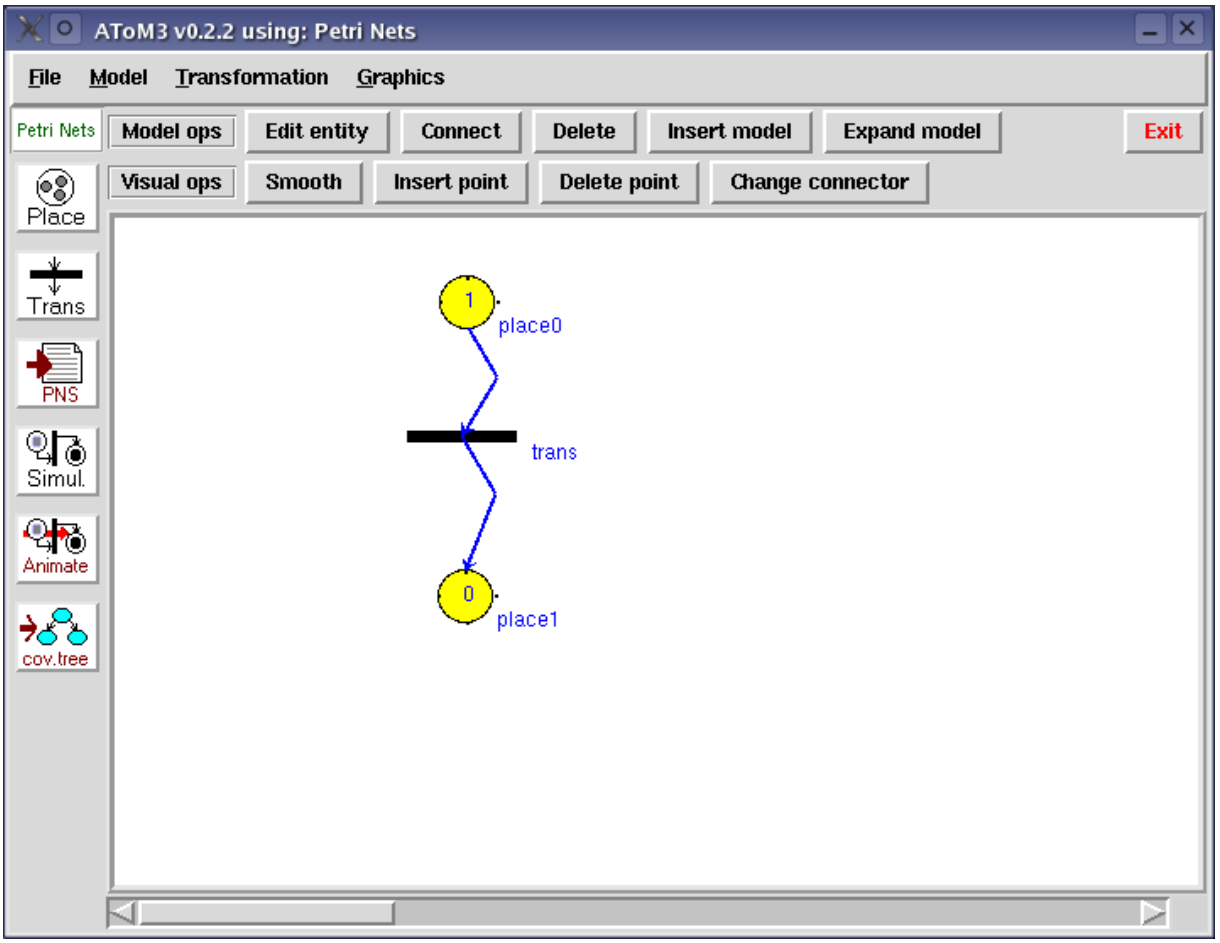

Figure 1.4: AToM^3 meta-modeling environment with the PetriNet meta-model loaded

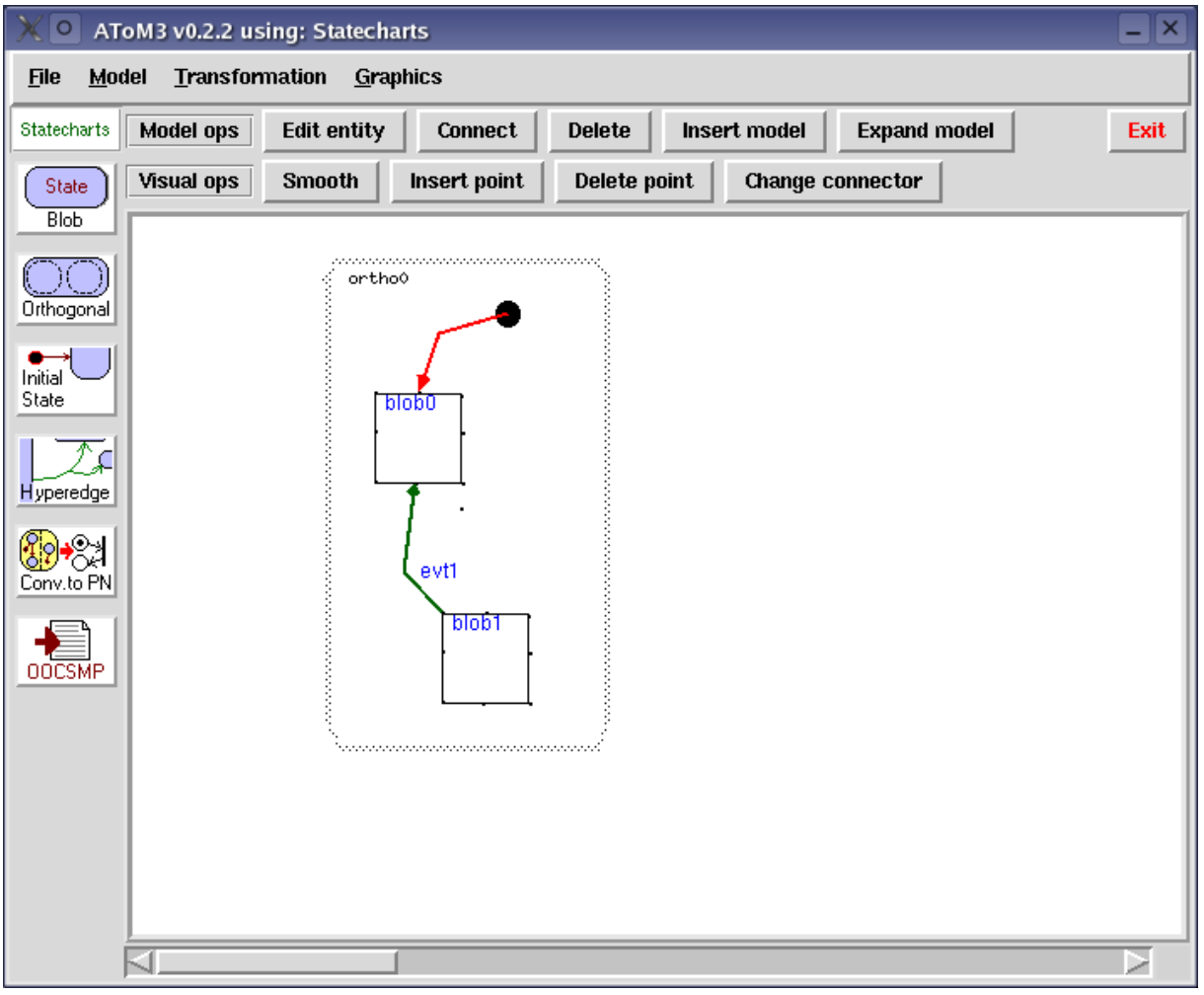

Figure 1.5: AToM^3 meta-modeling environment with the statecharts meta-model loaded

Figure 1.4 and Figure 1.5 show the AToM^3 environment with the PetriNet meta-model and the statecharts meta-model loaded in it, respectively. The buttons shown on the left panel vary with the loaded meta-models. Simulation in AToM^3 is discussed later.

1.2 The Statecharts Formalism

Statecharts, introduced by David Harel [4], are a visual and executable formalism for modeling complex reactive systems. It has roots in the Finite State Automata (FSA) formalism and adds new concepts to it. Those new concepts make the formalism suitable for specifying discrete event systems.

1.2.1 Finite State Automata

The syntax of the FSA formalism is defined by means of meta-model in Figure 1.1. An FSA consists of states and transitions between states. A state has three properties:

- `Name`, a string that denotes the unique ID of a state.
- `isInitial`, a boolean that decides whether a state is the initial state. There must be exactly one initial state in each FSA model.
- `isFinal`, a boolean that decides whether a state is a final state. There must be at least one final state in each FSA model.

A transition, when *triggered*, changes the model from one state (*source state*) to another (*destination state*). (The destination state may be the same as the source state.) In the ER meta-model, a transition is represented as a relation between states.

Input symbol is a property of transitions, which is not visible from the graphical representation of the meta-model. It defines a single symbol that triggers the transition. When that symbol is received and the model is in the source state of a transition (the starting of the arc), the transition is triggered, and the model changes to the new state (the ending of the arc with an arrow).

The input symbols are taken one by one from an input sequence. If there is no *enabled* transition (a transition is *enabled* if and only if the model is in its source state and the current input is its input symbol) for one of the symbols, an error is raised, and the FSA halts. This error means the FSA does not accept such an input sequence, or more formally, the *language* (the set of accepted input sequences) defined by the FSA does not include such a *sentence* (one single input sequence ended with an *end mark*).

The FSA formalism also requires that when an accepted input sequence ends, the FSA must be in one of the final states. Otherwise, the input sequence is not accepted.

A simple FSA example is shown in Figure 1.6. Its initial state is state 1 (a state with a black dot pointing to it). Its final state is state 5 (a state with a double-line border). This FSA accepts the language:

$$\{af, ae(dc)^*g, b(cd)^*cg\}$$

As a result, *af*, *aedcg*, *aedcdcg*, *bcg*, *bcdcg* are all accepted sentences, while *a*, *aedg* and *bcd* are unaccepted.

1.2.2 Statecharts Extensions to FSA

David Harel has added extensions to FSA to make it a practical and expressive formalism [4]. With those extensions, it becomes possible to specify the complete reactive behavior of a system in a model.

The meta-model of statecharts is drawn in the ER diagram in Figure 1.7. Elements in the diagram are described below:

- The name `Blob` is used to distinguish hierarchical states in statecharts from states in FSA. Since hierarchy and orthogonal components are introduced in statecharts, the states may have inner structures and are known as blobs. A blob has a `Name` attribute.

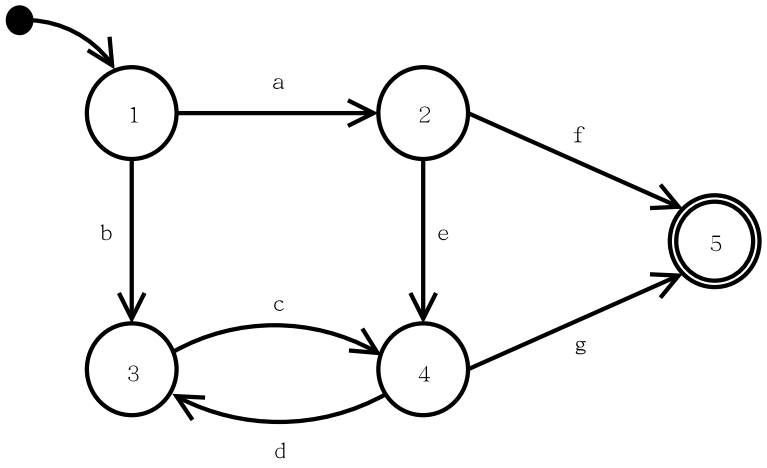

Figure 1.6: A simple FSA example

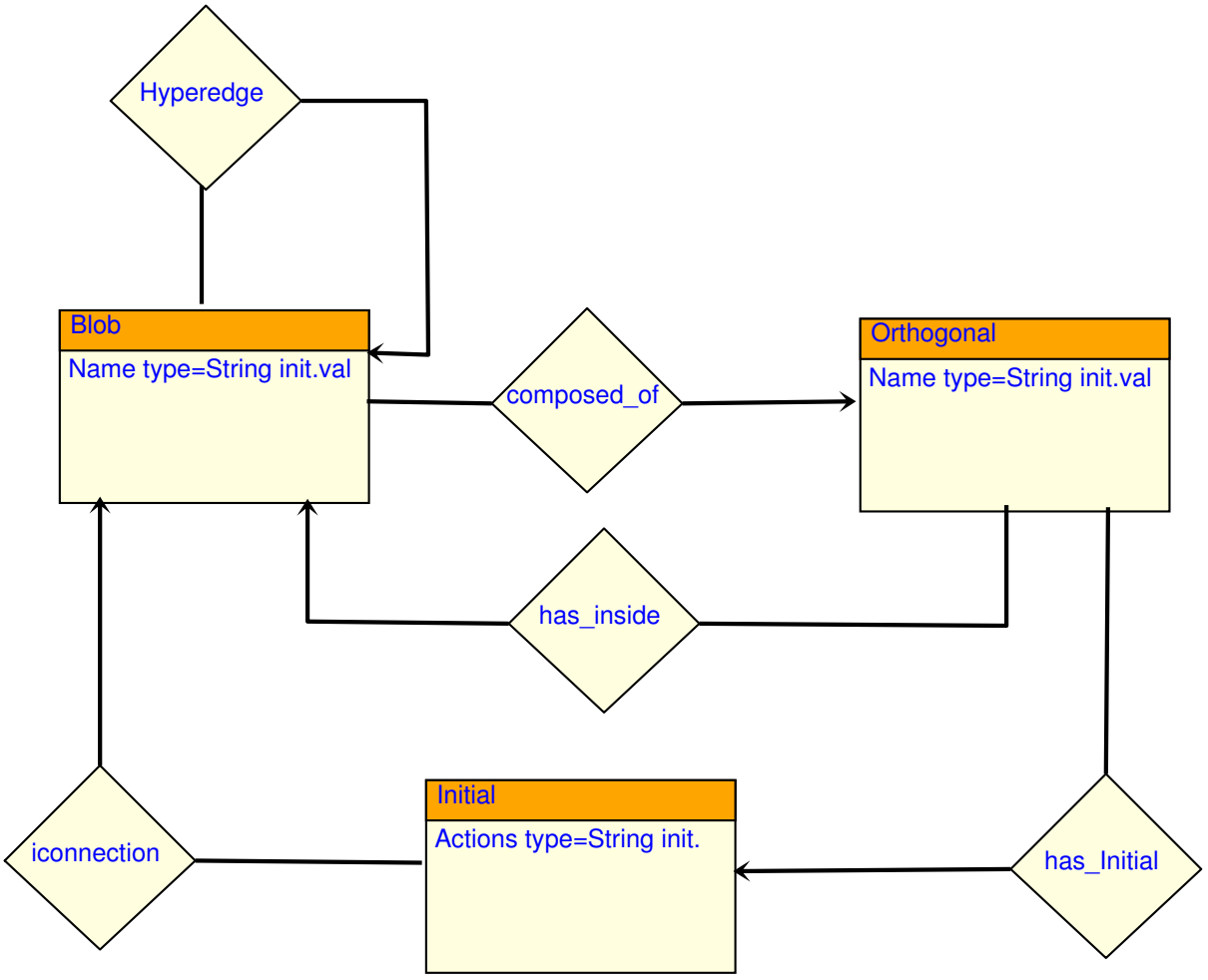

Figure 1.7: The statecharts meta-model in an Entity-Relationship diagram

S	a set of admissible states
$ta : S \rightarrow R^{+}_{0,+\infty}$	time advance function
$\delta_{int} : S \rightarrow S$	internal transition function
X	a set of admissible external inputs
$\delta_{ext} : Q \times X \rightarrow S$	external transition function
	where $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$
Y	a set of possible outputs
$\lambda : S \rightarrow Y \cup \{\emptyset\}$	output function

Table 1.1: Atomic DEVS $< S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda >$

- A `Hyperedge` connects two states and denotes a transition between them. A transition has an `event` property, and may or may not have properties `guard` and `output`. Those properties and their meanings are discussed later.
- A blob may consist of one or more orthogonal components. Each of them maintains a local current state. The Cartesian product of the current states of all the orthogonal components belonging to the same parent is equal to the current state of the parent state. An orthogonal component may have blobs inside it.
- Like FSA, every statecharts model has an initial state. Within every blob or orthogonal component there is also a default state.

The statecharts semantics has many variants. One popular semantics is David Harel's STATEMATE semantics [5]. Another one is the statecharts semantics described in the UML (Unified Modeling Language) [6]. These variants are not compatible with each other. There will be more discussion about the statecharts variants in the latter part of this thesis work.

1.3 The DEVS Formalism

DEVS (Discrete EVent Systems specification) was created by Bernard Zeigler [7] [8]. It is a modular formalism for deterministic and causal systems. It allows for component-based design of complex systems. A DEVS model may contain two kinds of DEVS components: Atomic DEVS and Coupled DEVS. An Atomic DEVS does not contain any component in it. It only has a mathematical specification of its behavior. A Coupled DEVS is a modular composition of one or more Atomic DEVS'.

According to the closure under coupling property of DEVS, a Coupled DEVS can be substituted by an Atomic DEVS with equivalent behavior. A Coupled DEVS can be used to compose more complex DEVS components.

A Coupled DEVS specifies connections between the components in it. Two connected components send messages via well-defined ports.

1.3.1 Atomic DEVS

An Atomic DEVS is a functional atom in a model, which cannot be further divided into sub-components. Its behavior is described by implementation-independent mathematical functions and symbols.

Atomic DEVS is a tuple $< S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda >$ as shown in Table 1.1. All the states of the DEVS are in the admissible state set S. An execution of the model is to sequentially change its states, until ended explicitly. The change in its states is defined by two functions: internal transition function δ_{int} and external transition function δ_{ext}.

$\delta_{int} : S \rightarrow S$ defines the autonomous internal behavior. The time when these changes take place is defined by function $ta : S \rightarrow R^{+}_{0,+\infty}$. It takes a state as a parameter and returns a non-negative real value denoting the time interval between state changes. The time for a DEVS is not discrete, because the simulation is not

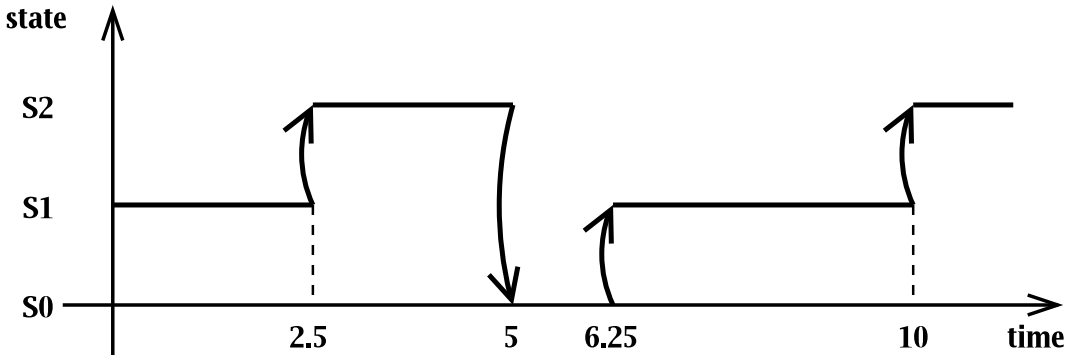

Figure 1.8: Atomic DEVS state trajectory

X_{self}	a set of admissible external inputs
Y_{self}	a set of possible outputs
D	a set of unique component references
$\{M_i \mid i \in D\}$	a set of components
$I_i, i \in D$	a set of influencees of component i
$\{Z_{i,j}\}$	a set of output-to-input translation functions
$select : 2^D \rightarrow D$	the select function

Table 1.2: Coupled DEVS $< X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select >$

based on time-slicing. An internal transition can be scheduled at any point in the future on the real time-line (Figure 1.8).

An external event may occur at any time. $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function. It defines which new state the DEVS should be changed to, when a certain external event is received. The new state depends on the old state and how long the DEVS has been in the old state (elapsed time). The old state s and the elapsed time e are usually represented as a tuple (s, e), where $s \in S$ and $0 \leq e \leq ta(s)$.

Only internal transitions are allowed to produce output. Y is the set of all possible output values. The output produced by a transition from the old state s to any other state can be calculated with the λ function. Value ø means no output is produced.

All the input values are defined in X and all the output values are defined in Y. They can be viewed as an interface exposed to the outside world. The outside world communicates with the Atomic DEVS only through input events and output events.

1.3.2 Coupled DEVS

As the coupling of one or more Atomic DEVS', a Coupled DEVS is a tuple

$$< X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select >$$

as described in Table 1.2.

M_i (where $i \in D$) is an Atomic DEVS as one of the Coupled DEVS' components. Its output is connected to the input of its influencees I_i (a set). Every output signal of component i is translated by the $Z_{i,j}$ function before it reaches an input of component j.

It is possible that internal transitions in different components occur at exactly the same time. The virtual time is not advanced until all these events at the same time are handled. The order in which the events are handled is important, because the change of state caused by a transition may affect the behavior of

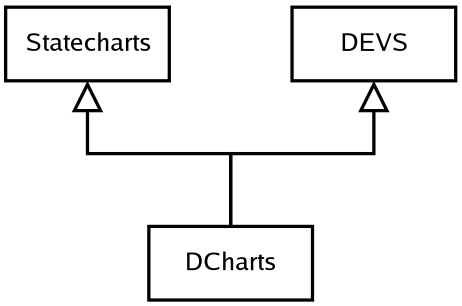

Figure 1.9: Generalization of DCharts

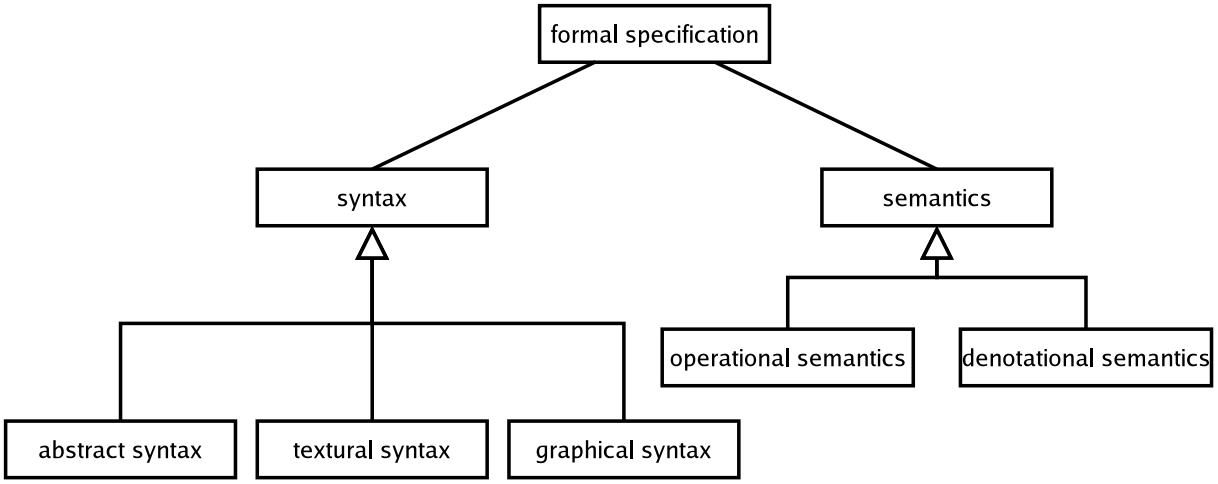

Figure 1.10: Specification of DCharts

subsequent transitions, though they are triggered at the same virtual time. The *select* function decides which transition must be triggered first when a conflict occurs.

DEVS is closed under coupling by construction. A Coupled DEVS can be rewritten as an equivalent Atomic DEVS, and thus be reused as a component in a larger DEVS. The outside world need not know whether a DEVS component is atomic or coupled, because both kinds expose the same interface.

1.4 Research Focus

The research and its results discussed in this thesis builds on the existing formalisms and tools. In particular, it is closely related to both statecharts and DEVS. It combines the syntax and semantics of statecharts and DEVS in a modular way (Figure 1.9), and provides a friendly user-interface and good expressiveness to model designers.

DCharts are a new executable formalism, which allows model design, model transformation, model simulation, model checking and verification, and code generation.

1.4.1 Formal Specification

The syntax and semantics of DCharts are formally specified in this thesis (Figure 1.10).

Three types of syntaxes are described:

- The abstract syntax is a symbolic language. It gives a symbol to every entity in DCharts. Relations are regarded as functions. This syntax is formal and it allows logical reasoning or inference on DCharts models. It also makes it possible to mathematically transform or simplify DCharts models.

- The textual syntax defines the textual form of the formalism. This form simplifies computer processing of a DCharts model. The textual syntax is implemented in this thesis work with a parser and interpreter.
- The graphical syntax gives a graphical representation for the formalism. It provides a way to design DCharts models in a modeling environment that supports GUI (Graphical User Interface), such as AToM^3. In many cases graphical model design is more user-friendly and understandable. Part of the graphical syntax is implemented in a DCharts meta-model in AToM^3, so that designers can manipulate DCharts models in this environment.

The semantics of DCharts is formally defined in two different ways:

- *Operational semantics* defines the meaning of DCharts models with a functional description or pseudo-code. From this description, an interpreter (SVM, Statechart Virtual Machine) and a code synthesizer (SCC, StateChart Compiler) for DCharts are constructed.
- *Denotational semantics* maps DCharts to other existing formalisms such as statecharts and DEVS. The DCharts semantics is made clear provided that the formalisms that we map onto have well-defined semantics. Denotational semantics provides a way to transform DCharts models into models in other formalisms.

1.4.2 Model Transformation

There are two kinds of model transformation. Intra-formalism transformations transform a model into another model in the same formalism. The result of this transformation is usually optimized for modularity or efficiency. Inter-formalism transformations transform a model into a new model in another formalism. The new model can thus be reused in the systems designed in the other formalism. The possibility of such transformations gives a meaningful comparison of expressiveness between the two formalisms. Another benefit of inter-formalism transformations is that, by transforming a model into a more extensively studied formalism, the model checking tools of that formalism can be used to prove certain properties of the model.

The model transformations discussed in this thesis are inter-formalism transformations. Intra-formalism transformations are not discussed.

Spencer Borland in his Master's thesis [9] has shown an approach with which statecharts models can be transformed to DEVS. This helps prove that DCharts can be transformed to DEVS, because DCharts are a modular combination of statecharts and DEVS. More transformations between statecharts, DEVS and DCharts are discussed in the later part of this thesis.

1.4.3 Simulation

DCharts are executable. Every DCharts model has a rigorous semantics and can be simulated in a simulation environment such as SVM (Statechart Virtual Machine), which is discussed later in detail.

As an overview of model simulation, Figure 1.11 illustrates the different simulation and execution strategies as a three-dimensional matrix. The meaning of the axes is described here:

- The *x* axis indicates whether the simulation (or execution) is sequential or parallel. *Sequential simulation* is step by step simulation that guarantees no overlap of two or more operations (such as change of states and execution of action code) at the same time. *Parallel simulation*, however, strives to perform operations in a parallel way typically to maximize performance. Due to the sequential nature of most of the models, it is very hard to tell which operation can overlap with another. The overhead of finding out potential overlapable operations and synchronizing different parallelized parts is usually so high that the simulation slows down rather than speeds up.
- The *y* axis indicates whether the simulation (or execution) is local or distributed. *Local simulation* is done on a stand-alone process. There are three kinds of local simulation: single-threaded, multi-threaded, and multi-process. In *distributed simulation*, multiple processes are involved in a single sim-

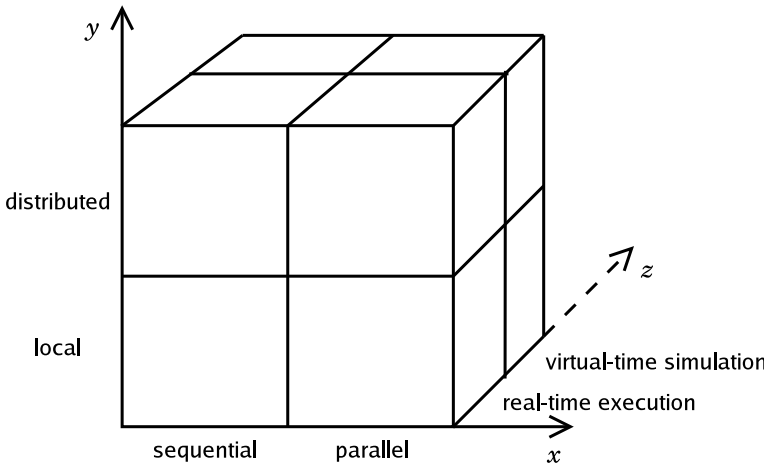

Figure 1.11: Matrix of simulation and execution

ulation. Components participating in the simulation are deployed among those processes in a modular way. They communicate with each other by means of messages via a network.

- The *z* axis indicates whether it is a virtual-time simulation or real-time execution. *Real-time execution* is synchronized with the real time (or wall-clock time). It may need to satisfy several time constraints in order to guarantee the real-time behavior of the model. The satisfaction of those time constraints usually requires support from the underlying operating system. *Virtual-time simulation* uses a timer that is usually not synchronized with the wall clock. If the timer is proportional to the real time, the simulation is called *scaled real-time simulation*; if the timer is a counter that keeps track of the time and it is advanced as fast as possible (i.e., as soon as all the components are waiting for their scheduled events), the simulation is called *as-fast-as-possible simulation.*

It is important and interesting to know all the combinations of these schemes, though only some of the combinations are reasonable, and only some of the reasonable combinations are relevant to this thesis work.

- *Sequential local real-time execution*. This type of execution is natively implemented in SVM, the interpreter for DCharts. All the operations are sequentially executed on a multi-threaded process. Different threads are synchronized to guarantee that only one operation is performed at a time.
- *Sequential local virtual-time simulation*. The two types of virtual time (scaled and as-soon-as-possible) are not directly supported by the SVM simulation kernel. However, scaled real-time simulation can be easily simulated with real-time simulation. This can be accomplished by redefining the macro that retrieves or schedules time (with the $after$ event), as discussed later. The later part of this thesis also shows that as-soon-as-possible simulation can be simulated with a *clock component* (section 9.2). As a result, there is no need to internally implement this kind of simulation.
- *Sequential distributed real-time execution.* For distributed execution, it is natural to allow parallel behavior between components on different computers. Sequential behavior can be simulated with parallel execution by means of global semaphores or a global clock component. However, this makes the execution less efficient than sequential real-time execution on a single computer (because of latency in the network and the overhead of synchronization). Because it is rarely useful, sequential distributed real-time execution is not directly supported by the SVM.
- *Sequential distributed virtual-time simulation*. For the same reason of inefficiency, sequential distributed virtual-time simulation is not supported. The users should use sequential local virtual-time simulation instead.

- *Parallel local real-time execution.* SVM provides support for multi-process simulation on a single machine. Those processes are highly parallel. They influence each other in the form of messages via ports. The execution is real-time so that each of the processes directly accesses the time given by the computer hardware.
- *Parallel local virtual-time simulation.* Virtual time is not directly supported by SVM. However, with a special clock component running as a separate process and providing time service to all the other processes, the two types of virtual-time simulation are made possible. As a result, there is no need to implement this kind of simulation internally in the SVM.
- *Parallel distributed real-time execution.* This kind of execution directly corresponds to distributed software systems and is thus interesting. SVM builds parallel distributed real-time execution on top of PVM (Parallel Virtual Machine) [10]. Ports are defined on the boundary of components. Individual components have parallel behavior. They communicate with messages sent via connections between ports over a network.
- *Parallel distributed virtual-time simulation.* This kind of simulation is simulated with parallel distributed real-time execution with an additional clock component. The clock component reveals its ID to all the other components and provides global timing service to them. This clock component is discussed later in general.

From this discussion, it is easily seen that SVM is a powerful simulation tool that supports most of the simulation schemes, though some are simulated by the others with extra components. The concept of a clock component is important because it reduces the requirements on the simulation engine. The simulator is thus minimal and optimizable.

1.4.4 Model Checking and Verification

Model checking refers to proving properties of the models without simulation or execution. For example, by enumerating all possible event sequences accepted by a state machine, the *dead states* (the states that the model never goes to) are discovered and deleted. Another example is by building a reachability graph of a PetriNet model, it can be easily proved whether or not the model allows deadlock (a state of which the model, once enters it, can never go out).

Model checking of DCharts is not easy, mostly because models in DCharts contain too much information about the execution detail. Usually, a model has to be abstracted and the irrelevant information in it must be removed before checking can be performed. This could be done by means of transforming the model into model(s) of other formalisms, such as DEVS and PetriNets. Because of its difficulty, model checking is not discussed in this thesis.

Another approach to find out properties of models and demonstrate their correctness is model verification. It is done by simulating or executing the models multiple times. A tool that analyzes the gathered output trace tells whether the models are running correctly or not. It may also analyze the performance of the models and discover possible bottle-necks in them.

1.4.5 Code Synthesis

The purpose of code synthesis is to maximize run-time performance. It is always much slower to simulate a model in an interpreted way than to execute the compiled code directly.

A lot of optimization can be done on the models at the time of code synthesis. This issue will be discussed thoroughly in this thesis. In particular, SCC (StateChart Compiler), a code synthesizer for DCharts, is implemented. It is able to generate Java, C++, Python and C# source code from textual DCharts model descriptions.

1.5 Related Work

This thesis work is done in the MSDL (Modeling, Simulation and Design Lab) of McGill University, headed by Prof. Hans Vangheluwe. It is closely related to other on-going projects in the MSDL:

- AToM^3 [1] [2] is a graphical modeling and meta-modeling environment. It is able to meta-model the syntax of many formalisms, as well as generate dedicated visual environments for the model design in those formalisms. The semantics of some of those formalisms, such as PetriNets, is usually modeled with graph grammars. In this way, AToM^3 can also be used as a simulation or execution environment, with graph grammars that transform the model from one state to another.

  A DCharts meta-model is built in AToM^3, which defines a subset of the DCharts syntax. This meta-model is discussed in later chapters. The semantics of DCharts is implemented in SVM, which can be loaded in AToM^3 as a simulation engine. It makes it possible to simulate DCharts models and at the same time highlight the current states and enabled transitions in the AToM^3 visual environment.
- PythonDEVS [11] [12] is a virtual-time DEVS simulator implemented by Hans Vangheluwe and Jean-Sébastien Bolduc. It provides a practical basis for DCharts simulation, as DCharts is a modular combination of statecharts and DEVS.

  PythonDEVS is a set of DEVS templates and a DEVS simulator class. Those templates must be extended by the model designers by means of inheritance. SVM takes one step further by accepting a textual language of DCharts model descriptions. The users can easily write model descriptions conforming to a rigorously defined syntax. They may also use the AToM^3 environment to graphically model DCharts, and then generate model descriptions by pressing a button.
- Real-time PythonDEVS is the real-time version of PythonDEVS. It is modified by Spencer Borland from non-real-time PythonDEVS.
- Spencer Borland in his Master's thesis [9] describes a way to transform statecharts into DEVS and hence proves that DEVS has at least the same expressive power as statecharts [13]. (Actually, DEVS is even more expressive than statecharts.) This provides another means to simulate DCharts other than simulating them directly in SVM: transform DCharts (which take a similar form as statecharts) into DEVS, and use Real-time PythonDEVS to simulate them.
- Alison Stewart has compared the functionality between Real-time PythonDEVS and SVM, and has built an MP3 player on both of them. In her report, she concludes that though neither of the two is perfect, SVM is much more user-friendly. The report is available on-line [14].

Information about the above projects can be obtained from the MSDL website:

`http://msdl.cs.mcgill.ca/`

This thesis work is also related to several research projects outside of McGill University.

- David Harel has created the statecharts formalism [4] [5], which has been the basis of DCharts. Many of the DCharts constructs, as they are defined in latter chapters, can be found in statecharts. The semantics of those common constructs is the same in both formalisms.

  DCharts have extended statecharts to make them more rigorous and expressive. The syntax of DCharts is a superset of the statecharts syntax. The semantics of DCharts is a superset of the semantics of David Harel's statecharts. Hence, SVM is also a simulator for statecharts.
- Bernard P. Zeigler has created DEVS [7] [8], a modular and expressive formalism. The idea of blocks and connections between them via ports is reused in DCharts. As a result, DCharts are much more modular than statecharts, which are not modular in their nature. Many other concepts in DEVS are useful for the creation of DCharts. In particular, the select function is invented in DEVS to solve transition conflicts. This idea is absorbed by DCharts, though they support a different mechanism based on transition priorities to solve those conflicts.

- Ptolemy II [15] [16] [17] is a heterogeneous modeling and simulation environment implemented by Prof. Edward A. Lee and his students at EECS, University of California at Berkeley. Its viewpoint of directors and actors in component-based models and its Java code generation have important impact on the design of DCharts and such tools as SVM and SCC.

  Unlike SVM, a dedicated simulator for DCharts, Ptolemy II is a modeling and simulation environment for multiple formalisms. Different formalisms may be used to model components (or actors) in a single model. Directors manage the interaction between those components. Discrete-time components and continuous-time components are allowed to coexist and communicate in a single system in this framework.

- The Parallel and Distributed Simulation (PADS) lab of Georgia Institute of Technology, headed by Prof. Richard Fujimoto, has implemented PDNS (Parallel and Distributed Network Simulator). It is an advanced distributed simulation environment.

  The research at the PADS lab is more oriented to distributed simulation than system design. Environments for high-performance distributed simulation are being built, which support the testing and analysis of complex and large systems, such as aircrafts and global troop deployment.

  Similar functionality will be supported by the future version of SVM. It will support the distributed timewarp simulation with DCharts.

- The research on model checking is active and advanced in the Model Checking group of Carnegie Mellon University, headed by Prof. Edmund Clarke. In particular, their research on explicit state model checking [18] [19] is interesting and useful.

  As SVM currently has very limited support for model checking, studying the research results of the Model Checking group will be helpful to the future of a model checker for DCharts. The checker will be able to formally prove properties of DCharts models without simulating or executing them in SVM.

- Prof. Joanne M. Atlee, Prof. Nancy A. Day and their WatForm (Waterloo Formal Methods) research group at University of Waterloo are seeking a way to enhance the power of statecharts and to make them more expressive and practitioner-friendly. [20] [21] [22] Their work is closely related to the creation of DCharts.

  In [23], they discussed a parametrized template capable of expressing the semantics of all the statecharts variants. It is meaningful to describe their framework in DCharts. This enables SVM to simulate models of any statecharts variant.

- Prof. Ivan Porres at Software Construction Laboratory of Åbo Akademi University in Finland has developed SMW (System Modeling Workbench), “a collection of tools to edit, store, analyze and verify models.” [24] [25] Those tools are reusable and comparable with AToM^3.

  SMW with appropriate extension can be used as a visual environment to design DCharts models. With a plugin that invokes SVM, DCharts simulation in SMW is also possible.

2

ABSTRACT SYNTAX AND SEMANTICS OF DCHARTS

The definition of DCharts 1.0, the current version of DCharts, contains two parts: the syntax answers the question "*what is the structure of a DCharts model?*" while the semantics answers "*what is the meaning of a DCharts model?*" The following criteria influence the syntax and the semantics:

- The syntax must be rich enough to allow the specification of a complete semantics. I.e., if a semantic element cannot be specified according to the syntax, there is no way to design a model that uses it, and hence the element becomes useless.
- Every syntactically correct model must have a unique meaning according to the semantics.
- The definition of syntax must facilitate both computer processing and human understanding.
- The semantics should be as platform/implementation-independent as possible. This allows the formalism to be implemented by different tools and to be used in different systems.

The following sections describe the basic syntax and semantics of DCharts 1.0.

2.1 The DCharts Meta-model

Figure 2.1 shows the meta-model of DCharts in the AToM^3 meta-modeling environment. It is modified from Spencer Borland's statecharts meta-model (Figure 1.7). It defines the abstract as well as graphical syntax. Iconic representations of entities in the DCharts formalism are not shown. Some of the DCharts constructs, such as importation and transition priorities, are not explicitly modeled for simplicity. (They are marked as UML-style comments in the graphical representation of DCharts models.)

Figure 2.2 shows the AToM^3 environment with the DCharts meta-model loaded in it. The buttons on the left panel give access to all the entities to be used in a DCharts model. The buttons on the top allow the user to connect entities and edit their properties.

2.2 Overview of Abstract Syntax and Semantics

This section defines the abstract syntax of DCharts 1.0. Part of this definition is subject to change in later versions. This possibility of change is discussed wherever appropriate.

2.2.1 Overview

A model M in DCharts 1.0 is defined by a tuple

$$< S, T, C, V, \Delta, P, L >$$

where:

- $S = \{s_1, s_2, ..., s_m\}$ is a set of finite and enumerable states. s_i ($i \in [1, ..., m]$) is a tuple

$$< SN, DS, CS, HS, TP, EN, EX >$$

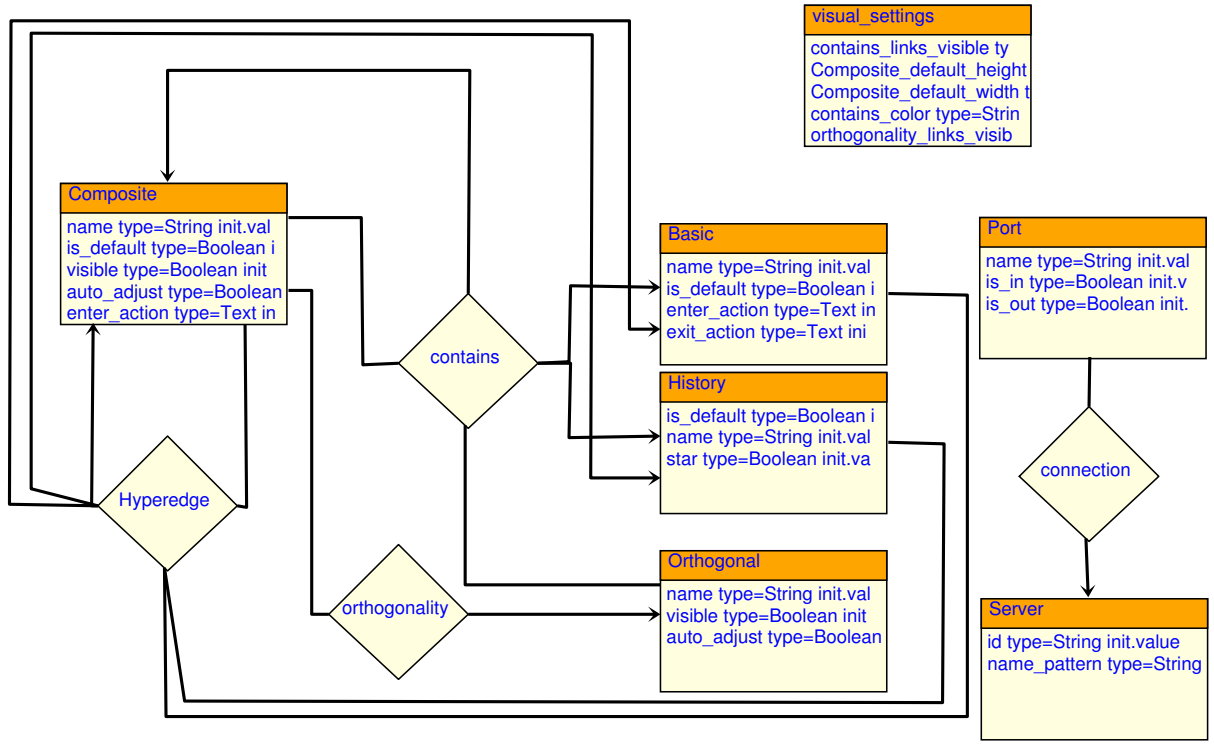

Figure 2.1: The DCharts meta-model in an Entity-Relationship diagram

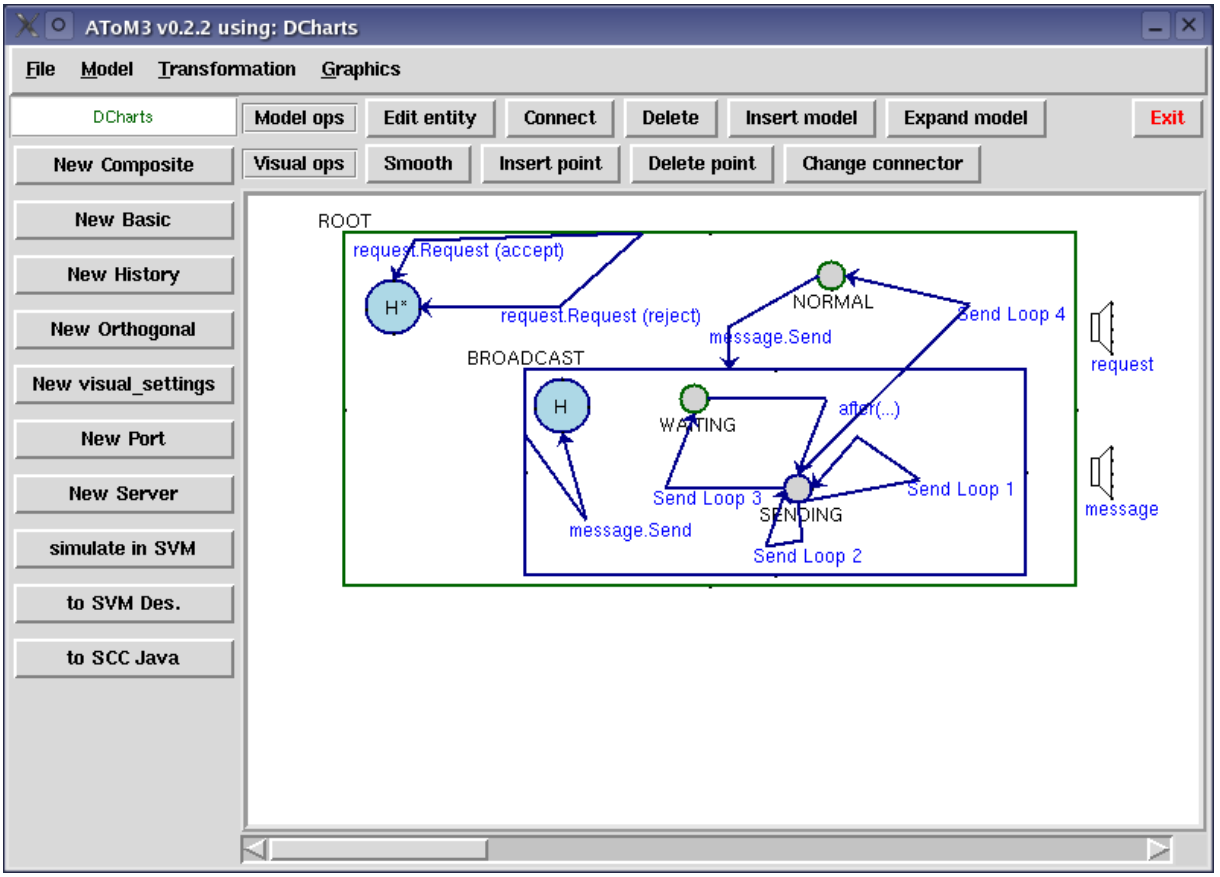

Figure 2.2: The AToM^3 development environment for DCharts

where:

- SN is a string that represents a globally unique identifier (GUID) of the state. Every state in a given model has a unique GUID. GUIDs of states in different models may be the same.
- DS is a boolean value that specifies whether state s_i is a default state of its parent state. Among the children of a parent state there must be exactly one default state, unless all those children are orthogonal components (discussed later).
- CS is a boolean value that specifies whether state s_i is an orthogonal component. If a state is an orthogonal component, all its *siblings* (other children of the same parent) must also be orthogonal components, and all those states are default states in their nature ($DS = true$).
- HS is an enumerated value that specifies whether state s_i has a history in it, and, if it has a history, whether the history is deep history. Its possible values are defined with an enumerated type (explained later):
$$HS = \{None, Normal, Deep\}$$
- TP is an enumerated value that specifies the transition priority within the scope of state s_i. The *scope* of a state includes the state itself, all its children states, and all the transitions *from* that state and its children states. Possible values of TP are defined below (explained later):
$$TP = \{Keep, ITF, OTF, RTO\}$$
- EN is a list of sequentially executed enter actions. These actions are executed when the state is entered (whether from other states or from this state itself). In DCharts 1.0, there is no strict definition of actions. This part is subject to change in later versions, where a more rigorous definition of action code will be given. However, actions must conform to several rules. Those rules are discussed in section 2.2.8.
- EX is a list of sequentially executed exit actions. These actions are executed when the state is exited (whether the destination is another state or this state itself). This part is subject to change in later versions, where a more rigorous definition of action code will be given. Some rules of actions are further discussed in section 2.2.8.

- $T = \{t_1, t_2, ..., t_n\}$ is a set of transitions. t_i ($i \in [1, ..., n]$) is a tuple
$$< SRC, DES, E, \gamma, G, \lambda, HS_T, Prio >$$

where:

- $SRC \in S$ is the source state.
- $DES \in S$ is the destination state.
- E is a string that represents the event name. This event triggers the transition. The event name should not contain a "." (which is used in inter-model communication via ports), unless the transition handles an incoming message. In that case, the event name is the input port name followed by a dot, and then followed by the message name.
- $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_k\}$ is a set of variables that represent the formal parameters. Each parameter is a variable. In DCharts 1.0, there is no strict definition of variables. Section 2.2.4 offers a loose definition of variables.
- G is a boolean expression that specifies the guard. In DCharts 1.0, there is no strict definition of guards. One of the few requirements is that they can be evaluated to a boolean result at the time when the event of the transition is received. Guards are further discussed in section 2.2.8.
- λ is a list of sequentially executed actions. In DCharts 1.0, there is no strict definition of actions. This part is subject to change in later versions, where a more rigorous definition on action code will be given. Several rules of actions are discussed in section 2.2.8.

- HS_T is a boolean variable that specifies whether the transition goes to the history of the destination state (if it has a normal history or deep history) or the default substate of it (if it has substates).
- *Prio* is an integer number (may be positive or negative) that specifies the priority of the transition. In case a conflict occurs that cannot be solved by transition ordering (discussed later), the transition with the smallest *Prio* number has the highest priority.

- $C : S \rightarrow 2^S$ is a function that defines the parent-children relationship of all the states. It maps any state to the set of its children states. All the states in a model and their parent-children relations form a tree (with states as nodes and relations as edges).
- $V = \{v_1, v_2, ..., v_s\}$ is the set of variables. In DCharts 1.0, there is no strict definition of variables. The only requirement is that every variable has a GUID and provides a certain amount of storage. This part is implementation-dependent. It is subject to change in later versions, where a more rigorous definition of variables and the operations on them will be given.
- $\Delta : S \rightarrow M$ is the mapping of importations. If Δ is defined for a state s_i, the result of the function gives the definition of a model which is imported into s_i. *Importing* a model into a state means, in theory, including all its states and transitions in that state. The states defined in the imported model become substates of the *importation state*. The transitions between the states of the imported model are preserved. An implementation of DCharts must provide a means to modify the GUIDs in the submodels, so that they never conflict with the GUIDs of the importing model. (Note that the imported model itself may import models, and a model may import multiple models.) It is allowed for a model to import itself. Such a recursive specification allows for the dynamic creation of arbitrary-sized models (discussed later).
- $P = \{p_1, p_2, ..., p_t\}$ is the set of ports. In DCharts 1.0, a port p_i ($i \in [1, ..., t]$) is a tuple $< PN, PT >$ (later versions of DCharts may add more information to a port to further specify it), where:

  - *PN* is the GUID of the port. This GUID has a different name space from the GUIDs of states. As a result, even if the *PN* of a port is equal to the *SN* of a state, no conflict occurs. There is no restriction on a port name, except that it must not contain a "." or a space.
  - *PT* is the type of the port. In DCharts 1.0, the following types are defined (later versions of DCharts may add more information):

    $$PT = \{InOut, In, Out\}$$

- $L = \{l_1, l_2, ..., l_u\}$ a set of mappings from ports in one model to the ports of other models. l_i, with $i \in [1, ..., u]$, is a tuple $< PN_1, < M, PN_2 >>$, where:

  - PN_1 is the GUID of a port in this model (the model that contains the definition of this l_i).
  - M is another model, which the currently specified model connects to.
  - PN_2 is the GUID of a port in model M.

  Note that if P_1 of model M_1 is connected to P_2 of model M_2 with a single connection, either M_1 or M_2 specifies this connection, but not both. The model in with the connection is specified looks up the other model when the simulation or execution of it starts.

  If in a simulation or execution, more than one model (or component) $M_1, M_2, ..., M_v$ has exactly the same definition ($M_1 = M_2 = ... = M_v$), a link $l =< PN_1, < M_i, PN_2 >>$ ($i \in [1, 2, ..., v]$) in L implies that PN_1 is connected to PN_2 of all those models with the same definition.

2.2.2 State Set S

State set $S = \{s_1, s_2, ..., s_m\}$ defines all the states in a model, regardless of their parent-children relationship.

The choice of *SN*, the GUID of a state, is a decision of the model designer. According to the definition of sets, there should not be two identical elements in a single set. Two states in the same state set differ from each other at least in their GUIDs.

DS defines whether the state is a default state of its parent. If the state is at the top level (i.e., it has no parent) *DS* defines whether it is a default state of the model. If *CS* is *true* for a state, it is an orthogonal component of its parent, and as a result, all its siblings are orthogonal components. Due to the nature of orthogonal components (also known as "and states"), all orthogonal components must be active simultaneously, and are in some sense default states. Hence, $CS = true$ always implies $DS = true$.

When started, the model is always in its top-level default state(s), and the default substate(s) of the top-level default state(s). At any given time during a simulation or execution, the model is in its *current leaf state* (a *leaf state* is defined as a state at the lowest level, which does not contain any substate), or *current state* for short. A model is *in state s* if and only if its current state is *s* or a substate of *s*. Hence, a model in state s' is also in s'' provided that:

$$\exists k \in N^+, s_1, s_2, ..., s_k \in S \cdot s' \in C(s_1) \wedge s_1 \in C(s_2) \wedge ... \wedge s_{k-1} \in C(s_k) \wedge s_k \in C(s'')$$

Here, $C(s)$ is the children set of state *s*. $< s'', s_k, s_{k-1}, ..., s_1, s' >$ is called the *path from superstate s'' to substate s'*. Such a relation between s' and s'' is formally written as $s' \in Substate(s'')$.

Each orthogonal component has a default state defined in it. At any time in a simulation or execution, all orthogonal components in an active state have a current state. All the transition from those current states (or superstates of the current states) with their guards evaluated to *true* are enabled and can be triggered by an event. However, at any time there is at most one triggered transition, and the execution of a transition must be finished without any interleaving operation. Usually, a transition is implemented as a critical section. In this interpretation, orthogonal components are not concurrent threads. The Cartesian product of all the orthogonal components gives a unique current state of a model. This model can be transformed to an ordinary FSA, which has no orthogonal components.

HS specifies the history of a state. History is regarded as a property of a state. If a state has a normal history or deep history, a transition with that state as destination can be either to its default substate or to its history. The transition specifies this choice by means of its HS_T property. As a special case, if a transition with $HS_T = true$ goes to a state without history, HS_T is automatically ignored or changed to $false$.

The difference between normal history and deep history, as they were first introduced in statecharts [4], is that normal history only records the last visited child of a state, while deep history records all the last visited substates of a state so that when the model goes back to this history, those substates are restored.

TP defines the priority of transitions within the scope of a state. The TP definition of a substate always overrides the TP definition of its superstates. Transition priorities are discussed in section 2.2.5.

EN and EX actions are implementation-dependent. However, they are restricted to sequences of single actions. Hence, loops and if-else conditions are not allowed. These structures must be explicitly modeled.

2.2.3 Transitions T

T is a set that contains all the transitions in a model. A transition, when enabled, changes the model from the source state to the destination state. A transition is *enabled* if and only if the following conditions are all satisfied:

1. The model is in the source state *SRC*.
2. Event *E* is received and is at the head of the *global event list* (a logical list that contains all the events to be processed by a model in the sequence of their arrival).
3. All the formal parameters defined in γ have their values of the correct types (if the specific implementation supports types).
4. Guard *G* is valid and evaluated to *true*.

Provided that the guard (though it is implementation-dependent) is only a boolean expression without any side effect (i.e., potential change to the model state), and the execution of a transition is in a critical section so that no other actions affect the decision, the order of the above conditions is not important. For example, if a specific implementation uses exactly the same order to decide enabled transitions, it may ignore the evaluation of guards if any one of the first three conditions is not satisfied. (This method is known as short-circuit.)

There may be more than one transition (in the same component) enabled by a single event. In that case, the transition with the highest priority is *fired*, which means its λ is executed, and the state changes to DES.

γ is a set of variables that act as formal parameters received with the event E. If a transition requires parameters, the event must provide *at least* the same number of parameters with the same types (if types are supported). It may even provide more parameters (after those required), which are simply omitted.

λ is a list of sequentially executed output actions. Control structures such as loops and if-else conditions are not allowed. Those structures must be explicitly modeled (see section 5.5). λ may contain actions that further broadcast events. If such actions are included, the newly broadcast events are appended to the end of the global event list.

HS_T is used in combination with HS of a state. However, HS_T is ignored if the HS property of the destination state DES is *None*.

Prio is an arbitrary integer that represents transition priority. The smaller this number is, the higher priority the transition has. However, this number is used only for conflicts that cannot be resolved by the scheme of transition ordering (see section 2.2.5).

2.2.4 Variables

Variables are involved in several parts of a model specification. Though they are implementation-dependent, the minimum requirement for a variable is described here:

- A variable has a GUID. This GUID uniquely identifies a variable within a model. Variable GUIDs, state names (*SN*) and port names (*PN*) have different name spaces and hence GUIDs need not be unique among them.
- A variable stores a certain amount of data. Those data may or may not have types, depending on the specific implementation.
- A variable can only be modified by the λ of a transition, or EN or EX of a state. Other parts of the model cannot modify variables.
- The data stored in the variables can be retrieved in G and λ of transitions, and EN and EX of states. In particular, G of transitions may use variables to determine their $true/false$ value. Other parts of the model cannot retrieve the data in variables.

Provided these rules are satisfied, an implementation of DCharts 1.0 may choose any scheme to implement variables.

2.2.5 Transition Priorities

Transition Priorities are an extension of concepts found in STATEMATE statecharts, UML statecharts and DEVS (the select function). They are used to solve run-time *conflicts* between multiple transitions enabled by the same event at a given time. This case would make the model non-deterministic. [26]

Two possible kinds of transition conflicts are described in [5].

1. At least two transitions are enabled by the same event, the source state of one of the transitions is a substate (or superstate) of the source state of the other transition(s).
2. At least two transitions are enabled by the same event, they have the same source state.

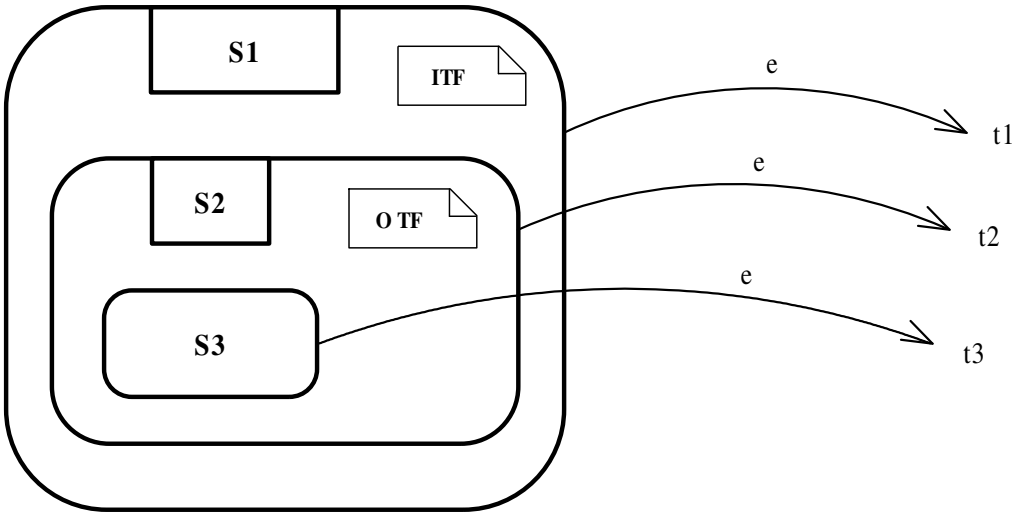

Figure 2.3: An example of transition priorities

(Note that if those transitions have source states in different orthogonal components, no conflict occurs and all those transitions are triggered by the same event. In that case, the order of the firing of those transitions is random or implementation-dependent.)

Solutions to the first kind of conflicts are found in both the STATEMATE semantics [5] and the UML [27]. Unfortunately, the solutions from the two sources are opposite: in UML, if the source state of a transition is a substate of the source state of the others, it gets higher priority; however, in the STATEMATE semantics, it gets lower priority.

In DCharts, it is possible to customize the priority of transitions by setting the TP attribute of states (1st-round decision) and the *Prio* attribute of transitions (2nd-round decision). The semantics of the different values of TP is formalized below (function $Priority(t)$ is the total priority number of transition t; t_1 and t_2 are transitions; SRC_1 is the *SRC* of t_1; SRC_2 is the *SRC* of t_2):

- $TP_{SRC_1} == ITF \wedge SRC_2 \in Substate(SRC_1) \Rightarrow Priority(t_1) > Priority(t_2)$
  If SRC_1 is set to be inner-transition-first (ITF), and SRC_2 is a substate of SRC_1, then the total priority of t_1 is lower than that of t_2, or the total priority number of t_1 is larger than that of t_2.
- $TP_{SRC_1} == OTF \wedge SRC_2 \in Substate(SRC_1) \Rightarrow Priority(t_1) < Priority(t_2)$
  If SRC_1 is set to be outer-transition-first (OTF), and SRC_2 is a substate of SRC_1, then the total priority of t_1 is higher than that of t_2, or the total priority number of t_1 is smaller than that of t_2.
- If $TP_{SRC_1} == Keep$, SRC_1 preserves the TP setting of its parent. This means, if its parent is inner-transition-first, SRC_1 is inner-transition-first, too. And vice versa.
- If $TP_{SRC_1} == RTO$, SRC_1 reverses the TP setting of its parent. This means, if its parent is ITF, SRC_1 is OTF. And vice versa. In the case where $s_2 \in C(s_1) \wedge TP_{s_1} == RTO \wedge TP_{s_2} == KEEP$, s_2 preserves the transition order of s_1 rather than the reverse-transition-order (RTO) property itself. Similarly, if $TP_{s_1} == RTO \wedge TP_{s_2} == RTO$, s_2 reverts the actual transition order of s_1 rather than the reverse-transition-order (RTO) property itself.

Suppose there are three transitions `t1`, `t2` and `t3` as illustrated in Figure 2.3. When event `e` occurs, they are all enabled, so there is a conflict of the first kind. To understand the priority of these transitions, one must first consider the outermost state and step inward from there. Because `S1` is specified to be ITF, the priority of `t1` must be lower than both `t2` and `t3`. Since `S2` is OTF, `t2` has a higher priority than `t3`. So the ordering by priority is `t2`, `t3`, `t1`. Detailed explanation of the graphical representation of DCharts models is in section

4.1.

The above scheme cannot solve the second type of conflicts. In case of such a conflict, the transition with the smallest *Prio* number is fired. It is the designer's responsibility to ensure that this situation does not occur, or even if it occurs, there is a unique decision among the conflicting transitions according to their *Prio* numbers. If the choice is not unique (more than one transition has the smallest *Prio* number), the transition that is fired is random or implementation-dependent.

2.2.6 Importation

Importation allows reusing a model in another by placing all its states and transitions in a state of the importing model. That state of the importing model is called *importation state*. An importation state is not allowed to have substates prior to importation (and is thus a leaf state). After importation is done, that state becomes a non-importation state.

After importation, some of the importing model's elements must be changed to reflect the new model with more states and transitions. For example, the GUIDs of the states in the imported model are modified so that the new GUIDs still uniquely identify those states after importation. An implementation of DCharts could choose a prefix and add it to the head of all the original GUIDs in the imported model. The *C* (children) function must be changed, since all the states of the imported model become substates of the importation state (and the top-level states become its children). The transitions in the imported model are added to the T set of the importing model, with the GUIDs of their *SRC* and *DES* properties accordingly changed. Other GUIDs in the imported model must be modified in a similar way.

Importation is a dynamic operation. It is done at run-time when the imported model is needed. This allows recursive importation, where a model explicitly or implicitly imports itself. A theoretically infinite state hierarchy can be created in this way.

The imported model is forbidden to transition to the states of the importing model. This breaks the modularity of the imported model. To interact with the importing model, the imported model should send (i.e., broadcast) events that trigger transitions in the importing model. After the imported model is merged with the importing model, there is no distinction between transitions in the importing model and the imported model any more. An event may be handled by any transition in the combined transition set.

The concept of leaf states becomes relative when importation is considered. Leaf state *s* may import another model. Since the importation is done dynamically, it is possible that *s* is a leaf state before a transition is fired, while it becomes non-leaf after that because of an importation. It is assumed that notation $Leaf(s,t)$ or simply $Leaf(s)$ returns whether or not *s* is leaf at a certain point in time (t).

At any time in a simulation or execution, an importation state is always a leaf state. (Be reminded that after a submodel is imported into it, it is no longer called importation state.)

2.2.7 Ports and Connections

Ports and connections provide a means for a model to communicate with other concurrently and independently running models. This is different from importation, where a model is imported into an importation state of another model, and the combined model runs sequentially as a whole.

Connections are the communication channels between those concurrent models. After they are established, messages can be sent and received via those channels. Except when two connected models communicate, they are independent, and they have no other means to affect the behavior of each other.

A message is a tuple $< MN, \gamma >$ where *MN* is the message name. The names of different messages may be the same. Actually, there is no way to guarantee uniqueness, since every model runs independently and concurrently. There is no restriction on *MN*, except that it should not contain a dot ".". $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_k\}$ is a set of parameters. Each parameter is a variable.

To establish a connection, a *server model* with at least one port must be started first. The L link set of the server may be empty, since it usually does not connect to other models at start-up time. A *client* must also have at least one port. When it starts running, the simulation/execution environment connects it to the

server(s) according to the L set defined in it. That is, for each link $l_i =< PN_1, < M, PN_2 >>$ defined in L, connect port PN_1 of the client model with port PN_2 of the server model(s) M. All the connections are established at start-up time. If any of the connections cannot be established, the client model cannot be simulated or executed. Its simulator or executor should immediately terminate, without even placing the client model in its default state(s).

If any of the following situations occurs, the establishment of connection $l_i =< PN_1, < M, PN_2 >>$ is considered a failure:

- The model with l_i in its L set has no port called PN_1.
- Model M cannot be found in a certain scope that the simulator or the executor is interested in.
- Model M has no port called PN_2.
- Both PN_1 and PN_2 are in-ports.
- Both PN_1 and PN_2 are out-ports.
- Model M does not respond to the connection request within a certain timeout (an implementation-dependent parameter).

Once connections between two models are established, they are never disconnected.

A port of a client may connect to multiple ports of one or more servers. Reversely, a port of a server may be connected by multiple ports of one or more clients.

Messages can be sent in any part of a model where action code can be written, such as the output λ of a transition, and EN and EX of a state. To send a message, a model simply broadcasts an event whose name starts with the port name and a following dot ".". On the one hand, since neither the port name nor the message name can have a dot in it, the only dot in this representation separates the two parts. On the other hand, no transition handles an event sent internally with a name that contains a dot, the simulator or executor knows that it is an out-going message instead of a normal event.

Before the message is sent via a connection, the port name and the dot are removed. When the simulator or executor of the receiver receives this message, it first adds the name of its input port to the message name (again separated with a dot), and then broadcasts the event internally. The parameters of the message are regarded as the parameters of the event.

2.2.8 Actions and Guards

There is no strict definition of actions in the semantics of DCharts 1.0. As a loose definition, an *action* is a statement in an action language, which modifies the variables of the model, outputs events, or interacts with other parts of the system that are not modeled with DCharts. An implementation of DCharts may support an action language that is specific to it. For example, because SVM is implemented in Python and Python is an interpreted language, Python is chosen as the action language for SVM. Other DCharts implementations may use different action languages.

An action language must satisfy the following rules:

- An action must not modify the state hierarchy of the model. Nor can it reflect upon the current state of the model.
- Actions are executed sequentially. They must not contain any control flow structure, such as branches and loops.
- An implementation of DCharts usually provides primitive actions to model designers. With those primitive actions, the models are able to interact with the simulation or execution environment, or broadcast events. Examples of models' interaction with the simulation or execution environment include but are not limited to snapshot requests. (The complete state of the model is taken by a snapshot, which may be used later to roll back the model.) Snapshot is discussed later.

- A primitive action must be provided to access the elapsed time since the simulation or execution was started. The increment of this time must be synchronized with the wall-clock. I.e., the time obtained by this primitive action is increased by 1 after 1 second is elapsed in reality. (The accuracy depends on the operating system.)

The case of constraint language is similar. A constraint language is used to express guards of transitions. A *guard* is a side-effect-free statement that can be evaluated to either $true$ or $false$ at the time when the simulator or executor decides whether a transition is enabled. (In particular, for a timed transition with a guard, the guard is evaluated after the scheduled amount of time instead of at the time when the transition is scheduled.)

A constraint language must satisfy the following rules:

- A guard written in the constraint language must be side-effect-free. The language must not allow any change on the states or variables. Nor should it allow guards to control other parts of the system, which are not modeled.
- A guard is allowed to access the current state of the model. This makes it possible for an orthogonal component to dynamically decide whether a transition should be fired based on the current states of other orthogonal components.
- A guard should never affect the simulation or execution process (for example, by causing the model to sleep for a certain amount of time or by snapshotting or rolling back the model).
- Guards must be deterministic. Provided that all the states or variables (including those in the parts that are not modeled) that a guard depends on are not changed, the guard must always yield the same $true$ or $false$ result.

The above are the minimal requirements for DCharts actions and guards. Later versions of DCharts may explicitly define an action language and a constraint language, or specify more requirements for them.

2.3 Algorithms

This section discusses some important algorithms for the implementation of DCharts. They define part of the operational semantics.

2.3.1 Firing a Transition

Every time an event is received, the transition that becomes enabled with the highest total priority is fired. To fire a transition T, the following algorithm is used:

1. Output actions in λ of T are executed.
2. Here are some definitions:

   Common superstates are the set of common superstates of two or more states (s_1,s_2,...,s_n). They are defined as:

   $$Common(s_1, s_2, ..., s_n) = \{s | s_1 \in Substate(s) \wedge s_2 \in Substate(s) \wedge ... \wedge s_n \in Substate(s)\}$$

   The *Closest Common Superstate CCS* (David Harel has called it LCA, which is short for *Lowest Common Ancestor*) of $s_1, s_2, ..., s_n$ is defined as:

   $$CCS(s_1, s_2, ..., s_n) = s \cdot (s \in Common(s_1, s_2, ..., s_n) \wedge \neg \exists s' \cdot (s' \in Substate(s) \wedge s' \in Common(s_1, s_2, ..., s_n)))$$

   In this step, $CCS(SRC, DES)$ is decided. The model exits *SRC* and all the states in the path from $CCS(SRC, DES)$ (not including $CCS(SRC, DES)$ itself). The exit actions of a state at a lower level are

executed before the exit actions of a state at a higher level. If $CCS(SRC, DES)$ cannot be found (i.e., *SRC* and *DES* have no common superstate, or $Common(SRC, DES) = \emptyset$), all the current states are exited, and all their exit actions are executed in the correct order. (If we imagine there is a *top state* $\top$ that encloses all the states in a model, the *CCS* in this case is $\top$.)

Exit actions of orthogonal components belonging to the same parent are executed in an implementation-dependent order. This is because orthogonal components are logically simultaneous, and the designers should never make the behavior of their models dependent on the execution sequence of the actions among different orthogonal components.

3. All the states in the path from $CCS(SRC, DES)$ (not inclusive) to *DES* are entered. The enter actions of a state at a higher level are executed before the enter actions of a state at a lower level. The enter actions (if any) of *DES* are executed last.

   Enter actions of orthogonal components belonging to the same parent are executed in an implementation-dependent order.

If $SRC = DES$, in which case the transition is a *self-loop*, the enter/exit actions of the state are executed, because $CCS(SRC, DES)$ is equal to the parent state of *SRC* (or *DES*).

2.3.2 Alternate Algorithm for Firing a Transition

The algorithm in the previous section for firing a transition conforms to David Harel's statecharts, whose semantics is described in [5]. Alternatively, an implementation of DCharts may also support another algorithm for firing a transition. This alternate algorithm is not compatible with David Harel's statecharts. However, it sometimes better expresses the behavior of a system to be modeled.

In this algorithm, $CCS_{alt}(SRC, DES)$ is defined in a different way:

$$CCS_{alt}(SRC, DES) = \begin{cases} SRC, & DES \in Substate(SRC) \\ DES, & SRC \in Substate(DES) \\ SRC, & SRC = DES \\ CCS(SRC, DES), & otherwise \end{cases}$$

The steps of the firing of a transition is described below:

1. $CCS_{alt}(SRC, DES)$ is decided. The model exits *SRC* and all the states in the path from $CCS_{alt}(SRC, DES)$. The exit actions of a state at a lower level are executed before the exit actions of a state at a higher level. If $CCS_{alt}(SRC, DES)$ cannot be found, all the current states are exited, and all their exit actions are executed in the correct order.
2. Output actions in λ of T are executed.
3. All the states in the path from $CCS_{alt}(SRC, DES)$ to *DES* are entered. The enter actions of a state at a higher level are executed before the enter actions of a state at a lower level. The enter actions (if any) of *DES* are executed last.

Compared to the algorithm described in the last section, this algorithm reverses the first operation and the second operation. As a result, when the λ of a transition is executed, the model is not in *SRC* but $CCS_{alt}(SRC, DES)$. This better reflects the fact that λ is executed *while* transition T is fired, not *before* or *after* that.

Suppose *DES* is a history state, the *history leaf substates* $History(DES) = \{h_1, h_2, ..., h_n\}$ are defined as the set of *DES*' substates recorded in its history, where $Leaf(h_1)$, $Leaf(h_2)$, ..., $Leaf(h_n)$ are all $true$. The *default leaf substates* $Default(DES) = \{d_1, d_2, ..., d_n\}$ are defined as the set of *DES*' default substates, where $Leaf(h_1)$, $Leaf(h_2)$, ..., $Leaf(h_n)$ are all $true$. It is possible that $History(DES)$ and $Default(DES)$ contain more than one elements, considering that the *DES* may have orthogonal components as its substates.

In the special case where $SRC = DES$, or though $SRC \neq DES$, SRC is in the path from DES to any state in $History(DES)$ or $Default(DES)$, the enter/exit actions of DES (and its superstates) are not executed. This is because a self-loop to DES is not considered as a state change. This is the reason for using CCS_{alt} instead of CCS.

An implementation of DCharts *must* implement the first algorithm described in the previous section. The algorithm discussed here is highly optional.

2.3.3 Importation

When model M' is imported into state s of model M, a part of its definition is merged with M, while the rest is ignored. The algorithm below describes this merging:

1. As an stimulus of this importation, a transition t must be fired, or the submodel is placed in a default state of the model so that it is required at the very beginning of a simulation or execution. t has the following properties:

   - Its DES is an importation state (a state s that has a value at $\Delta(s)$), or an importation state s appears in the path from $CCS(SRC, DES)$ (or $CCS_{alt}(SRC, DES)$ depending on the algorithm used) to DES, or one of the default substates of DES is an importation state s.
   - $\Delta(s) == M'$. M' has not been imported into s yet. (If M' has already been imported into s because of the firing of a previous transition, it is never removed, and s is no longer an importation state.)

   If such a transition is detected, the simulation/execution environment must first prepare the substates of s by importation (since it is at that time a leaf state) before actually firing the transition.

   If an importation state is a default state of a model, the simulation/execution environment must import the appropriate submodel at the beginning. If the imported model requires more submodels, they are also imported at that time. It is the designer's responsibility to make sure that this repeated importation process ends in finite and acceptable time.
2. To import model M' into importation state s of M, the following merging operations are performed:

   3. The GUIDs of the states in M' are modified to make them globally unique within the states name space. This may be implemented by adding a prefix. The SRC and DES of each transition are modified accordingly. The parent-children relationship function C is modified accordingly.
   4. The state hierarchy of M' is merged with M. All the states in M' become substates of s. The C function of M' is combined with the C function of M.
   5. T of M' is merged with T of M. If it is enforced that the SRC of any transition in M cannot be a state in submodel M', the two transition sets do not have overlapping elements. (However, the DES of some transitions may be states in M'.) Note that the DES of a transition in M' cannot refer to a state in M, since the simulation or execution environment modifies it to be a unique GUID at the time of importation.
   6. V of M' is merged with V of M. If a variable in M' has the same name as a variable in M, it is considered the same variable. If they have different types in a type system, a run-time exception is raised.
   7. Δ is merged after the GUIDs of states are changed.

   And the following properties of M' are ignored:

   - The FS properties of all its states are set to $false$. This is because the behavior of M' should not affect the original behavior of M. If those final states are kept after importation, they would stop the simulation or execution of M unexpectedly.
   - P and L of M' are ignored, since the submodel M' cannot open a port or establish a connection dynamically.

1. s becomes a non-importation state, because M' has already been imported into it. s is no longer a leaf state, either. When the importation is finished, there is no knowledge of model M' any more.
2. Repeat steps 2.3.3 to 2.3.3 until no importation state s is found in the path from $CCS(SRC, DES)$ to DES or DES itself.

After this merging of one or more submodels, transition t can be fired according to any of the algorithms described in previous sections.

2.4 Closure under Importation

Importation is a kind of tight coupling between models.

Strictly speaking, DCharts are not closed under importation. I.e., it may not be possible to find such a model M' so that it has exactly the same behavior as the original model M but it has no importation state. A counter-example can be easily found. Suppose $\exists s \in S \cdot \Delta_M(s) = M$, which means model M imports itself in state s. This creates an infinite structure. It is impossible to find a non-recursive model M' with the same behavior.

If recursive importation is not considered, closure under importation can be proved with the algorithm described in section 4.2.9.

Theorem 1 *Non-recursive DCharts models are closed under importation. I.e., models with importation states can be replaced by models without importation states, which have exactly the same behavior.*

Proof The *expanded model* M' of non-recursive DCharts model M is found by the following algorithm:

function expand(M)
  M'=M
  for s in S of M'
    if $\Delta_{M'}$ *is defined at s then*
      M'_s=*expand*($\Delta_{M'}(s)$)
      import M'_s *into* M' *according to the algorithm in section 4.2.9*
  return M'

Since M is non-recursive, this algorithm always terminates in a finite number of steps. The model M' returned does not contain any importation state.

Obviously, M' is still a DCharts model. According to the semantics of dynamic importation (section 4.2.9), it has exactly the same behavior as M. □

2.5 Asynchronous Communication and Synchronous Communication

Asynchronous communication and synchronous communication are the two different types of inter-model communication. They define the semantics of sending and receiving messages via ports, after necessary connections are established.

Asynchronous sending means sending without waiting for response. DCharts require that an action be provided to send messages asynchronously to a specified port. A message, as discussed in section 2.2.7, is a tuple $< MN, \gamma >$ where MN is its name, and γ is a set of parameters. The message is buffered and the action returns immediately.

In many cases, the sender is not interested in when or whether a message is received by the receiver. However, mechanisms (possibly in the constraint language) should be provided to check this result. There are two possible mechanisms:

1. Checking function. It is a boolean function that returns whether a message has been received. A model may use this function in the actions or guards to check the status of a message sent previously.
2. Callback. A certain event is sent by the simulation/execution environment of the sender when the confirmation from the message receiver is received by the sender. The event name is specified by

the sender model at the time when it sends out the message. The parameters of a callback event is implementation-dependent.

Synchronous sending means sending a message and waiting until the receiver confirms the receipt of the message with an acknowledgment. Some communication protocols allow to test whether a message is correctly received, or to wait until it is received. A DCharts implementation on such a protocol may not require acknowledgments.

Since a model is simulated or executed sequentially, synchronous sending blocks the whole model until the simulation/execution environment knows the message is received. During the blocking period, all the incoming events are queued by the simulation/execution environment. Some scheduled transitions may be delayed because of this blocking. As a result, if a model uses synchronous sending, it is the designer's responsibility to make sure that the time constraints of scheduled transitions (if any) are satisfied. (Event scheduling and timing of DCharts models are discussed in chapter 3.)

The following algorithms use the facilities of asynchronous sending to simulate synchronous sending:

1. If a checking function is provided, the model asynchronously sends a message, and goes to an isolated state. A transition is repeatedly scheduled after a certain period (*timeout*). When this transition is triggered, it checks the status of the message in its guard. If the checking function returns *true*, the model goes back to the previous state; otherwise, the same transition is scheduled after the same timeout period.
2. If callback is supported, the model asynchronously sends a message, and goes to an isolated state. The callback event triggers a transition from that state back to the previous state.

In both cases, the isolated state must accept all other events and sequentially record them in a variable. At the time when the model goes back to the original state, those recorded events are re-broadcast in the same order. This explicitly models part of the global event list of the simulator/executor.

3 Timing

As part of the DCharts semantics, the timing of DCharts models is defined in this chapter.

The DCharts formalism requires that its implementation (whether it is a simulator or an executor) strives to provide the real-time timing scheme for every model.

3.1 The Real-time Concept

The term *real-time* is defined by FOLDOC [28] as following:

1. Describes an application which requires a program to respond to stimuli within some small upper limit of response time (typically milli- or microseconds). Process control at a chemical plant is the classic example. Such applications often require special operating systems (because everything else must take a back seat to response time) and speed-tuned hardware.
2. In jargon, refers to doing something while people are watching or waiting. “I asked her how to find the calling procedure’s program counter on the stack and she came up with an algorithm in real time.”

(Used to describe a system that must guarantee a response to an external event within a given time. [28])

Definition 1 puts the requirement of time more on the underlying operating system than the specific DCharts implementation. This is because the small upper limit of response time can only be guaranteed if the operating system supports it. For many common-purpose systems that model designers would most probably use, this guarantee is hard to achieve. For example, Linux only provides a very limited support for real-time computation; Windows and many other multi-tasking operating systems perform even more unsatisfactory within the real-time domain. To allow DCharts to be implemented on most systems and platforms, the real-time requirement cannot be formalized as strict as that in definition 1.

Definition 2 makes the real-time concept more general: the DCharts implementations provide real-time support for models within the extent of their capability. The model users watch the simulation/execution of the models and wait for them to respond.

The real-time concept is defined by Webopedia [29] in a similar way:

1. Occurring immediately. The term is used to describe a number of different computer features. For example, real-time operating systems are systems that respond to input immediately. They are used for such tasks as navigation, in which the computer must react to a steady flow of new information without interruption. Most general-purpose operating systems are not real-time because they can take a few seconds, or even minutes, to react.
2. Real time can also refer to events simulated by a computer at the same speed that they would occur in real life. In graphics animation, for example, a real-time program would display objects moving across the screen at the same speed that they would actually move.

Definition 2 is useful for the understanding of the real-time required by DCharts. Similarly, the requirement of “the same speed” is not strict. DCharts implementations should provide this support as much as possible, given the restrictions of the operating systems that they are built for.

MSN Encarta [30] also defines the real-time concept, which is stricter than the real-time concept in DCharts:

1. Computing immediacy of data processing: the time in which certain computer systems process and update data as soon as it is received from some external source, e.g. an air-traffic control or antilock brake system. The time available to receive the data, process it, and respond to the external process is dictated by the time constraints imposed by the process.
2. Actual time of occurrence: the actual time during which something happens.

In most cases, model designers may assume that 1 second elapsed in the model simulation or execution is approximately equal to 1 second in reality. Designers with time-critical requirements should turn to the documentation of specific DCharts implementations or operating systems to know whether they suit the need.

3.2 Virtual-time Simulation

Though real time is desirable for many practical applications, virtual time is still necessary in other cases. There are two kinds of virtual time simulation:

- In *scaled real-time simulation*, the time is proportional to the real time. The *scale factor* is a floating-point number, and may be 1.0 in some cases.

  Scaled real-time simulation can be easily simulated by a real-time simulation. This is done by multiplying all the time variables in a model with the scale factor. Macro redefinition discussed in later chapters allows for flexible change of the scale factor.
- In *as-fast-as-possible simulation*, a variable is used to keep track of time. It is increased to the smallest scheduled time (see section 3.3) when the model becomes *idle* (the global event list becomes empty). This time variable has no relation with the time in reality. It is consistent only in one single simulation. Within one simulation there is exactly one such variable, and the model retrieves *current time* from this variable, and schedules events with it.

  As-fast-as-possible simulation can be simulated by a real-time simulation. For a stand-alone model (a model that does not communicate with other models via ports), a *clock component* is designed and discussed later with examples (section 9.2). By importing the clock component as a top-level orthogonal component in the model, as-fast-as-possible simulation is enabled. The clock component maintains the time variable. To retrieve time, the model sends an event, which triggers a transition in the clock. That transition outputs another event with the value of the time variable as a parameter. The latter event tells other parts of the model the current time.

  To schedule an event, the model sends an event with the scheduled time as a parameter. The event triggers another transition in the clock component, which adds the scheduled time to its *schedule list* (a variable). The clock component also keeps track of the activity of the model. It broadcasts the *time advance* event with the smallest scheduled time in the schedule list as a parameter, when the model becomes idle.

  To simulate as-fast-as-possible simulation for distributed models (models that communicate via ports), the idea of a clock component is similar. However, the clock component must tolerate network delay, if some or all of the connections are established via a network. Because of this delay, a model may receive a message from another model, which is sent at a time in the past (in terms of the local virtual time of the receiver). In that case, the receiver must be rolled back to the message time. This timewarp issue is discussed in [31]. It is not in the scope of this thesis.

3.3 Special Event: $after$

In DCharts, $after$ is regarded as a special event. It is comparable to the *tm* (timeout) in David Harel's STATEMATE statecharts. Though it appears in the event part E of a transition, it is not really an event

but a schedule request. It also has a special syntax $after(t)$ where t is the schedule time (in seconds) as a parameter. For example, if t is 10, the transition with such an E will be triggered after 10 seconds in real-time (if the model stays in its *SRC* state).

The meaning of "after 10 seconds" must be clarified. *SRC* is the source state of transition t. When the model changes to *SRC* or any substate of *SRC* from the outside, transitions from *SRC* with $after(t)$ events are collected. Their scheduled times t are evaluated. The t is usually a constant float number. However, a specific implementation may allow to use expressions to specify t. Those expressions are evaluated at run-time. Each of such transitions will be triggered after the resulting number of seconds, counting from the moment when the t of all those transitions are evaluated. Of course, there might be slight difference in the timing of those transitions. The accuracy is implementation-dependent.

When the model leaves state *SRC*, the scheduled transitions from *SRC* or its substates that have not been triggered yet, are *canceled*. In particular, if Harel's algorithm for firing a transition (section 2.3.1) is used, scheduled transitions from *SRC* are re-scheduled when a self-loop on *SRC* is triggered (because $CCS(SRC, SRC)$ is the parent of *SRC*); if the alternate algorithm (section 2.3.2) is used, they are not re-scheduled or canceled (because $CCS_{alt}(SRC, SRC)$ is *SRC* itself).

4

GRAPHICAL SYNTAX AND TEXTUAL SYNTAX

The abstract syntax of DCharts is not concerned with concrete implementation. The graphical syntax and textual syntax discussed in this chapter make DCharts usable by human beings and various tools. Models can be easily and formally specified. They can also be simulated or executed in DCharts implementations that support these syntaxes.

4.1 Graphical Syntax

DCharts allows multiple syntaxes. A designer chooses his/her favorite syntax or combination of syntaxes. This section describes the graphical syntax of DCharts. Note that only part of this syntax is implemented in AToM^3. Initializer, finalizer, macros, transition priorities and submodel importation cannot be specified in AToM^3 at this time, and thus require the designers to manually write them with the textual syntax. However, they do have a graphical representation, which will be implemented in AToM^3 in the future.

4.1.1 State Hierarchy

States in DCharts are shown as circles or round-corner boxes. If a state is drawn as a circle, it is a leaf state, which means it cannot contain any substate. When a state is drawn as a round-corner box, it is a composite state. A *composite state* must have at least one substate in it. Those substates can be composite states or leaf states. The set of composite states and leaf states, and the parent-children relations among them defined in a DCharts model, are called the *state hierarchy* of the model. As an example, Figure 4.1 shows the state hierarchy of an imaginary model: the model has states `A`, `B`, ..., `K`; `B` is a composite state with children `C`, `D` and `E`; composite state `D` has a child `F`; and so on. The name of a leaf state is shown inside the circle. The name of a composite state is shown in a rectangle above or inside the round-corner box.

Specific modeling tools may depict the state hierarchy in slightly different ways. In this particular DCharts meta-model in AToM^3, composite states are drawn as blue rectangles. The names are in black, shown beside the states.

4.1.2 Naming Convention

Different states may have the same name, provided that the states with the same name are not top-level states or children of the same composite state. For example, changing the name of state `C` in the previous example to `F` does not cause a conflict. However, changing its name to `D` makes it conflict with another state `D`, since both of them are children of composite state `B`.

The *path name* or *full path* of a state is a unique string that identifies a state in a model (the GUID required in the mathematical syntax). It contains the state names from the top-level superstate down to the state that is identified. The name of each state within this path is a *name component*. Different name components are separated with a dot. For example, the following path names uniquely identify all the states in Figure 4.1: `A`, `B`, `B.C`, `B.D`, `B.E`, `B.D.F`, `B.E.G`, `B.E.H`, `B.E.I`, `B.E.I.J` and `B.E.I.K`.

4.1.3 Orthogonal Components

Orthogonal components are a special kind of composite states.

Orthogonal components belonging to the same (orthogonal or non-orthogonal) composite state are separated

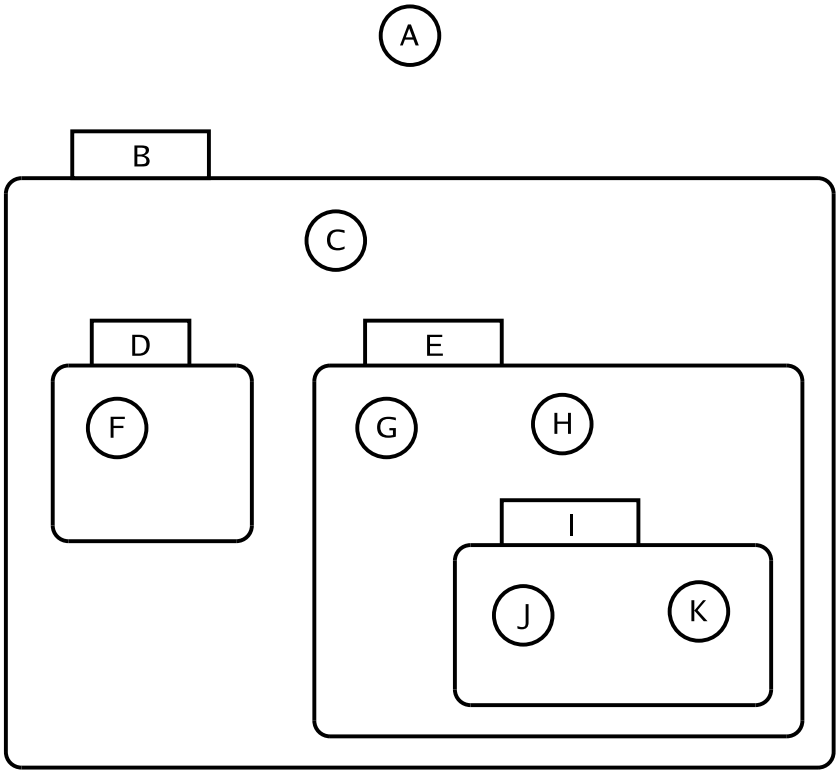

Figure 4.1: An example of the graphical representation of a state hierarchy

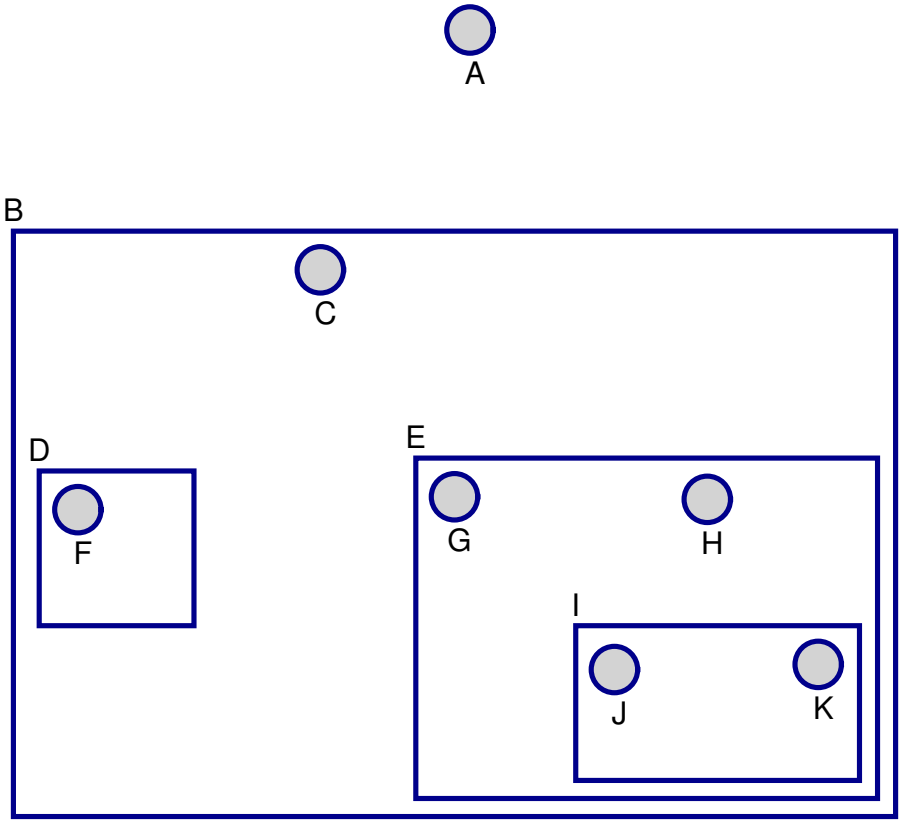

Figure 4.2: Alternate graphical representation of a state hierarchy in AToM^3

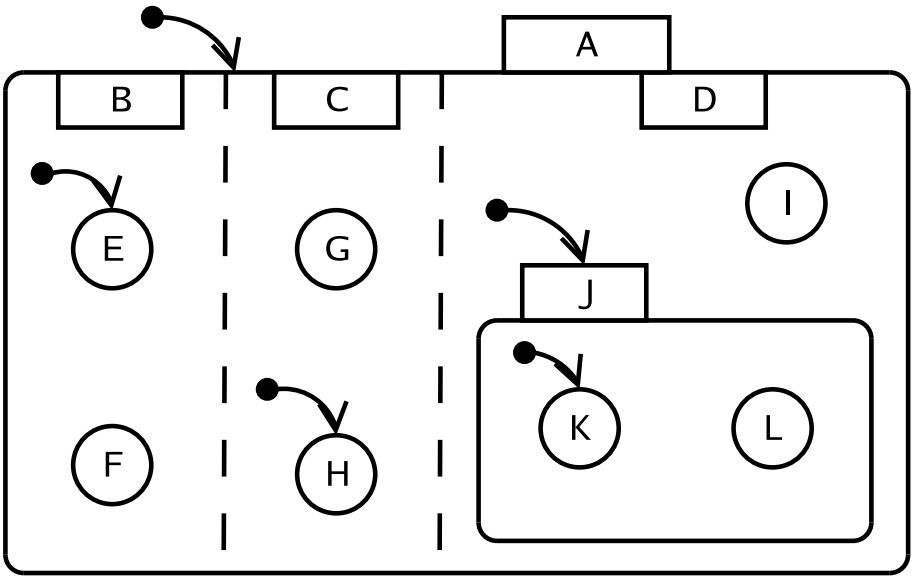

Figure 4.3: An example of the graphical representation of orthogonal components

by dashed lines across the round-corner box of the composite state. The current state of that composite state is the Cartesian product of the current states of all those orthogonal components. Substates may be defined inside each of the orthogonal components. If no substate is defined in it, the orthogonal component is a leaf state.

If a composite state has an orthogonal component as one of its children states, all its other children states must also be orthogonal components.

It is possible that an orthogonal component has orthogonal components as its children. Suppose `M.A` and `M.B` are orthogonal components of `M`, and `M.B.C` and `M.B.D` are orthogonal components of `M.B`. According to the definition of orthogonal components, the current state of `M` is equal to the Cartesian product of the current states of `M.A` and `M.B`, i.e., $S(M) = S(M.A) \times S(M.B)$ ($S(M)$ is the function to compute the current state(s) of M). Similarly, $S(M.B) = S(M.B.C) \times S(M.B.D)$. As a result, $S(M) = S(M.A) \times S(M.B.C) \times S(M.B.D)$.

The names of the orthogonal components are shown in rectangles inside them. According to the naming convention, orthogonal components of the same composite state should have different names.

Figure 4.3 shows an example of orthogonal components. Composite state `A` has three orthogonal components defined in it: `A.B`, `A.C` and `A.D`, each of which has its inner structure.

Alternately, AToM^3 shows the same example in a slightly different way (Figure 4.4).

4.1.4 Default States and Final States

Default states define the states where a model starts running, or which substates are the actual destination of a transition. *Final states* define where a simulation or execution of the model terminates. In the graphical representation of a model, a default state is drawn as a circle with a black dot pointing to it. A final state is a state with a double-line border. An example of default states and final states is given by Figure 4.5. In this example, states `A`, `B.D`, `B.D.F`, `B.E.G` and `B.E.I.J` are default states. States `B.D` and `B.E.I.K` are final states.

In this example, `B.D` is a non-leaf state. Assigning the final property to a non-leaf state is equivalent to making all its substates final. When a transition is fired with a non-leaf final state as its destination, the model will be changed to the leaf substate(s) of the destination state. Those leaf substate(s) (may be more than one because of orthogonal components in the destination state) are all final. The simulation or execution stops after the enter actions of those states are executed. If the destination state or any of its substates is an

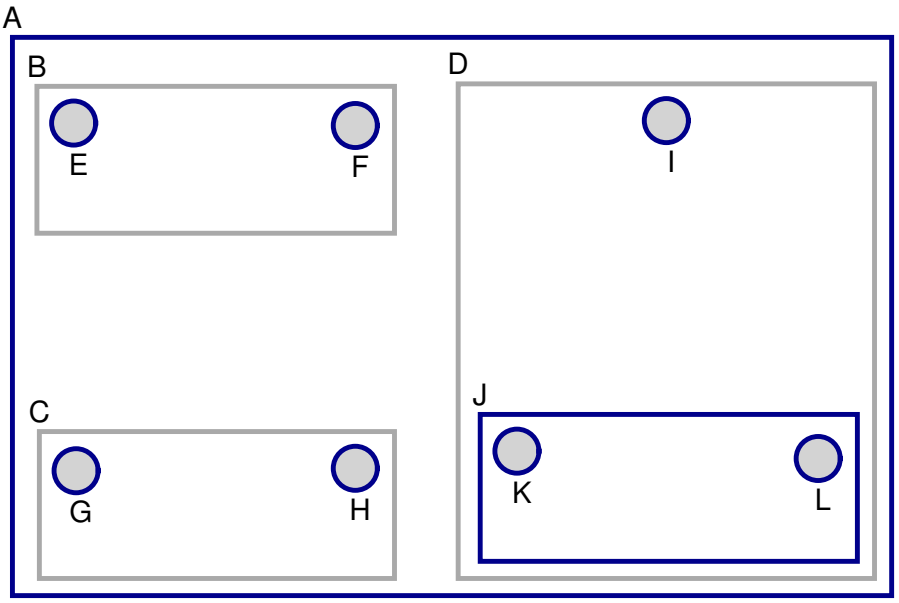

Figure 4.4: Alternate graphical representation of orthogonal components in AToM^3

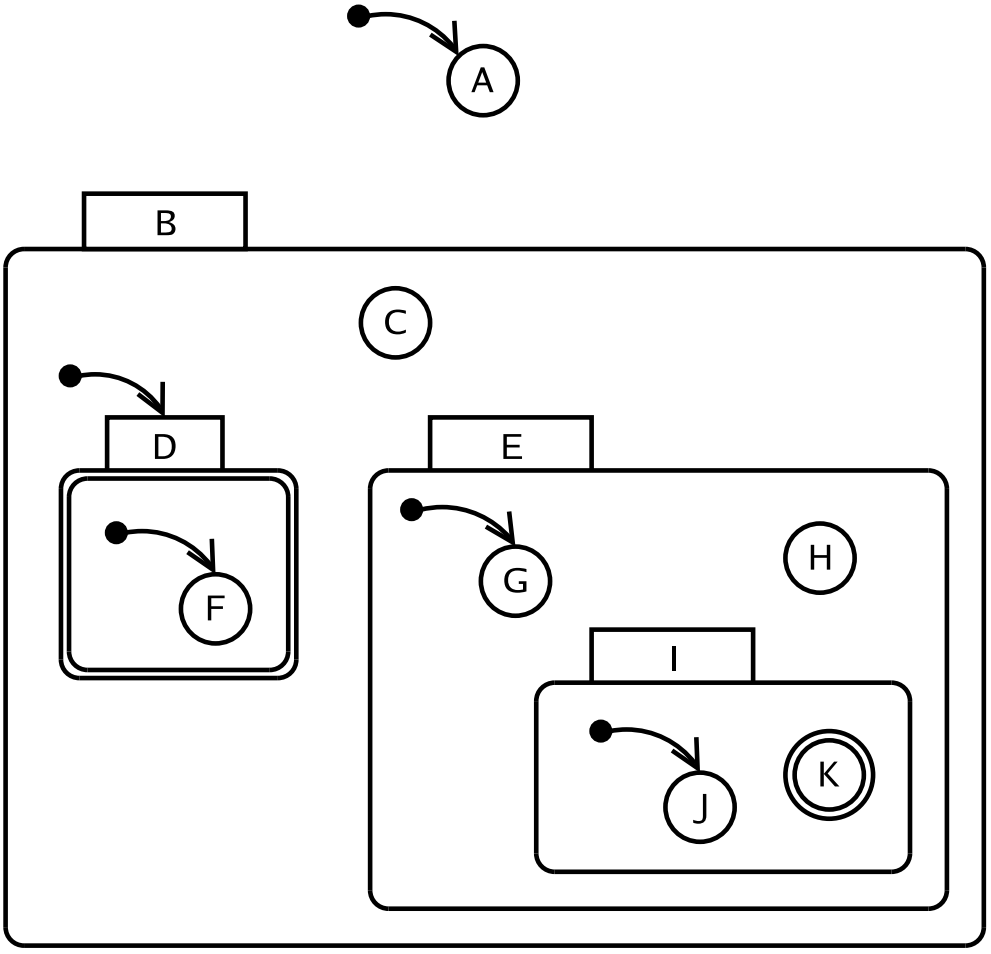

Figure 4.5: An example of the graphical representation of default states and final states

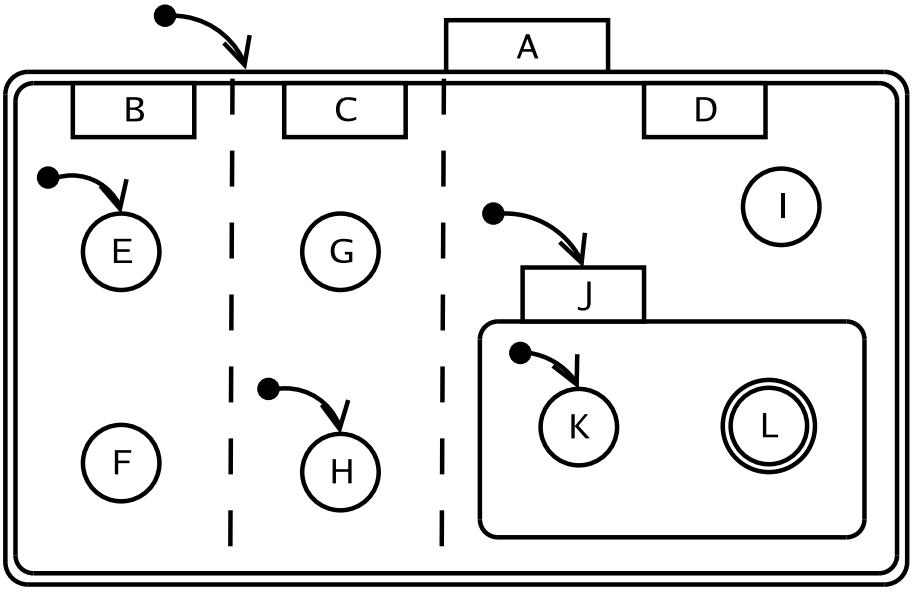

Figure 4.6: An example of the graphical representation of default states and final states with orthogonal components

importation state, the submodel must be imported and all its states become final states. Enter actions in the submodel are also executed. If a finalizer is defined for a model (discussed in section 4.3.4), the finalizer is executed as the last step before the simulation or execution halts. Note that all the above operations are triggered by a single transition. It is illegal to transition out of a final state, whether its final property is assigned by the designer or inherited from its superstates.

It is important that there must be default states among all the children of a composite state or an orthogonal component. Orthogonal components are default in their nature, so a composite state that contains orthogonal components need not explicitly specify default substates. For the example in Figure 4.6, all the three orthogonal components of state `A` are default. Besides this, states `A.B.E`, `A.C.H`, `A.D.J` and `A.D.J.K` are also default states. State `A` is a final state, and hence all its substates, including the three orthogonal components in it, are final states. Besides these, state `A.D.J.L` is a final state explicitly specified by the designer.

AToM^3 uses a different color to represent default states. The current version of AToM^3 does not support the specification of final states.

4.1.5 Transitions

Transitions of a model are triggered by events, and they react to them. They may or may not change the state of the model. They are graphically shown as arcs or arrow lines.

A transition has several properties:

- The *event name* is placed on the arc of the transition. The event name of the $after$ special event is shown as $after(t)$, where t is a float number or an expression that can be evaluated to a float number at run-time.
- The *guard* is placed after the event name between a pair of square brackets ("`[`" and "`]`").
- One or more *output actions* are placed sequentially after the event and the guard, with a leading slash "`/`". They are separated by comma "`,`" or semicolon "`;`".

The guard and output of a transition are optional and may be omitted from the graphical representation of the model for better conciseness. It is not allowed to create a transition without an event name. For a transition

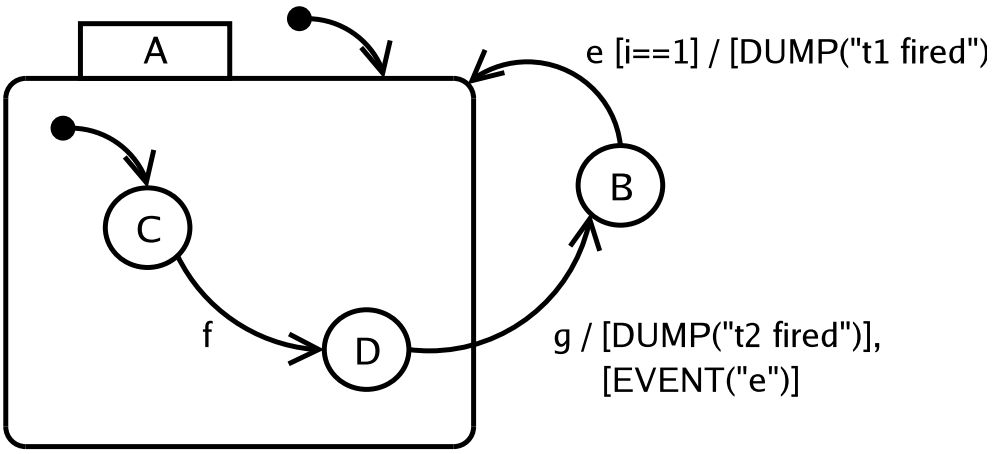

Figure 4.7: An example of the graphical representation of transitions

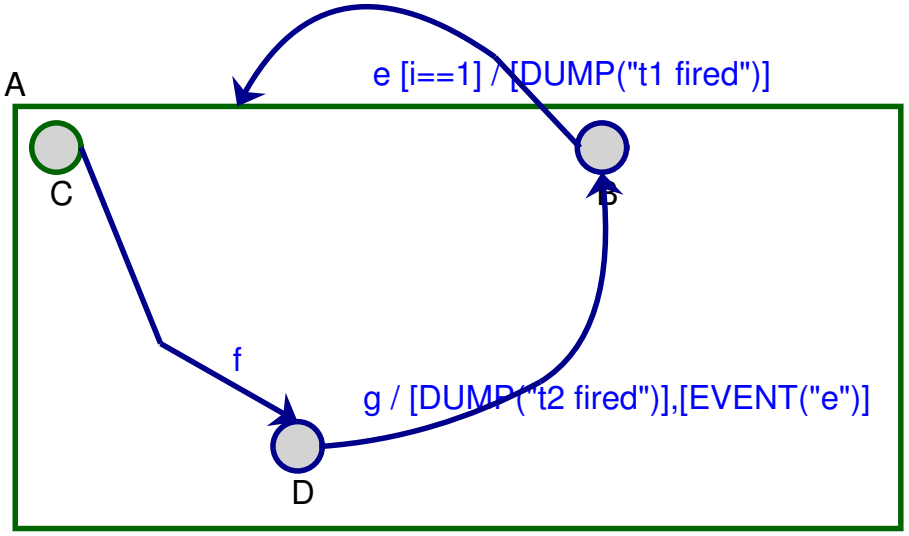

Figure 4.8: Graphical representation of transitions in AToM^3

whose triggering does not depend on any event, i.e., the transition is triggered whenever the source state is entered (and with its guard evaluated to true, if any), the model designer must explicitly specify $after(0)$ as the event name. Because DCharts is a real-time formalism, $after(0)$ does not mean to trigger the transition *at no time* but rather *as soon as possible* (after all the currently queued events have been handled).

Figure 4.7 shows a model with three transitions. (Note that the arc from a black dot to a state is not a transition but part of the notation of a default state.) The transition from `B` to `A` reacts to event `e` if and only if condition `i==1` is satisfied. As a side effect of the triggering of this transition, action `[DUMP("t1 fired")]`[1] is executed.

Figure 4.8 shows the graphical representation of the same model in AToM^3.

4.1.6 History

Though history is a state property, it is graphically shown as a leaf state in a composite state (according to David Harel's syntax). An `H` or `H*` is placed in a circle inside the composite state, depending on whether the history is a normal history or a deep history. A transition with a history as its destination is a transition with

[1] `DUMP` is a predefined macro for the specification of actions. See section 4.3.1 for a detailed description of macros.

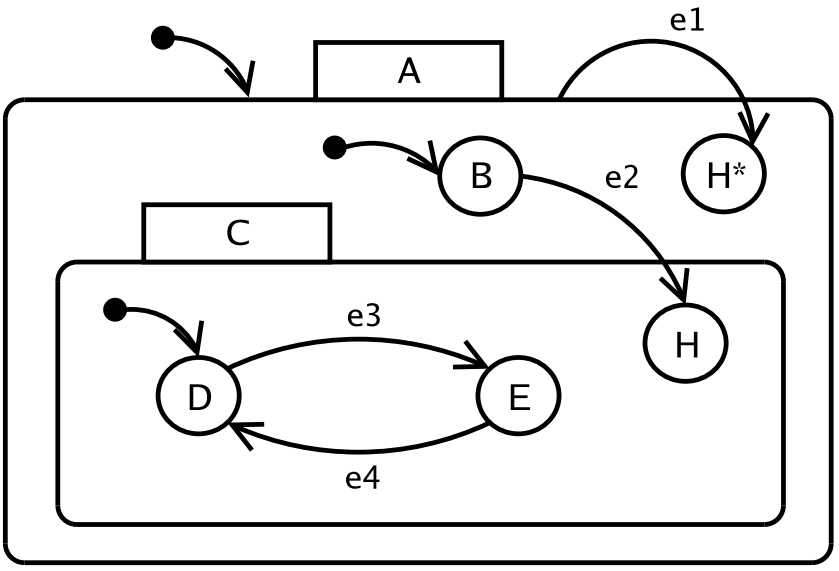

Figure 4.9: An example of the graphical representation of history states

$HS_T = true$ to the owner of the history in the mathematical syntax.

This graphical representation assumes that only composite states can have histories. Leaf states, since nothing can be placed inside them, cannot have histories. As such, this restriction has a positive effect on the well-formedness of a model. Leaf states have no substates and thus they need not have any history. (As an exception, importation states may have histories, and those histories apply to the models that they import. In those cases, importation states are drawn as composite states rather than leaf states.)

An example of history states is shown in Figure 4.9. In this example, state `A` has a deep history, and state `A.C` has a normal history. If the transition reacting to event `e1` is represented in the abstract syntax, its *DES* is `A`, and its HS_T is equal to *true*. Similarly, for the transition reacting to `e2`, *DES* is `A.C` and HS_T is *true*.

Obviously, a composite state cannot have both a normal history and a deep history. In such a case, its deep history always overrides the normal history, and the latter is ignored. This is also because in the abstract syntax, history is a property of a state with value *None*, *Normal* or *Deep*. It is impossible to assign different values to this single property.

Figure 4.10 shows the graphical representation of history states in *AToM*^{3}.

4.1.7 Enter/Exit Actions

Enter actions and exit actions are shown as UML notes in a state. An example is given in Figure 4.11.

In AToM^3, enter/exit actions are not graphically visible. They are hidden properties of states.

4.1.8 Importation

Importation is graphically shown as UML comments. The name of the file that contains the imported model is given in the comment.[2] As an example, Figure 4.12 includes state `B`, where submodel `submodel.des` (`des`, short for “model description”, is the postfix used by SVM) is imported.

The current version of the DCharts meta-model in AToM^3 does not support the specification of importation.

4.1.9 Ports

An example of ports is shown in Figure 4.13. There are three ports: `p`, `q` and `r`. A port is graphically represented as a box with one or two openings. Its name is placed beside the icon. There are different graphical

[2]It is assumed that each model is defined in a separate file, whether the graphical syntax or the textual syntax is used.

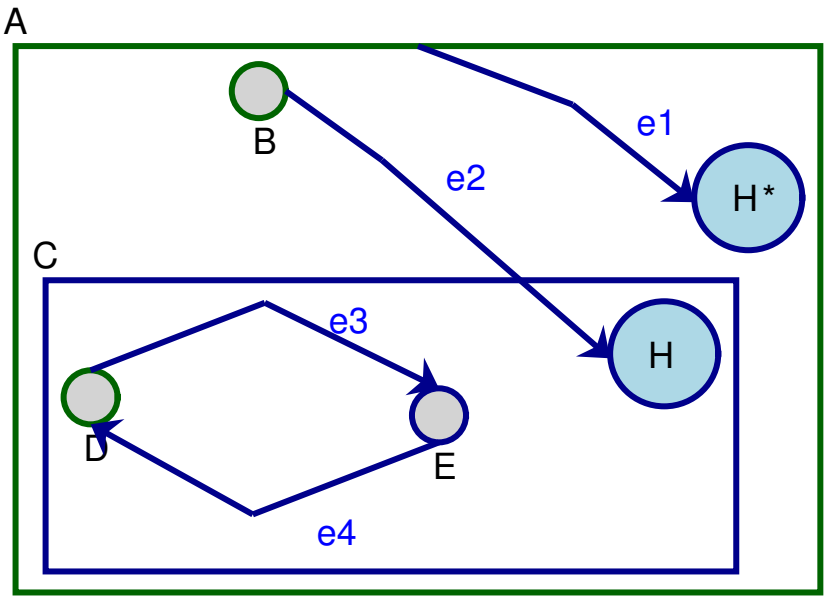

Figure 4.10: Graphical representation of history states in *AToM*^{3}

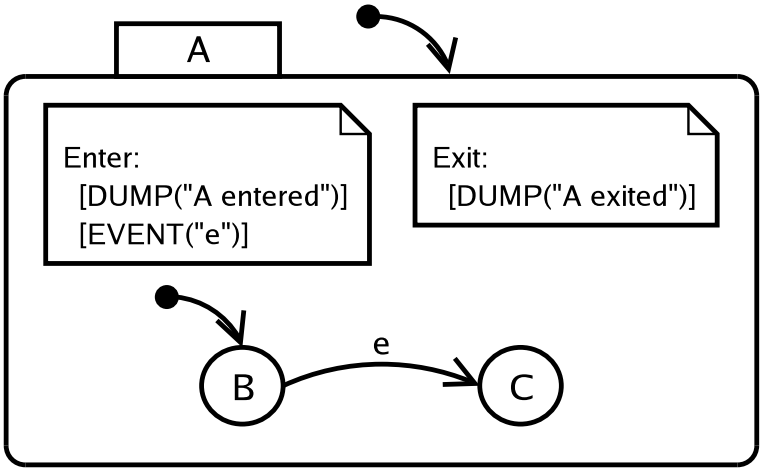

Figure 4.11: An example of the graphical representation of enter actions and exit actions

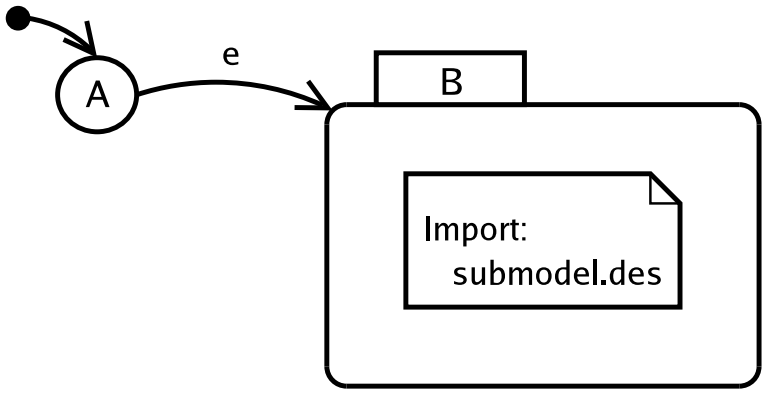

Figure 4.12: An example of the graphical representation of importation

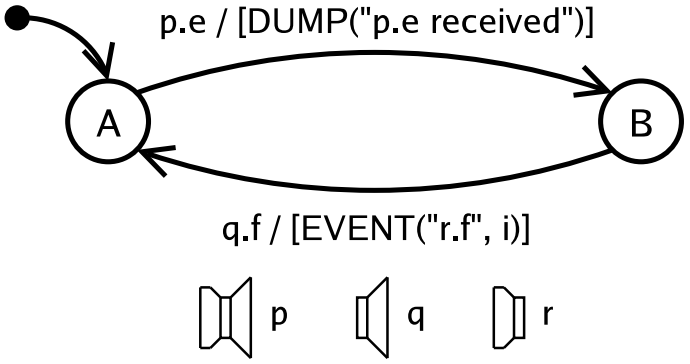

Figure 4.13: An example of the graphical representation of ports

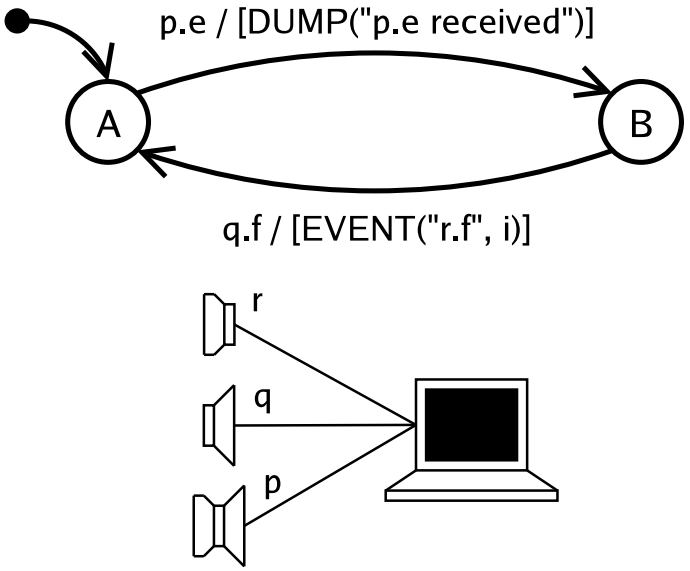

Figure 4.14: An example of the graphical representation of connections

representations for different types of ports. In this example, `p` is an inout-port, `q` is an in-port, and `r` is an out-port. (The direction of a port icon does not matter. Different types of ports have different icons.)

Once a port is defined, transitions in the model can refer to it by name. For example, the transition from `A` to `B` reacts to event `p.e`, where `p` is the name of a port, and `e` is the name of a message coming from that port. The other transition from `B` to `A` reacts to event `q.f`. In its actions, event `r.f` is output with parameter `i`. As described in the abstract syntax, an event generated by the model with a dot is considered as an out-going message. As a result, message `f` will be sent asynchronously via port `r` with parameter `i`.

4.1.10 Connections

The definition of connections is not required for servers, the models that passively wait for incoming service requests, and provide those services to the clients.

In the clients, connections must be specified in addition to the ports. The clients must also locate the servers. A specific implementation of DCharts may provide a number of ways to locate those servers. The following mechanisms are most common:

- Locate servers by their names. The clients specify name patterns of the servers that they want to connect to. All the servers with names matching those name patterns are selected by the simulation/execution environments. They attempt to establish the required connections between the servers and the clients. The name patterns are strings of UNIX regular expressions.
- Locate servers by their types. The clients specify types of the servers. The *type of a model* is an unordered list of the types of all its ports. For example, the type of a model with 2 in-ports, 1 out-port and 3 inout-ports is $\{in, in, out, inout, inout, inout\}$. This scheme selects all the servers that match a given type.
- Locate servers by their behavioral keywords. A server may define several keywords that define its behavior or the services that it provides. If so, the clients can use the keywords to match the servers.
- Any combination of the above schemes.

The graphical representation of three connections is given in Figure 4.14. The scheme to locate the server is a property of the connections and is not shown graphically. Connections are the links between the ports

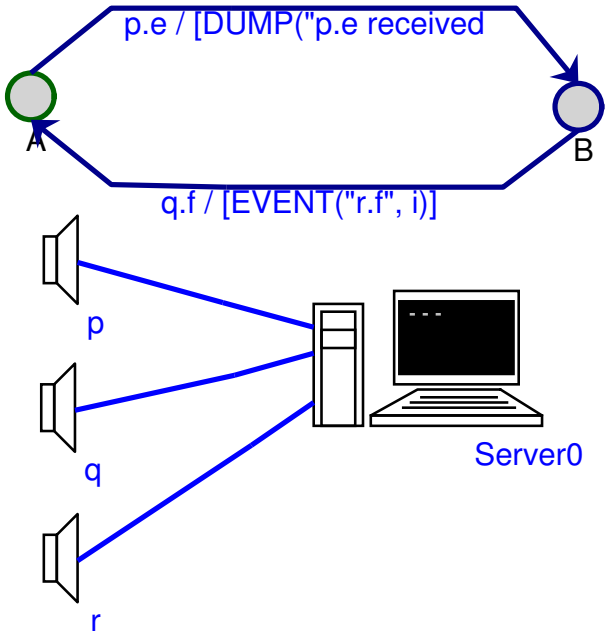

Figure 4.15: Alternate graphical representation of connections in AToM^3

and the server. A connection must also specify the port of the server to which the client is connected. This specification is not shown graphically, either.

In the figure, port `p`, because it is an inout-port (supporting input, output, or both), can be connected to ports of any type of the server; `q` can only be connected to out-ports or inout-ports of the server; `r` can only be connected to in-ports or inout-ports of the server.

The graphical representation of connections is a little different in AToM^3 (Figure 4.15).

4.2 Textual Syntax

The textual syntax of DCharts makes it possible to manually write a model in a text file. Such a file can be easily processed by a simulator or executor. The syntax discussed in this section is specific to SVM. Other implementations of DCharts may use other textual syntaxes or other file formats.

For the simulation in SVM, each model is written in a separate file. A model may import others by assigning identifiers to their file names and using those identifiers as properties for its importation states.

4.2.1 Descriptors

A model description consists of several parts, each of which starts with a *descriptor*, such as `STATECHART` and `TRANSITION`. A model may specify the same descriptor many times.

In the text file of a model description, a descriptor is a single line with an ending colon ":". All the following lines pertain to that descriptor until a new descriptor appears. *Empty lines* are lines that contain only spaces, tabs and/or comments (see below). Empty lines are automatically ignored.

Descriptor `STATECHART` is necessary for every model, which defines the state hierarchy. As least one default state must be defined for the hierarchy. Other descriptors are optional.

4.2.2 State Hierarchy

The state hierarchy of a model is written using indentation so it is easily understood by designers. Descriptor `STATECHART` starts the definition of the state hierarchy. If there are multiple `STATECHART` descriptors in a single model, the definitions under all of them are literally combined.

```
A
B
    C
    D
        F
    E
        G
        H
        I
            J
            K
```

Table 4.1: An example of the textual representation of a simple state hierarchy

Symbol	Meaning	Note
[DS]	default state	The state is a default state of its parent or of the model.
[FS]	final state	The state is a final state.
[CS]	concurrent state	The state is an orthogonal component.
[HS]	history state	The state has a normal (1-level) history.
[HS*]	deep history state	The state has a deep history.
[ITF]	inner transition first	
[OTF]	outer transition first	
[RTO]	reverse transition order	

Table 4.2: State properties in the textual syntax

Under the STATECHART descriptor, the names of the states are written on separate lines. The indentation of those names represents the parent-children relationship. A name with more leading spaces becomes a child state of the state defined in the previous line. For example, the graphical model in Figure 4.1 is written as the textual representation in Table 4.1.

The amount of indentation spaces for the first child of a composite state is not important, as long as all its children have exactly the same indentation. From this example, B.C is the first child of B with 4 more leading spaces. For B.D and B.E to be children of B, they should have exactly the same indentation.

The use of TAB is not recommended, since different text editors display a TAB character with different numbers of spaces. In SVM, a TAB is always equivalent to 4 spaces.

The naming convention is the same as the graphical syntax. Different states may have the same name, provided that their path names are different (i.e., they have globally unique fully qualified names).

4.2.3 State Properties

State properties are written after the names of the states. A state may have 0 or more properties. Each property is enclosed by a pair of square brackets. There can be 0 or more spaces between the state name and the first property, and between two adjacent properties.

The state properties are explained in Table 4.2.

As an example, the model in Figure 4.5 is textually written in Table 4.3.

4.2.4 Orthogonal Components

Orthogonal components are states with [CS] properties. As required by their semantics, [CS] always comes with [DS]. If any of a composite state's children has a [CS] property, all other children of the same parent should also have [CS].

```
STATECHART:
    A [DS]
    B
        C
        D [DS] [FS]
            F [DS]
        E
            G [DS]
            H
            I
                J [DS]
                K [FS]
```

Table 4.3: An example of the textual representation of state properties

```
STATECHART:
    A [DS]
        B [CS] [DS]
            E [DS]
            F
        C [CS] [DS]
            G
            H [DS]
        D [CS] [DS]
            I
            J [DS]
                K [DS]
                L
```

Table 4.4: An example of the textual representation of orthogonal components

```
STATECHART:
    A [DS]
        C [DS]
        D
    B

TRANSITION:
    S: B
    N: A
    E: e
    C: i==1
    O: [DUMP("t1 fired")]

TRANSITION:
    S: A.C
    N: A.D
    E: f

TRANSITION:
    S: A.D
    N: B
    E: g
    O: [DUMP("t2 fired")]
       [EVENT("e")]
```

Table 4.5: An example of the textual representation of transitions

Table 4.4 shows the textual description of the same model as Figure 4.3.

4.2.5 Transitions

Each transitions is written under the `TRANSITION` descriptor. A transition may have the following 5 properties. Some of them are optional.

- The `S` (source state) property is the *SRC* of a transition in the abstract syntax.
- The `N` (new state) property is the *DES* of a transition in the abstract syntax.
- The `E` (event) or `T` (time) property is the E of a transition in the abstract syntax.
- The `C` (condition) property (optional) is the G of a transition in the abstract syntax.
- The `O` (output) property (optional) is the γ of a transition in the abstract syntax.

The `S` property, the `N` property, and either the `E` property or the `T` property are obligatory for each transition. The value of the `E` property is the name of the event that triggers the transition. The value of the `T` property, which may be a Python expression to be evaluated at run-time, is the time t (in seconds) to be scheduled in advance. It is equivalent to the $after(t)$ event in the abstract syntax. A transition cannot have both `E` and `T` properties.

The properties of a transition are specified in separate lines after the `TRANSITION` descriptor. Their order is not important. For transitions that have multiple output actions, each action is written on a single line, and all those actions are left-aligned with 0 or more leading spaces. Similarly, more than one guard can be written in consecutive lines with left-alignment. Those guards have "*and*" relations. Alternatively, they can also be written in a single line with the *and* operator in the constraint language (Python) between them.

```
TRANSITION:
    S: A
    N: B
    T: 0.5 + i
    C: x==1
       y==2
```

Table 4.6: An example of the textual representation of a timed transition

```
STATECHART:
  A [DS]

TRANSITION: [1]
  S: A
  N: A
  E: e
  O: [DUMP(``t1'')]

TRANSITION: [0]
  S: A
  N: A
  E: e
  O: [DUMP(``t2'')]
```

Table 4.7: An example of the textual representation of priority numbers

As an example, the model in Figure 4.7 is textually written as the textual representation in Table 4.5.

Timed transitions are a special kind of transitions that have the `T` property instead of `E`. An example of timed transition is shown in Table 4.6. This transition is triggered $0.5 + i$ seconds after state `A` is entered. The transition is enabled after the scheduled time only if $x == 1$ and $y == 2$.

4.2.6 Priority Numbers

An integer number *Prio* is assigned to each transition. Whenever there is a conflict that cannot be solved with the ITF and OTF scheme, the priority numbers are used. The priority number is placed between square brackets after the `TRANSITION` descriptor. By default, each transition has a priority number of 0.

For example, two transitions are defined in Table 4.7. The model is initialized in state `A`. When event `e` occurs, both transitions are enabled and hence there is a conflict that cannot be solved with the ITF and OTF convention (because they have the same source state *SRC*). In this case, their priority numbers are used to solve the conflict. Since the second transition has a smaller priority number, it has higher priority and is thus fired.

4.2.7 History

A state with history is simply written as a state name followed by an `[HS]` or `[HS*]` property. Above this, there must be a means for a transition to choose whether the destination is a state itself or the history of the state. This is accomplished by an additional `[HS]` attribute after the `TRANSITION` descriptor. A transition with `[HS]` after its `TRANSITION` descriptor goes to the (normal or deep) history of its destination state *DES*; a transition without this attribute goes to *DES* or the default substates of *DES*.

As an example, the model in Figure 4.9 is textually written in Table 4.8. The transitions reacting to events `e1` and `e2` have an `[HS]` attribute, so they go to the histories of their *DES* states. Since states `A.C.D` and

```
STATECHART:
    A [DS] [HS*]
        B [DS]
        C [HS]
            D [DS]
            E

TRANSITION: [HS]
    S: A
    N: A
    E: e1

TRANSITION: [HS]
    S: A.B
    N: A.C
    E: e2

TRANSITION:
    S: A.C.D
    N: A.C.E
    E: e3

TRANSITION:
    S: A.C.E
    N: A.C.D
    E: e4
```

Table 4.8: An example of the textual representation of histories

```
STATECHART:
    A [DS]
        B [DS]
        C

ENTER:
    N: A
    O: [DUMP("A entered")]
       [EVENT("e")]

EXIT:
    S: A
    O: [DUMP("A exited")]

TRANSITION:
    S: A.B
    N: A.C
    E: e
```

Table 4.9: An example of the textual representation of an enter action and an exit action

`A.C.E` do not have history, adding `[HS]` attribute to the transitions reacting to `e3` and `e4` does not change the behavior of those transitions.

4.2.8 Enter/Exit Actions

Enter actions of a state are written under the `ENTER` descriptor. Exit actions are written under the `EXIT` descriptor. There are two obligatory properties and one optional property for enter actions and exit actions:

- The `N` (source state) property or the `S` (new state) property specifies the state of those actions. For enter actions, since they are executed when a state is entered, the `N` property is used to identify the state. Conversely, for exit actions, since they are executed when a state is exited, the `S` property is used.
- The `O` (output) property specifies the actions to be executed. If there are multiple actions, they are written on consecutive lines and left aligned.
- The `C` (condition) property specifies the guard to be satisfied. It is similar to the `C` property of a transition. The guard is evaluated when the state is entered/exited.

  There may be multiple parts of enter/exit actions defined for a single state. This is done with multiple `ENTER` or `EXIT` descriptors with the same `N` or `S` property. The guards of those parts are mutually independent. Each guard only controls the execution of the actions under one descriptor.

The model with an enter action and an exit action in Figure 4.11 is translated into the textual representation in Table 4.9.

4.2.9 Importation

Definition of importation in a model is separated into two parts. Under the `IMPORTATION` descriptor, one or more models can be defined as submodels. Unique IDs are assigned to those models. Those IDs can then be used as properties of states in the definition of the state hierarchy. A state with a submodel ID as a property becomes an importation state. It is not allowed to define substates for it.

There can be one or more submodel definitions under an `IMPORTATION` descriptor, and there can be multiple `IMPORTATION` descriptors in a single model. Each submodel definition is written as "`ModelID = FileName`"

```
IMPORTATION:
    sub0 = submodel.des

STATECHART:
    A [DS]
    B [sub0]

TRANSITION:
    S: A
    N: B
    E: e
```

Table 4.10: An example of the textual representation of an importation

on a single line, where `ModelID` is the user-defined ID of the submodel, and `FileName` is the name of the file that contains the model to be imported.

For example, the model in Figure 4.12 is written as Table 4.10. `sub0` is the ID of the submodel defined in file `submodel.des`. Designers can choose any ID consisting of characters and numbers, except the pre-defined state properties. The submodel is imported into state `B`.

4.2.10 Ports

A `PORT` descriptor is used to specify a port of a model. Properties of the port are written on separate lines after the descriptor.

- The `name` property specifies the GUID of a port. Every port of a model must have a unique ID.
- The `type` property specifies the type of a port. Possible values are `in`, `out` and `inout`.
- The `buffer` property is reserved for later versions. Its may be used to specify a queue or stack that stores the incoming messages.

Properties `name` and `type` are obligatory and must be specified exactly once for each port. `buffer` is optional (not implemented currently).

As an example, the model in Figure 4.13 is written as Table 4.11.

4.2.11 Connections

Before connections can be established, servers that passively wait for incoming connection requests must be located. The name patterns or types of those servers are specified under the `COMPONENT` descriptor. Like the `PORT` descriptor, each `COMPONENT` descriptor is followed by the properties of a component or a group of components that matches a certain criteria:

- The `id` property defines a GUID for the component (or group of component). Each group of components that matches a name pattern must be assigned a unique GUID.
- The `name` property, unlike the name of a port, specifies a name pattern for the group of components. The format of the name pattern follows the conventions of UNIX regular expressions. All the components with a name matching that pattern are selected as members of the group. Hence, a message sending to a group of components will be broadcast to all its members.

  For example, names "`model1`", "`model`" and "`model123`" match name pattern "`model[0-9]*`".
- The `type` property specifies the type of a group of components. It is a string that lists the in-ports, out-ports and inout-ports. An in-port is listed as `in`; an out-port is listed as `out`; and an inout-port is listed as `inout`. Multiple ports are separated by one or more spaces. An integer number can be added before

```
STATECHART:
    A [DS]
    B

PORT:
    name = p
    type = inout

PORT:
    name = q
    type = in

PORT:
    name = r
    type = out

TRANSITION:
    S: A
    N: B
    E: p.e
    O: [DUMP("p.e received")]

TRANSITION:
    S: B
    N: A
    E: q.f
    O: [EVENT("r.f", i)]
```

Table 4.11: An example of the textual representation of ports

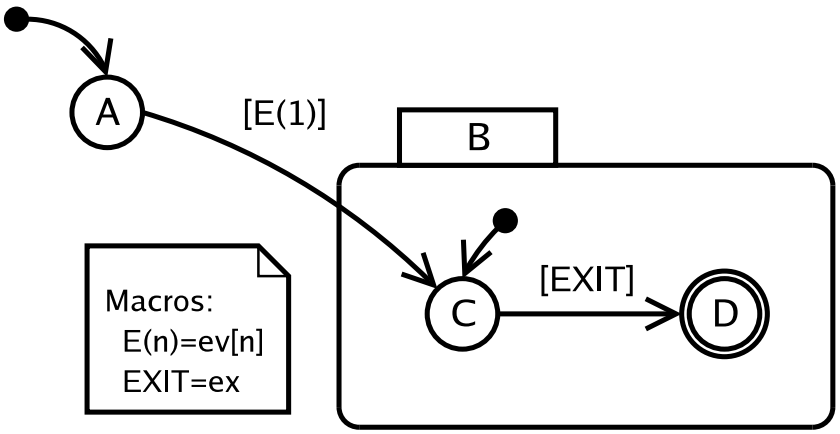

Figure 4.16: An example of the graphical representation of macros

a type to specify multiple ports of the same type. For example, "`type = in in out inout inout`" matches components with 2 in-ports, 1 out-port, and 2 inout-ports. "`type = 2 in out 1 inout 1 inout`" has exactly the same effect.

The `name` property or the `type` property or both must be specified for each group.

Connections are established between ports and the ports of the servers under the `CONNECTIONS` descriptor. One or more links can be defined. The left-hand side of a link is connected with the right-hand side with double hyphens ("`--`"). Suppose the ID of a component group is `C` and the model wants to connect to port `p` of the group with its port `q`, the link should be written as either "`q -- C.p`" or "`C.p -- q`".

As an example, suppose a server is given ID "`Server0`" and it has inout-port `p`, in-port `q` and out-port `r`, the model in Figure 4.14 is written as Table 4.12.

4.3 Extended Syntax

SVM takes advantage of the textual model description format and extends the DCharts textual syntax. The syntactic extensions discussed in this section do not have special graphical representations. In the graphical form of DCharts models, they are usually shown as UML comments where appropriate.

4.3.1 Macros

Macros are used to literally substitute texts in model descriptions. They are written under the `MACRO` descriptor.

An Example

In the graphical form, macros are defined as a UML comment at the top level, as shown in Figure 4.16. The textual description of the same model is in Table 4.13. In this example, macros `E(n)=ev[n]` and `EXIT=ex` are defined. To use those macros in the model description, put the name of a macro in a pair of square brackets, and replace all the necessary parameters with values. In the figure, macro `EXIT` is used as an event name (`[EXIT]`). It is equivalent to `ex` in this case.

Macros can be used wherever text is written. For example, they can be used in event names, in guards, in actions, in importations, and so on. In the textual form of a model, they can even be used in the specification of state hierarchy, ports, connections and so on. In particular, the values of macros can also be used as descriptors, with the exception of the `MACRO` descriptor.

On the right-hand side of a macro definition, other macros that are defined before it can be used without ambiguity. However, it may be a fatal error to use this macro itself or the macros defined after it.

```
STATECHART:
    A [DS]
    B

PORT:
    name = p
    type = inout

PORT:
    name = q
    type = in

PORT:
    name = r
    type = out

COMPONENT:
    id   = Server0
    name = model[0-9]*
    type = in out inout

CONNECTIONS:
    p -- Server0.p
    q -- Server0.r
    r -- Server0.q

TRANSITION:
    S: A
    N: B
    E: p.e
    O: [DUMP("p.e received")]

TRANSITION:
    S: B
    N: A
    E: q.f
    O: [EVENT("r.f", i)]
```

Table 4.12: An example of the textual representation of connections

```
STATECHART:
    A [DS]
    B
        C [DS]
        D [FS]

MACRO:
    E(n) = ev[n]
    EXIT = ex

TRANSITION:
    S: A
    N: B.C
    E: [E(1)]

TRANSITION:
    S: B.C
    N: B.D
    E: [EXIT]
```

Table 4.13: An example of the textual representation of macros

Parameters

A macro may carry 1 or more parameters. In the left-hand side of a macro definition, the formal parameters are specified in a way similar to the parameters of a Python function. Default values can be given to all or some of the parameters. If only part of the parameters are given default values, the parameters that do not have default values must be specified before the parameters that have default values. For example, the following specifications of the left-hand sides of macro definitions are valid:

- `my_macro(p1, p2, p3)`
- `my_macro(p1, p2, p3="hello")`
- `my_macro(p1, p2, p3=[another_macro(1)])` (Suppose macro `another_macro(n)` is defined before `my_macro`.)

On the right-hand side, parameters are referred to with their name between square brackets. For example, parameters `p1`, `p2` and `p3` are referred to with `[p1]`, `[p2]` and `[p3]`, respectively.

To use a macro, all the parameters must have their values. Values must be explicitly assigned to the parameters that do not have default values. 0 or more ending parameters that have default values can be omitted. For example, to use macro `my_macro(p1, p2, p3="hello")`, the following statements are valid:

- `[my_macro(1, 2, "hello")]`
- `[my_macro(1, 2)]`
- `[my_macro(p2=2, p1=1)]`
- `[my_macro(p3="hello", 1, 2)]`

In all these use cases, the actual values of `p1`, `p2` and `p3` are `1`, `2` and `"hello"`, respectively.

Macros can be used as parameters of other macros.

Brackets for parameters cannot be omitted even if in the definition of a macro, all the parameters have default values, or there is no parameter specified between the brackets. For example, to use `my_macro(p="hello")`

= ... or my_macro() = ..., the user must include the brackets ([my_macro()]). However, to use macro my_macro = ..., simply write [my_macro].

Pre-defined Macros

SVM pre-defines a number of macros. Pre-defined macros can be used in every model without being explicitly defined.

- EVENT(ev, p=[]) = eventhandler.event([ev], [p]).

  This macro is used to raise an event. The event name is given by parameter ev. Parameter p can be used as a parameter or a list of parameters for the event. By default it is an empty Python list.

  eventhandler is an internal object of the SVM simulation environment. For each simulation, there is exactly one instance of eventhandler. Its event method appends an event to the end of its global event list.

- EXTEVENT(ev, p, rec=None) = eventhandler.external_event([ev], [p], [rec])

  This macro is used to send an external event (or in another word, send a message to a remote component). ev is the event name, p is the parameter, and rec is a set of specific components that receive the message.

  The event name contains the name of a port and the message, separated by a dot. For example, to send a massage m via port p, the value of ev is equal to "p.m". If the user wants to restrict the receivers to the components named model0 and model1, the value of rec should be equal to ["model0", "model1"].

  When rec is not given or rec is equal to None, all the components in the group identified by the port name will receive the (possibly duplicated) message. In that case, macro EXTEVENT is the same as EVENT, except that EXTEVENT sends the message immediately in an asynchronous way, while EVENT queues the message in the global event list, and sends it asynchronously later when the simulator/executor is free.

- DUMP(msg) = dump_message([msg])

  This macro dumps a message to the output device. If SVM is run in the text mode, the message is printed on the console. If SVM is run with the default graphical interface, the message is displayed in the output box of the main window.

- INSTATE(state, check_substate=0) = eventhandler.is_in_state([state], [check_substate])

  This macro checks whether the model is currently in a specific state. The state parameter is the name of the state. check_substate is default to 0, which means the simulator does not check the substate of the given state. Hence, if the given state is not a leaf state and check_substate is equal to 0, the result is always 0. If the model is in the given state or its substates, and check_substate is equal to 1, the result is 1. eventhandler.is_in_state is a method of the simulator that handles this inquiry.

  This macro should only be used in the guards of transitions, as the DCharts formalism requires that the actions of a model cannot reflect upon the current state of the model itself.

- PARAMS = eventhandler.get_event_params()

  This macro can only be used in the guards or output of transitions. It returns the parameter of the event that triggers a transition. If the parameter is a list, [PARAMS][i] can be used to access individual elements in the list, where i is an integer between 0 and $len([PARAMS]) - 1$ (inclusively).

- SENDER = eventhandler.get_event_sender()

  This macro can only be used in the guards or output of transitions, and the transitions must be triggered by messages from remote components. It returns the sender (a model name) of the message.

- SYNCALL(event, params, listento) = eventhandler.synchronous_call([event], [params], [listento])

```
IMPORTATION:
    sub0 = submodel.des

STATECHART:
    A [DS]
    B [sub0] [DUMP(msg)=print "sub0 says: "+[msg]]

TRANSITION:
    S: A
    N: B
    E: e
```

Table 4.14: An example of the textual representation of a macro redefinition

This macro provides the synchronous call facility for the action language. A model uses this macro to send a message to a remote component, and waits for a reply via a port. `event` is the event name, as is discussed for macro `EXTEVENT`. `params` is a parameter or a list of parameters. `listento` is the name of the event to be waited for. It also follows the convention of event names discussed in `EXTEVENT`. For example, if a model sends message `m1` via port `p1` without parameter, and waits for a reply `m2` from port `p2`, the call is written as `[SYNCALL("p1.m1", [], "p2.m2")]`. The return of this call is the parameter(s) received with the reply.

This call does not return until the reply is received. If it is not the last action in the output, the actions after this are executed only after the call is finished.

- Macros `SNAPSHOTREQ` and `SNAPSHOTRET` are used for snapshot purpose. They are discussed in section 4.3.5.

Importation Parameters (Macro Redefinition)

Macros in a submodel are interpreted before the submodel is imported. They have no effect on the importing model. The importing model is allowed to modify the behavior of the submodel by redefining its macros. All the macros that are defined in the submodel and all the pre-defined macros can be redefined by the importing model. They act as parameters to the submodel.

This mechanism enhances the expressiveness of DCharts 1.0. It is the only means by which the behavior of submodels is modified. This is necessary for model reuse. Moreover, model reuse with macro redefinition protects the well-defined behavior of submodels. Only the macros defined in them (or pre-defined macros) are allowed to be modified. The importing model cannot change other parts of the submodels. [26]

To redefine macros as parameters for a submodel, the importing model imports the submodel into one of its states as described in section 4.2.9. The macro redefinitions are specified as properties of the importation state in the state hierarchy. Redefining a macro is similar to macro definition under the `MACRO` descriptor, except that it is placed between square brackets following the name of the state.

For example, the model in Table 4.14 imports submodel `submodel0.des` and assigns ID `sub0` to it. It is imported into state `B`. The importing model redefines macro `DUMP` of the submodel. `DUMP` is originally a pre-defined macro to display a message. It is redefined to print the message to the console with prefix "`sub0 says: `". Multiple macro redefinitions can be written on the same line.

4.3.2 Once Timed Transition

By default, SVM timed transitions are *repeated timed transitions*. This means a timed transition is fired repeatedly if $SRC = DES$. The simulator considers a self-loop as a state change, and hence it reschedules the timed transition from the same source state. Repeated timed transitions are equivalent to transitions with *after* event in DCharts 1.0 or the *after* transitions in David Harel's semantics.

```
TRANSITION:
    S: A
    N: A
    T: 1 [OTT]
    O: i = i + 1

TRANSITION:
    S: B
    N: B
    T: 1
    C: i < 10
    O: i = i + 1
```

Table 4.15: An example of the textual representation of a once timed transition

On the contrary, *once timed transitions* are not rescheduled for self-loops. They are fired only once even if $SRC = DES$. (Of course, if $SRC \neq DES$, the transition is always fired once.) The semantics of once timed transition is different from the special event *after* described in DCharts 1.0. They must be explicitly specified with the `[OTT]` property.

Consider the two transitions in Table 4.15. The first transition is a once timed transition (with the `[OTT]` property). When state `A` is entered from the outside, it is scheduled after 1 second. When it is fired, it increases `i` by 1. It is not rescheduled. Suppose the original value of `i` is 0. When the model is stable, the value of `i` becomes 1. The second transition is a repeated timed transition. It is rescheduled each time after it is fired. As a result, without the guard, `i` would be increasing forever if no other transition brings the model to a state other than `B`. However, with the guard `i < 10`, when the model is stable, the value of `i` is 10.

4.3.3 Global Options

Global options of a model are specified under the `OPTIONS` descriptor. Currently, three global options are supported:

- The `ModelName` global option specifies the name of the model. By default, the model name is the file name that contains the model description with the `.des` postfix removed. Designers are allowed to explicitly define model names with this option. Other models use the model name as an ID to locate the model and establish connections to it.
- `Harel` global option specifies whether the simulator should strictly obey David Harel's statecharts semantics and DCharts 1.0, or use alternate algorithms in the simulation (refer to section 2.3.2). By default `Harel` is equal to 1, which means SVM strictly follows David Harel's statecharts algorithm, and hence it can also be used as a statecharts simulator. When `Harel = 1`, the following different points are made:

  1. The alternate algorithm is used to fire all the transitions, which is different from David Harel's algorithm.
  2. Self-loops are not considered as state changes. For example, a transition with $SRC = DES$ does not cause the exit actions or the enter actions of the state to be executed. This is because the algorithm uses $CCS_{alt}(SRC, DES)$ rather than $CCS(SRC, DES)$ to compute the closest common state.
  3. Because self-loops are not state changes, timed transitions are by default once time transitions, unless the model design explicitly assigns the `[RTT]` (Repeated Timed Transition) property to them.

```
INITIALIZER:

FINALIZER:

INTERACTOR:
    setup_gui_debugger(eventhandler, debugger)
```

Table 4.16: Default values for initializer, finalizer and interactor

- `InnerTransitionFirst` global option specifies whether the model follows the inner-transition-first convention or the outer-transition-first convention. This option affects all the top-level states so that their default behavior conforms to this setting. For example, having `InnerTransitionFirst = 1`, all the top-level states get the `[ITF]` property by default, and their substates inherit this behavior by default. However, this default behavior can always be modified with an explicit `[ITF]`, `[OTF]` or `[RTO]` property for a state. `InnerTransitionFirst = 1` is just a short-hand notation for specifying `[ITF]` for every top-level state.

  The default value of `InnerTransitionFirst` is set to 0, which means all the top-level states are outer-transition-first (according to the STATEMATE semantics [5] of David Harel).

The above options are global in the scope of the whole model. They cannot be imported with submodels. When a submodel is imported, its global options are ignored.

4.3.4 Initializer, Finalizer, and Interactor

Initializer, finalizer and interactor are SVM extensions to DCharts 1.0. In most models, they are highly simulation-oriented.

Initializer is the Python code to be executed before a model starts running. It is executed even before the model is placed in its default states (so that it may be illegal to test the current state within the initializer). This code usually initializes the environment where the model is simulated or executed. For example, this code can be used to initialize all the variables that the model uses.

An initializer is written under the `INITIALIZER` descriptor. Arbitrary Python code can be written, including function definitions, if-else or switch-case conditional structures and loops.

Finalizer is used to finalize the model. It is executed after the model changes to a final state. If the final state has enter actions, those enter actions are executed before the finalizer. A finalizer is written under the `FINALIZER` descriptor. Similar to initializer, arbitrary Python code can be written.

Interactor is used to define a model-specific interface. As discussed in later chapters, SVM provides a default textual interface, a default curses interface (for Unix or Linux systems) and a default graphical interface. However, in many cases designers may want to redesign the interface for specific models. They can write Python code under the `INTERACTOR` descriptor for this purpose. The difference between the interactor and the initializer is that the initializer is executed *before* the model starts running, while the interactor is executed *while* the model is running. In the SVM implementation, an extra thread is allocated for the interactor, so that it is allowed to use an infinite loop in the interactor to handle user events received from the interface, and pass those events to the running model.

The default definition of these parts is shown in Table 4.16. The default actions of initializer and finalizer are empty. The default action of interactor, if the default graphical interface is used, is to setup the interface, which includes creating all the widgets, building a tree view of the state hierarchy, and handling GUI events afterward.

4.3.5 Snapshot

Snapshot is a powerful utility for model debugging and testing. SVM is able to snapshot a model during its simulation (but not execution). The state of the model that need to be stored (specified by the model designer), including its variables, is snapshot as a text file or a string. The snapshot contains enough information for a restore operation, which puts the model into its previous state so that simulation can restart at exactly that point. The snapshot is usually saved in a `.snp` file in the same directory as the model's. The file name of the snapshot file is the same as the text file of the model, with its postfix changed.

A snapshot request is a special event to SVM. This request may be sent from the model that is being simulated, or by the user from the user interface (for example, the default graphical interface of SVM provides a "snapshot" button). Function `eventhandler.snap_to_file(filename)` accepts a string parameter `filename` as the snapshot file name (usually ending with `.snp`), and schedules a snapshot right after the current event is handled. SVM assigns the highest priority to this request. The snapshot is taken before other scheduled events are handled, if any.

A model may also request a snapshot in the memory, and roll back to it at a later time. Pre-defined macro `[SNAPSHOTREQ(time)]` requests a snapshot (with the highest priority) and labels it with `time` (an arbitrary but increasing integer that uniquely identifies a snapshot). Macro `[SNAPSHOTRET(time)]` is used to retrieve a snapshot taken previously, and roll the model back to it. In this way, the model is able to interact with the simulator at run-time.

The designers may customize snapshotting in their models. Descriptors `BEFORESNAPSHOT`, `AFTERSNAPSHOT`, `SNAPSHOT` and `RESTORE` are dedicated for snapshot purpose.

- The `SNAPSHOT` descriptor specifies the variables (or objects) that need to be recorded in a snapshot. Each variable is written on a single line under the descriptor.
- The `BEFORESNAPSHOT` descriptor specifies a piece of arbitrary Python code to be executed immediately before a snapshot is taken. It usually is used to synchronize different parts of a model, because a snapshot can be requested at any time during a simulation, even when the model is in an unstable state. For example, if the model uses a native library that requires extra threads, the code under `BEFORESNAPSHOT` may synchronize those threads to ensure that all the variables are stable and meaningful. Similar to initializer and finalizer, the code here may contain function definitions, loops and other action-language-specific control structures.
- The `AFTERSNAPSHOT` descriptor is similar to the `BEFORESNAPSHOT` descriptor, except that the code specified under it is executed immediately after a snapshot is taken.
- the `RESTORE` descriptor is used to specify a piece of code that is executed after a snapshot is restored. For example, in Table 4.17, the code under the `RESTORE` descriptor starts playing (initially the CD is stopped) at time `cd.time`, whose value is restored by SVM, if the CD was playing at the time when the snapshot was taken.

In the example in Table 4.17, suppose `cd` is the instance of a CD-Rom controller that plays CD music. Its `time` attribute is constantly updated when the CD is playing, which indicates the current time in a track. To enable snapshotting during the playing, the designer may include such a description segment in his/her model. Its `SNAPSHOT` descriptor tells SVM that `cd.time` and `cd.playing_before_snapshot` (variables as attributes of the `cd` instance) need to be recorded in a snapshot. The code under `BEFORESNAPSHOT` pauses the playing so that the `time` attribute is not changed during the snapshot operation. When the snapshot operation is finished, the code under `AFTERSNAPSHOT` resumes the playing, if it is paused previously by the code under `BEFORESNAPSHOT`.

This model enables the user to play CD music, snapshot at any time during the playing and save the recorded state in a `.snp` file. Later when the user runs the `.snp` file with SVM, the variables specified under the `SNAPSHOT` descriptor are restored, and the code under the `RESTORE` descriptor is executed, which starts playing at exactly the recorded time.

```
SNAPSHOT:
    cd.time
    cd.playing_before_snapshot

BEFORESNAPSHOT:
    if cd.is_playing():
        cd.pause()
        cd.playing_before_snapshot = 1

AFTERSNAPSHOT:
    if cd.playing_before_snapshot:
        cd.resume()
        cd.playing_before_snapshot = 0

RESTORE:
    if cd.playing_before_snapshot:
        cd.play()
```

Table 4.17: An example of the textual representation of a snapshot/restore description

4.3.6 Model Description

The `DESCRIPTION` descriptor is used to give a short description to a model. The description is the content between the `DESCRIPTION` descriptor and the next descriptor. Empty lines are automatically removed.

If the default textual interface or the default curses interface of SVM is used, the description (if specified in the model) is printed to the console. If the default graphical interface is used, the description is displayed in the output box of the main window. In the simulation/execution environment, the description is a string stored in `eventhandler.description`. It may be used by the actions of the model.

4.3.7 Comments

Contents after a sharp "`#`" mark on the same line are considered as comments. Comments are ignored by SVM. An example of comments is given in Table 4.18.

```
#-------------------------------#
# This is an example of RESTORE #
#-------------------------------#
SNAPSHOT:  # variables to be snapshot
    cd.time
    cd.playing_before_snapshot

BEFORESNAPSHOT:  # before snapshot
    if cd.is_playing():
        cd.pause()
        cd.playing_before_snapshot = 1

AFTERSNAPSHOT:  # after snapshot
    if cd.playing_before_snapshot:
        cd.resume()
        cd.playing_before_snapshot = 0

RESTORE:  # after restore
    if cd.playing_before_snapshot:
        cd.play()
```

Table 4.18: An example of the textual representation of comments

5 MAPPINGS

Mappings between different formalisms are discussed in this chapter.

Mapping from DCharts to another formalism proves that DCharts have *at most* as much power as that formalism. This mapping provides a means to express the behavior of any DCharts model in the other formalism. The *denotational semantics* of DCharts is defined in this way. (The semantics discussed in previous chapters is *operational semantics*.) Few formalisms allow recursion in the model structure. It is impossible to map the complete DCharts formalism to them. Recursive importation is not considered in the mappings from DCharts to other formalisms discussed in this chapter.

Mapping from another formalism to DCharts proves that DCharts have *at least* as much power as the formalism. It provides a means to express the behavior of any model in the other formalism with DCharts.

5.1 Mapping from Non-recursive DCharts to Statecharts with Variables

David Harel's statecharts [4] do not formalize variables. The state hierarchy of models only allows finite and enumerable number of states. Obviously, it is impossible to map DCharts to original statecharts, since the variable sets V of DCharts models may contain variables that have infinite and continuous state space.

Now suppose the use of variables is allowed in statecharts. This variant of statecharts is called *statecharts with variables*. It becomes interesting to show that non-recursive DCharts can be mapped to this statecharts variant. Statecharts with variables are simpler than DCharts. If this mapping can be proved, it is implied that the only DCharts extensions that enhance the expressiveness of statecharts are recursion and variables.

To show that non-recursive DCharts can be mapped to statecharts with variables, the following semantic extensions must be explicitly transformed into statecharts structures:

- Importation.
- Transition priorities.
- Transition parameters.
- Ports and connections.

Since the variable set V is supported by statecharts with variables, it is not discussed in this mapping. Other semantic elements of DCharts, such as state hierarchy, history, transitions, can be directly mapped to the corresponding entities of statecharts. They are not discussed, either.

Lemma 1 *Importation of non-recursive DCharts models can be flattened to be the state hierarchy of original statecharts.*

Proof The algorithm discussed in section 2.3.3 shows a way in which importation in non-recursive DCharts can always be flattened. The result of this flattening does not contain any importation state. □

Lemma 2 *There exists an ordering over all the transitions in every DCharts model. According to this ordering, when an event causes a conflict at run-time, the first enabled transition in the list always has the highest priority.*

Proof This lemma can be proved by an algorithm which manages to find out this ordering.

In this algorithm, the transitions in a model are sequentially appended to an initially empty list l. When $|l|$ is equal to the number of transitions in the model, the algorithm terminates, and the ordering of the transitions in l satisfies the requirements in this lemma.

Before the algorithm starts, the model must be flattened with the algorithm discussed in section 2.3.3.

The algorithm is summarized below:

function merge(l_s, l)
/ merge two sets of transitions with insertion sort*
l_s: the transitions to be merged with l. The SRC of these transitions has no parent-children relation with the SRC of transitions in l. Conflicts between transitions in the two lists can only be solved with their Prio number.
l: another set of transitions
return: the union of l_s and l. The transitions are sorted by priority.
**/*

```
for t in l_s
    added = false
    for t' in l
        if E_t = E_t' ∧ Prio_t <= Prio_t' then
            insert t into l right before t'
            added = true
            break
    if !added then
        append t to the end of l
return l
```

function order(states, ITF)
/ sort the transitions*
states: a set of states belonging to the same parent
ITF: whether the parent of the states in states is set to be inner-transition-first or not
return: the list l of transitions whose SRC is in states or Substate(s) where $s \in$ states. The transitions are sorted by priority.
**/*

```
l = []
    for s in states
        l_ts = [transitions with SRC = s]
        sort l_ts in the increasing order of the Prio numbers of the transitions
        if s is ITF then
            next_ITF = true
        elif s is OTF then
            next_ITF = false
        elif s is RTO then
            next_ITF = not ITF
        else
            next_ITF = ITF
        if ITF then
            l_s = l_ts + order(C(s), next_ITF)
        else
            l_s = order(C(s), next_ITF) + l_ts
        merge(l_s, l)
return l
```

Note that it is assumed there are not two or more transitions with exactly the same total priority. If such transitions exist, conflicts among them cannot be solved with the ITF and OTF scheme and their *Prio* number is the same. The ordering of such transitions with the above algorithm is implementation-dependent and not unique.

Suppose all the top-level states are in set Top_s, and parameter *InnerTransitionFirst* contains the global option of the model that decides whether its top-level states are inner-transition-first by default. The invocation $order(Top_s, InnerTransitionFirst)$ always terminates since there are finite number of states in the model. The result is the transitions sorted by their total priority. If a state is set to be ITF, transitions from this state are always placed after transitions from the substates of this state in the transition list. The opposite is true for states with the OTF property.

The `merge` function merges two lists of transitions. It assumes that both lists are sorted according to the ITF and OTF convention, and tries to further sort the merged list in the order of *Prio* numbers. Because the source states of transitions in the two lists do not have the parent-children relation, the merging does not affect the ITF and OTF sorting. It only guarantees that a transition with smaller *Prio* number appears before the transitions with larger *Prio* numbers triggered by the same event.

According to the semantics of transition priority, the ITF and OTF settings of transitions are considered before their *Prio* numbers. This algorithm is correct because it sorts *Prio* on the basis that the ordering of ITF and OTF is already created and is preserved over the merging of two lists. □

Comments 1 Although this algorithm ensures that the transition with higher total priority always appear before the others with lower total priorities in the sorted list, it does not remove all the potential conflicts. It is still possible that two transitions have exactly the same total priority. Those two transitions ($t1$ and $t2$) always have the following properties:

- $SRC_{t1} = SRC_{t2}$ or SRC_{t1} and SRC_{t2} belong to two sibling orthogonal components. In that case, the ITF and OTF settings cannot solve the conflict, and it is possible that the model is in SRC_{t1} and SRC_{t2} at the same time.
- $E_{t1} = E_{t2}$. When this event is raised and $G_{t1} = true \wedge G_{t2} = true$ at run-time, both transitions are enabled.
- $Prio_{t1} = Prio_{t2}$. In this case, the *Prio* number cannot solve the conflict.

It is the designer's responsibility to ensure that there are no such transitions in a model. The simulator cannot statically analyze the model and find out these transitions, since their guards usually cannot be evaluated statically. If these transitions are found at run-time, only one of them is fired. The choice is random or implementation-dependent.

Comments 2 In the implementation of SVM, this algorithm that sorts transitions in the order of their priorities is employed. It effectively decreases the run-time computation for choosing a transition in case of a conflict. Because the first enabled transition in the list always has the highest priority among all the enabled transitions, the simulator simply picks the first one and triggers it. The sorting is done only once for every model or submodel in a simulation.

Lemma 3 *Transition priorities can be simulated with additional guards.*

Proof Lemma 2 suggests a way in which all the transitions can be sorted in a list l. Suppose l is statically obtained. An additional guard for each transition checks whether the transition is the first enabled transition in the list. This guard ensures that the choice of a transition in a conflict is unique and deterministic. The chosen transition always has the highest priority. Other transitions that are enabled are not fired since they order after the fired one. For simplicity, conflicts that cannot be solved with transition priorities are not considered. □

Lemma 4 *Transition parameters can be simulated with variables.*

Proof Transition parameters are themselves variables. If each transition is given a GUID, and the GUID of the transition is added to the names of its formal parameters, those parameter names share the same name space as the variable set V of the model. All the transition parameters can thus be included in the variable set. To send an event with parameters, the action simply modifies the global variables converted from the parameters of the transition that handles the event, and sends the event without parameters.

Lists can be used as variables. So if more than one event in the global event list is going to trigger the same transition, parameters can be queued in a global parameter list. □

Lemma 5 *Ports and connections can be simulated in a stand-alone statecharts model.*

Proof Ports and connections in DCharts allow to connect multiple models and run them in a single simulation. Statecharts do not provide this mechanism. However, the behavior of a combination of multiple DCharts models connected with ports can be simulated with a single stand-alone statecharts model.

Ports restrict the receivers of a message. Connections are established between ports of a model and ports of remote components whose names match a pattern. To simulate this in a statecharts model, the messages are transformed into events. The parameters are transformed into global variables (see Lemma 4). The name patterns and the port names of remote components are additional parameters sent with the event. Each transition triggered by this event checks the name pattern in its guard. Only those transitions with names (inherited from their *SRC* states) matching the pattern are triggered.

To simulate the broadcast of messages, an event is duplicated. Each transition triggered by the event regenerates the event in its output actions. The event is repeatedly handled until it is ignored because no transition is able to handle it. To avoid handling a event more than once by the same transition, the transition must remember whether it has handled the most recent event. This implies adding states or variables to the model. □

Theorem 2 *Non-recursive DCharts models can be mapped to statecharts with variables that have the same behavior.*

Proof This is easily proved on the basis of Lemma 1 to Lemma 5. □

Theorem 2 proves that non-recursive DCharts are at most as powerful as statecharts with variables. Extensions such as transition priorities, importation and ports do not enhance the expressiveness of the formalism. However, they make it easier to design modular models.

5.2 Mapping from Non-recursive DCharts to DEVS

Intuitively, since non-recursive DCharts can be mapped to statecharts with variables, and statecharts with variables are at most as powerful as DEVS, one should be able to map DCharts to DEVS. Spencer Borland has already shown the mapping from statecharts to DEVS in his Master's thesis [9]. A general method that transforms statecharts models to DEVS models has been found.

Mapping variables to DEVS is trivial, since DEVS supports variables in its nature. The state space of a statecharts with variables is transformed into $S \times v_1 \times v_2 \times \ldots \times v_n$, where S is the state set of the statecharts, and $v_1, v_2, \ldots, v_n \in V$ are the variables that appear in the model. The total state space is the Cartesian product of the state space of the enumerable states and the state space of all those variables. This total state space, which is usually infinite and continuous, becomes the state space of a DEVS model. The values of the variables are changed by the DEVS' external transitions and internal transitions as a modification on the current state.

From the discussion above, since original statecharts have been mapped to DEVS, and variables can be easily transformed into DEVS states, statecharts with variables can be mapped to DEVS models. As a result, non-recursive DCharts can also be mapped to DEVS models. This proves that non-recursive DCharts are at most as powerful as DEVS.

5.3 Mapping from Statecharts to DCharts

Transforming statecharts models to DCharts is trivial, since all the semantic elements of statecharts can be found in DCharts. The state hierarchy is directly mapped to the DCharts state hierarchy. DCharts transitions includes all the elements of statecharts transitions. The state properties in DCharts form a superset of the state properties in statecharts. As a result, it is easy to transform any statecharts model into DCharts.

This proves that DCharts are at least as powerful as statecharts.

5.4 Mapping from DEVS to DCharts

DEVS models can also be transformed into DCharts. Because of the closure under coupling of DEVS, any Coupled DEVS can be replaced by an Atomic DEVS that has the same behavior. It is not necessary to consider coupled DEVS in proving the mapping from DEVS to DCharts.

The DEVS formalism discussed here is real-time DEVS, which use the real time instead of virtual time as global time. The time unit is default to second.

Theorem 3 *DEVS models can be transformed into DCharts that have exactly the same behavior.*

Proof Different parts of an Atomic DEVS are mapped to DCharts constructs as following:

- The state set S is mapped to a single variable v of a DCharts model, whose state hierarchy has only one default state s. The state space of v is a superset of S. This variable can always be found. It can be of a primitive type such as integer or string, a list which contains multiple elements, or any other types supported by the action language. This variable is used to keep track of the model state.
- The time advance function ta and internal transition function δ_{int} are transformed into transitions with $after$ events. Each of such DCharts transitions is a self-loop on the state s. It uses $after(t)$ as its event, where t is the ta of a DEVS state. The guard of the transition checks the current state of the variable v, and tests if the model is in the old state of the DEVS model. In the output of this transition, v is modified according to the δ_{int} of the DEVS model.
- The external transition function δ_{ext} is transformed into transitions with the same event names. This transformation is similar to the transformation between δ_{int} and DCharts transitions.

  The elapsed time of an external transition can be computed with the primitive action that allows access to the time since the simulation or execution starts. Suppose the time when the last state is entered is t_last. It is obtained with the time action in the enter actions of the state. The time when the external event is received is denoted by t_event. This time can be obtained in the guard (since the time action is side-effect-free) or the output of the transition. Then the elapsed time is equal to $t_event - t_last$, which can be known in the guard and the output.
- The X (input set) and Y (output set) of the DEVS model is ignored, since DCharts do not require to explicitly declare them.
- The output function λ is transformed into action code in the output of transitions. It produces the same output as the DEVS model, according to the current state of the model.

  The events sent in the output of a DEVS transition are different from the events broadcast in a statecharts, because the first kind of events are explicitly sent to an output port. Fortunately, this kind of events are equivalent to the out-going messages in DCharts, which are textually represented as a port name and an event name separated by a dot.

□

5.5 Mapping from Programming Language Control Flow Constructs to DCharts

Though there is no rigorous definition of an action language in DCharts, it is a rule that each piece of action code in the output, enter/exit actions and all other parts of a model that allow actions, is composed of a series

of *statements*. Those action statements are primitive commands that are not modeled explicitly. (The only exceptions are the extensions added by SVM, such as initializer, interactor and finalizer. Those constructs do not belong to the DCharts formalism.)

The problem whether DCharts are capable of modeling more complex programming structures is interesting. On the one hand, designers who are familiar with programming languages tend to think in a programming way. If a formalism allows the specification similar to a programming language, it is much friendly to those designers. On the other hand, this capability demonstrates the expressiveness of the formalism. It is possible to explicitly model any control structure with such a formalism. As a result, theoretically all the control structures in a system can be formally checked by modeling them in the formalism. When the model is checked thoroughly, part of it can be converted into native code or hardware to achieve better run-time performance.

This section discusses several programming constructs of programming languages in general (e.g., the C language [32]), and their mappings to DCharts submodels. Most of the submodels introduced here can be directly imported into larger models to simplify the task of designers.

5.5.1 Statements

Statements in the C language are categorized into simple statements and compound statements. A *simple statement* ends with a semicolon "`;`" and cannot be further divided. Here is an example of two simple statements.

```
i = 0;  // a simple statement
if (i == 0) i = i + 1;  // a simple statement
```

A simple statement that only contains a semicolon is called a *null statement*:

```
;  // null statement
```

A *compound statement* is a sequence of statements enclosed by a pair of curly braces. The statements in the curly braces can be simple statements or compound statements.

```
if (i == 0) {  // a compound statement
    int a = 0;
    a = 1;
    i += a;
}
```

A compound statement that only contains a pair of curly braces is called *empty compound statement*.

```
{  // empty compound statement
}
```

Statements are the union of simple statements and compound statements.

5.5.2 Compound Statements

Because of the restriction of actions in DCharts models, compound statements cannot be directly written in the action list of the output of a transition. Suppose `comp_stm1`, `comp_stm2`, ... are compound statements, and `simp_stm1`, `simp_stm2`, ... are simple statements. The model in Figure 5.1 is invalid since the output of a transition is a list of compound statements instead of simple statements. On the contrary, the model in Figure 5.2 is valid.

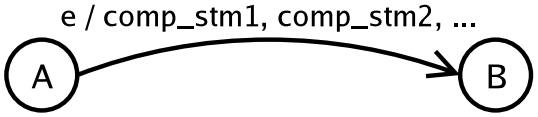

Figure 5.1: An invalid DCharts model that contains compound statements in the output

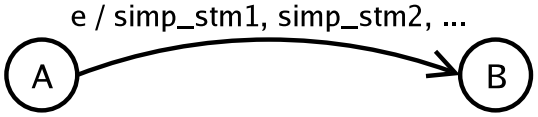

Figure 5.2: A DCharts model that contains simple statements in the output

It is important to transform a model with compound statements in its output into a valid form, since the statements discussed later are mostly compound statements. This transformation can always be found with the following method. (Since the composition of compound statements is still a compound statement, this method only consider transitions that have a single compound statement in their output.)

1. A GUID is assigned to each compound statement. It is assumed that there is no event with the same name as the GUID in the model. If there is an event whose name conflicts with a GUID, simply add an implementation-dependent prefix to all the GUIDs.
2. Create a top-level orthogonal component.
3. Suppose the compound statement contains the following substatements: stm_1, stm_2, ..., stm_n. There are such substates in the orthogonal component: s_0 (default state), s_1, s_2, ..., s_n.
4. Transition from s_0 to s_1 reacts to the GUID of the compound statement (as an event name) and executes stm_1 in the output. It generates a unique event e_1 with the parameters that it receives. The transition from s_1 to s_2 reacts to event e_1 and executes stm_2. It generates a unique event e_2 with the same parameters. Transition from s_2 to s_3 reacts to event e_2 and executes stm_3. It generates a unique event e_3 with the same parameters. ... Transition from s_{n-1} to s_n reacts to event e_{n-1} and executes stm_n. It generates a unique event e_n with the same parameters. Transition from s_n to s_0 reacts to event e_n and generates "return `GUID`" as an event.
5. For each transition t from *SRC* to *DES* with this compound statement as its λ, a new state s is added.
6. The original transition is replaced by two new transitions. The transition from *SRC* to s is the same as t, except that its *DES* is s, and its λ is an action that generates the GUID of the compound statement as an event. The parameters of this new transition become the parameters of the generated event. That event triggers the transition from s_0 to s_1 in the orthogonal component.
7. A transition is created from s to *DES* reacting to the "return `GUID`" event.
8. Repeat the above steps wherever a compound statement is found in the output, until all the output actions become simple statements or lists of simple statements (separated by a comma).

As an example, Figure 5.3 shows the transformation from a model with a compound statement in its output into a valid DCharts model. It assumes `comp_stm` to be a compound statement consisting of `stm1`, `stm2` and `stm3`, which may be simple statements or compound statements. The model in the upper part is converted into the model in the lower part. If `stm1`, `stm2` or `stm3` is not a simple statement or a list of simple statements, this transformation is repeated.

As a result of this transformation, the output of each transition becomes a simple statement or a list of simple statements. (Note that this transformation may be done with graph grammars [33].)

Sequential execution of substatements in the compound statement is guaranteed by this transformation.

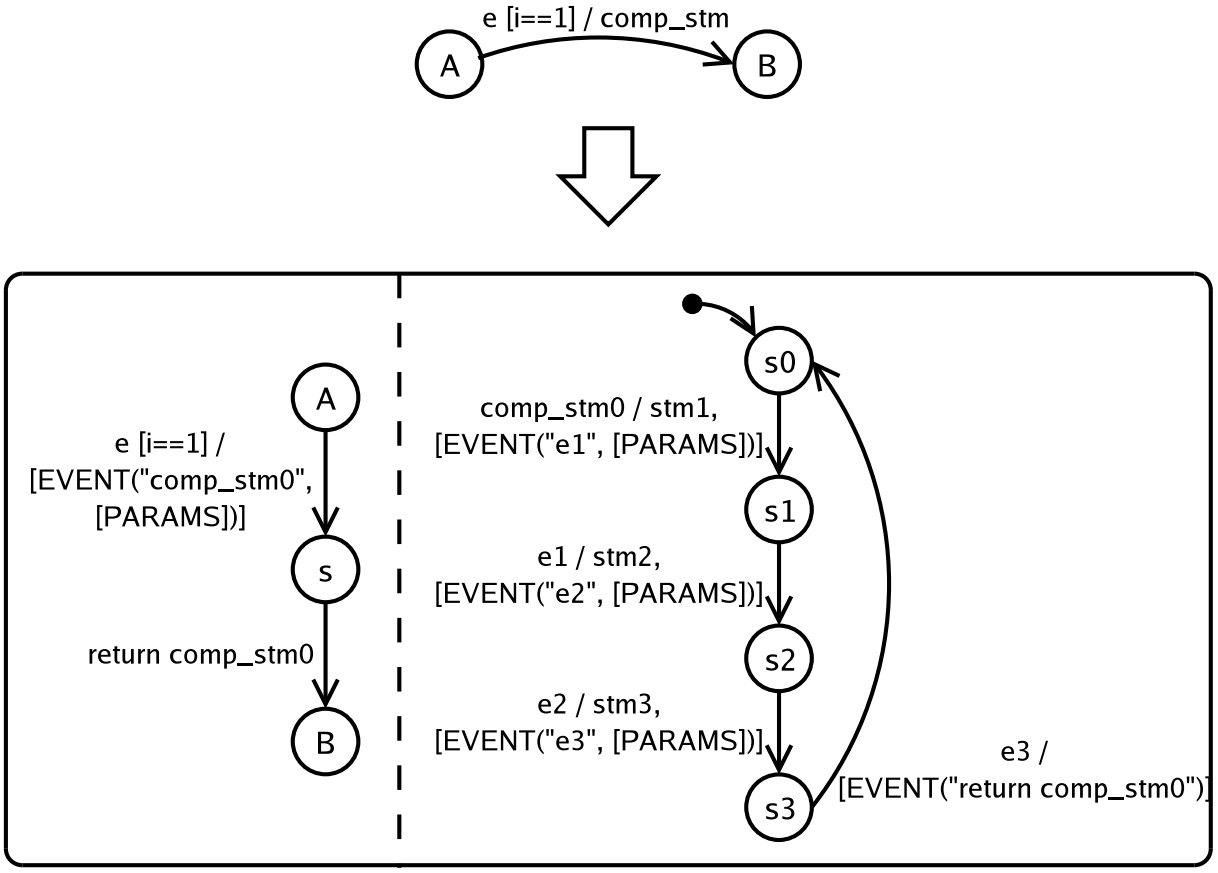

Figure 5.3: An example of the transformation from a compound statement in the output into simple statements

However, synchronization is lost. Actions in other orthogonal components may be executed during the execution of those substatements, and the execution result of those interleaving actions becomes unpredictable. This semantics is different from executing a compound statement in a critical session provided by the simulator or executor. To solve this problem, it is suggested that the simulator or executor provide actions that allow designers to control the critical sessions. If such actions are available, the orthogonal component generated by this transformation is explicitly placed in a critical session. All the statements in it are executed continuously without interleaving with other actions. This topic is out of the scope of this thesis.

5.5.3 Conditional Statements

If-else statements and switch statements are two kinds of conditional statements.

```
if (i == 0) {  // an if-else statement
    ...
} else if (i == 1) {
    ...
} else if (i == 2) {
    ...
} else {
    ...
}

switch (i) {  // a switch statement
    case 0: ...
            break;
    case 1: ...
            break;
    case 2: ...
            break;
    default: ...
}
```

Switch statements are actually nested if-else statements. Each case within a switch statement corresponds to a condition in the if-else statement. If-else statements are more powerful than switch statements, since the conditions of if-else statements are C expressions, while the cases in switch statements must be constants.

If-else statements can be easily modeled in DCharts. The guards in the transitions test the different cases, and the outputs perform the actions that correspond to those cases.

Figure 5.4 depicts an example of the transformation from an if-else conditional statement into guards of multiple transitions. Suppose `cond_stm` is such an if-else statement:

```
if (x == 0)
    stm1;
else if (x == 1)
    stm2;
else if (y == 0)
    stm3;
else if (y == 1)
    stm4;
else
    stm5;
```

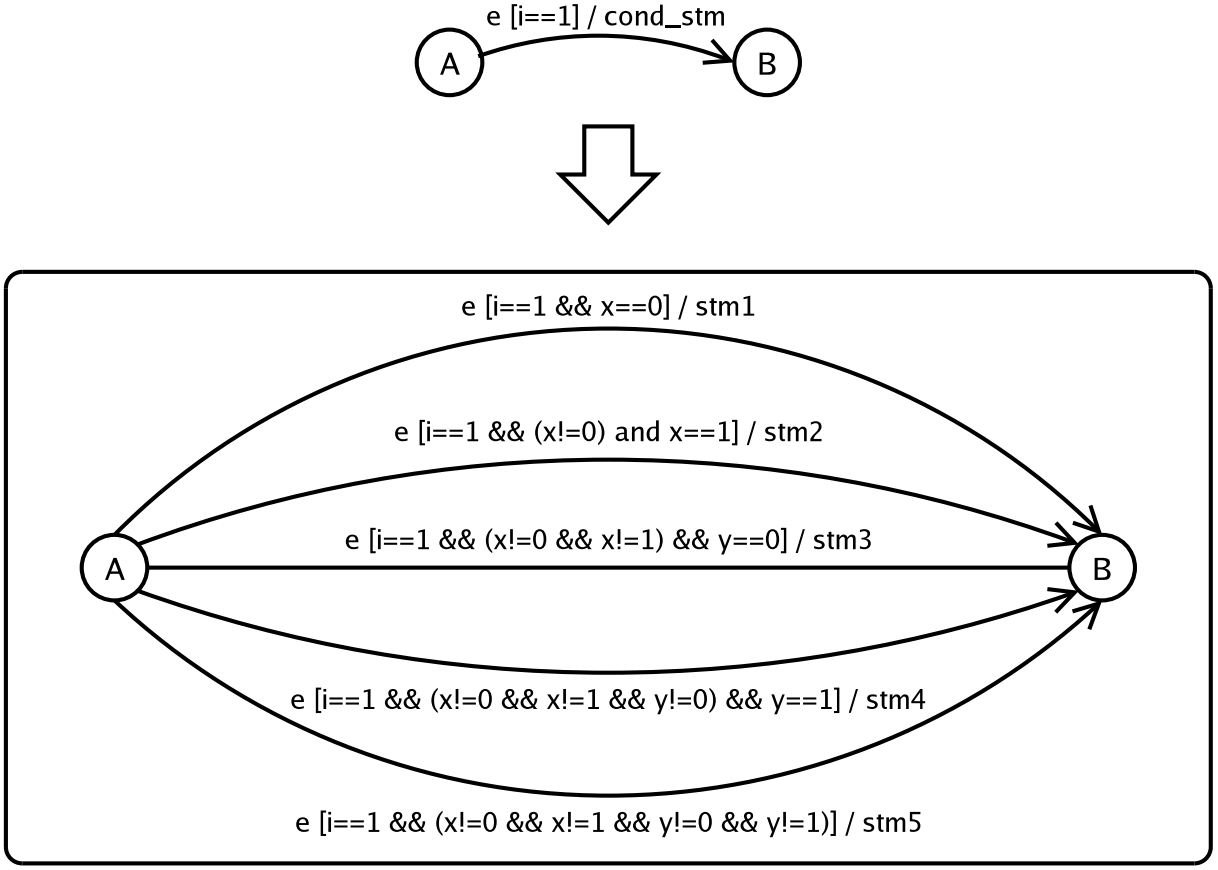

Figure 5.4: An example of the transformation from a conditional statement into guards

In the upper part, the model has a transition from state `A` to `B` reacting to event `e`. The transition is enabled only if the guard `i==1` is satisfied. It executes `cond_stm` as an output. This is not a valid DCharts model, since it violates the restriction of the action code. It is transformed into the valid DCharts model in the lower part. A transition is created for each condition in the if-else statement. The test cases of the conditions are added to the guards. For example, the first test case is `x==0`. It is added to the guard of the first transition. The second test case is `x==1` on the basis that the first test case is not satisfied. As a result `(x!=0) && x==1` is added to the guard of the second transition. The third test case is `y==0` is on the basis that neither the first test case nor the second test case is satisfied. As a result `(x!=0 && x!=1) && y==0` is added to the guard of the third transition. And so on.

If any statement in `stm1` to `stm5` is not a simple statement or a list of simple statements, further transform the model with the algorithm in section 5.5.2. If the model has other conditional statements, or `stm1` to `stm5` contain conditional statements, transform those conditional statements with the same method.

5.5.4 Loops

There are several kinds of loops in the C language:

- For-loop.

  ```
  for (init; cond; step)
      stm;
  ```

  Here, *init* is a statement (or a list of statements) to be executed before the for-loop. *cond* is a boolean expression that must be satisfied before each iteration. The for-loop stops when *cond* is evaluated to $false$. *step* is a statement (or a list of statements) to be executed after each iteration. *stm* is a statement to be executed in each iteration. It may be a compound statement enclosed by a pair of curly braces.

  For-loops can be transformed into statements with an if-else condition. The above for-loop structure is transformed into:

  ```
  init;
  loop_label:
  if (cond) {
      stm;
      step;
      goto loop_label;
  }
  ```

- While-loop.

  ```
  while (cond)
      stm;
  ```

  It can be transformed into a for-loop:

  ```
  for (; cond; )
      stm;
  ```

- Do-while-loop.

```
do
    stm;
while (cond)
```

It can be transformed into a for-loop with one extra execution of the statement *stm*:

```
stm;
for (; cond; )
    stm;
```

Since other types of loops can be simulated with for-loops, it is enough to show that for-loops can be modeled with DCharts. As shown above, for-loop can be transformed into:

```
init;
loop_label:
if (cond) {
    stm;
    step;
    goto loop_label;
}
```

Suppose *v* is a temporary boolean variable. This code is equivalent to the following piece of code (denote it with `comp_stm`). It brings the "goto" statement out of the conditional construct.

```
init;
loop_label:
v = cond; // evaluate cond and store the result in v
if (v) {
    stm;
    step;
}
if (v)
    goto loop_label;
```

A model with a transition from state `A` to `B` that has the above action code in the output (the upper part of Figure 5.5) is transformed into the model in the lower part of Figure 5.5. If compound statements are still found in the output of the generated transitions, further transform the model into valid DCharts models with the method in previous sections.

This proves that all kinds of loops can be modeled with DCharts.

This transformation does not take into account synchronization among actions in different orthogonal components either.

5.5.5 Break and Continue

The `break` statement and the `continue` statement in a loop can be transformed into extra transitions.

Suppose statement `comp_stm` is such a compound statement:

```
init;
loop_label:
```

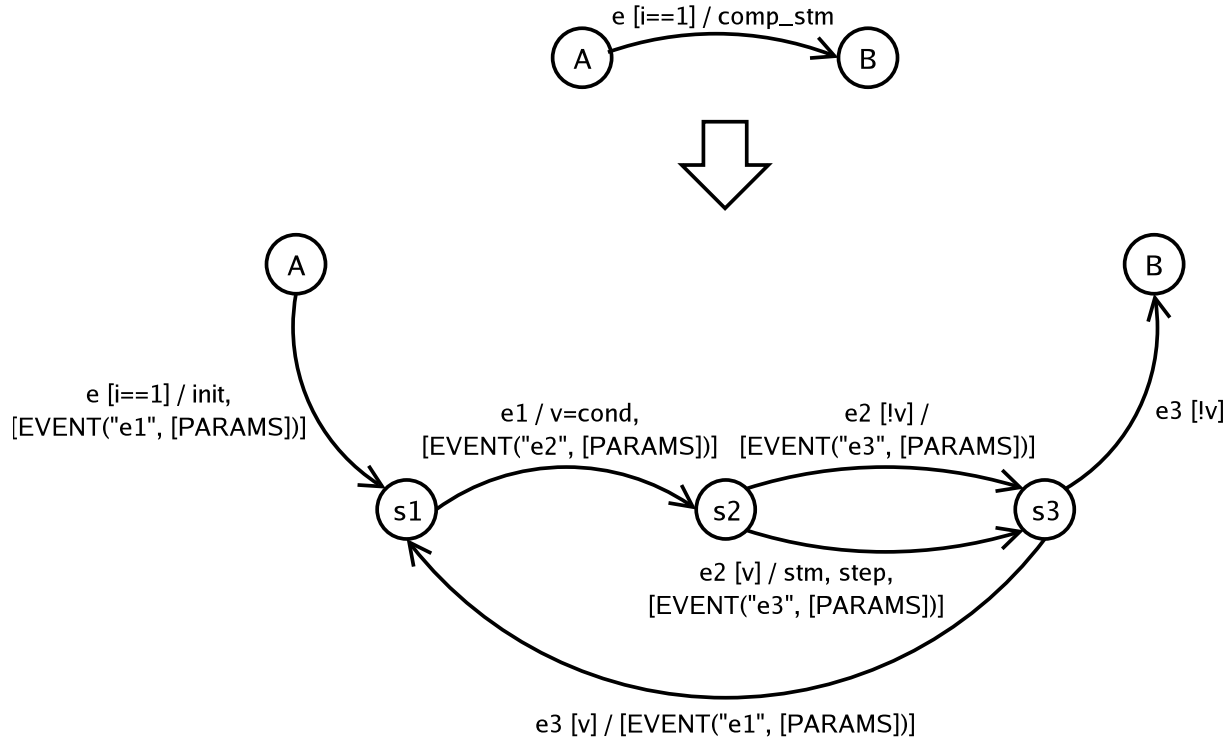

Figure 5.5: An example of the transformation from a for-loop into multiple transitions

```
if (cond) {
    stm1;
    if (finished)
        break;
    stm2;
    step;
    goto loop_label;
}
```

The `break` statement stops the for-loop by changing the execution point out of the compound statement. It is equivalent to:

```
init;
loop_label1:
v = cond; // evaluate cond and store the result in v
if (v) {
    stm1;
    if (finished)
        goto loop_label2;
    stm2;
    step;
}
if (v)
    goto loop_label1;
loop_label2:
```

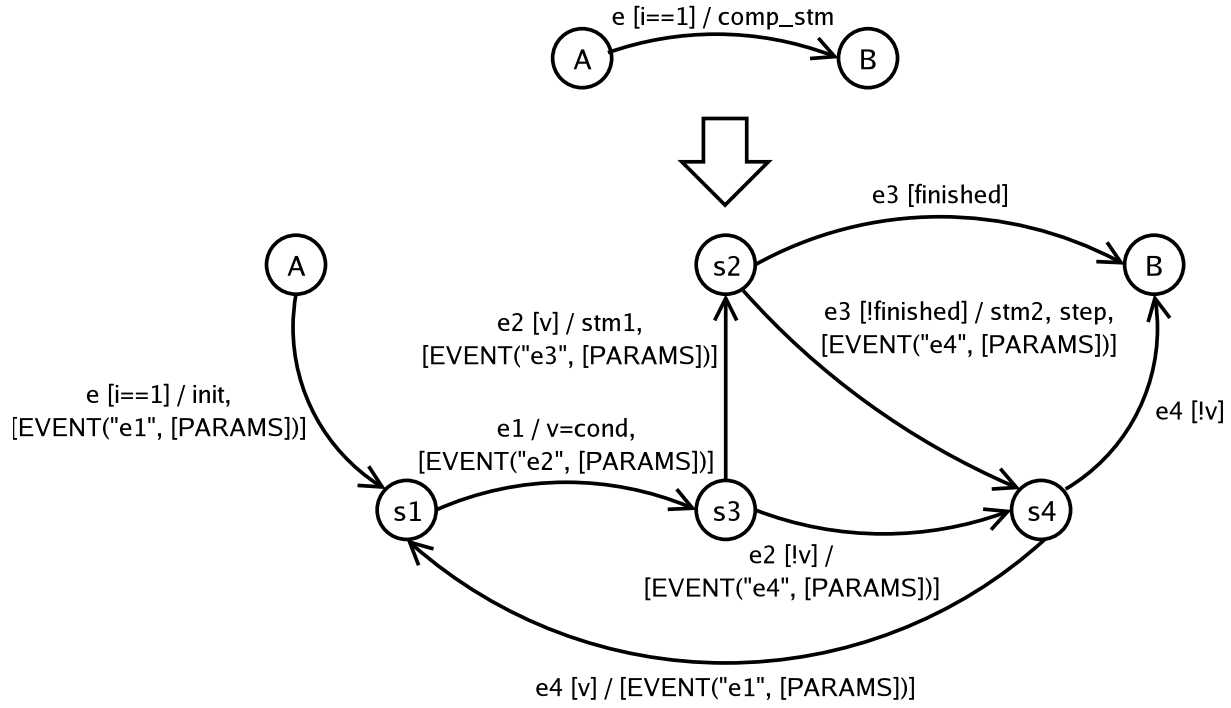

Figure 5.6: An example of the transformation from a `break` statement into DCharts transitions

The transformation of the model is illustrated in Figure 5.6. The `break` statement in the for-loop is eliminated in this example.

The `continue` statement in a loop can be eliminated in a similar way. If the `break` statement in the above `comp_stm` is replaced by the `continue` statement, it is equivalent to:

```
init;
loop_label:
v = cond; // evaluate cond and store the result in v
if (v) {
    stm1;
    if (finished)
        goto loop_label;
    stm2;
    step;
}
if (v)
    goto loop_label;
```

The transformation of this code with the `continue` statement is shown in Figure 5.7.

5.5.6 Tricks of Actions Specific to SVM

This section discusses the tricks of action code in SVM. Although it is forbidden to write arbitrary code in the output of transitions or enter/actions, the tricks discussed below still allow designers to write native code in a specific language. These tricks are specific to SVM. They are not in the DCharts 1.0 definition, and hence they are not portable.

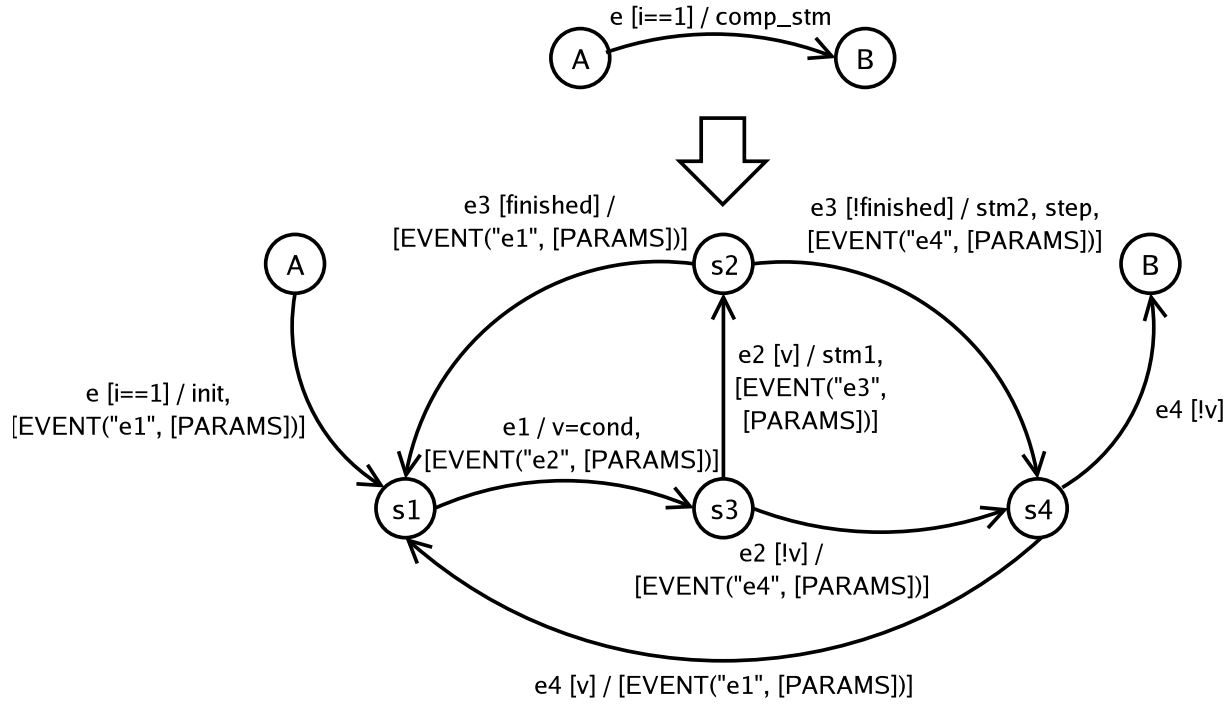

Figure 5.7: An example of the transformation from a `continue` statement into DCharts transitions

Python Native Libraries

SVM is completely implemented in the Python language [34] [35] [36]. It is possible to import libraries in the action code of a model. Those libraries can be Python standard modules [37] or user-defined libraries. For example, the following piece of code defines several functions in a library (saved in file `lib.py`):

```
def func1():
    ...
def func2(a, b, c):
    ...
import sys
def func3(x, y=0):
    ...
```

To import this library into an SVM model, include action `import lib` in its initializer, and make sure that `lib.py` is in the same path as the model or can be found in the `PYTHONPATH` environment variable. Hence, the functions defined in the library can be directly used in the action code.

The designers, if they implement part of the system with user-defined libraries, must decide what is to be implemented with the native libraries and what is to be modeled with the DCharts formalism. This usually raises a dilemma: implementation in the library is straightforward (for programmers) and efficient, while modeling explicitly with DCharts is formal and the benefits of modeling (model checking, analysis, transformation, simulation and code generation) are gained. As a general suggestion, a system is usually divided into three parts: user interface, control logic and hardware driver (Figure 5.8). The user interface is usually hard-coded in a library, since it contains the detail of the rendering of various widgets and their interaction with the users. The hardware driver is usually hard-coded in a library, too. This is because it deals with the detail of hardware control, threading, synchronization, interrupt, status polling, and so on. Only the control logic is explicitly modeled with DCharts. It is usually the most vulnerable part of a system. Tools

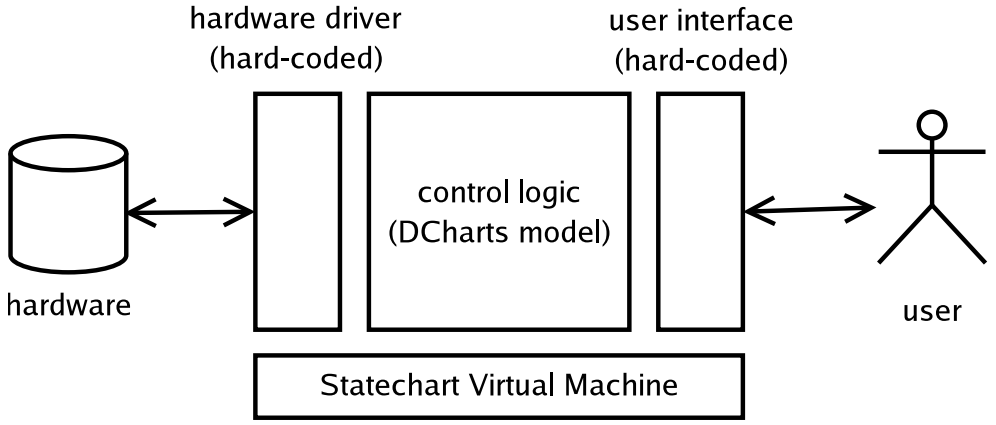

Figure 5.8: The three parts of a system

should be used as much as possible to thoroughly test and simulate the control logic before it is considered stable.

Though the separation of the three parts is far from deterministic or unique, there are some rules to be followed:

- The user interface should never interact with the hardware driver directly. It only sends user events to the DCharts model, and the DCharts model consumes those events. (In SVM, to send an event to the DCharts model, the user interface library must call function `eventhandler.event(ev, p)`, where `ev` is the event name and `p` is a parameter of any Python type.)
- The hardware driver should never interact with the user interface directly. It only generates hardware-specific events with the same `eventhandler.event` function. Those events are also consumed by the DCharts model.
- The API (Application Programming Interface) of the hardware driver should be generalized for various applications. The designer should not intentionally tune it in order to simplify a specific DCharts model.
- The designer is allowed to use yet another library to define functions that are considered primitive in the system. For example, sorting, management of data structures and well-know algorithms should be hard-coded in a library rather than being modeled explicitly. The latter approach only unnecessarily complicates the model and obscures the essence of the problem to be studied.

Function Definition in SVM Models

As another trick, it is also possible to directly define functions in SVM models, though this method is highly discouraged because of its lack of modularity and portability. The initializer of a model is among the parts that allow arbitrary Python code. A model designer may decide to implement some of the functions in the initializer of a model. Because Python is an interpreted language, SVM is able to dynamically interpret the definition of those functions, and make them available for the actions executed later.

For example, the textual representation of the model in Table 5.1 defines a function `print_a_to_b`, which prints integers from `a` to `b` to the console on a single line. (`a` and `b` are integer parameters of the function.) The model calls this function with `a=1` and `b=10` every 1 second with a timed transition. As a result, the user of the model gets the following output in the console:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

```
STATECHART:
    A [DS]

INITIALIZER:
    def print_a_to_b(a, b):
        while a<=b:
            print a,
            a = a + 1
        print

TRANSITION:
    S: A
    N: A
    T: 1
    O: print_a_to_b(1, 10)
```

Table 5.1: An example of the textual representation of a function definition in a DCharts model

```
1 2 3 4 5 6 7 8 9 10
...
```

A whole Python library can be written under the `INITIALIZER` descriptor. However, the more Python code is written here, the harder it is to port the model to another simulator or executor, and the harder it becomes to fully understand the meaning of the model. As the size of the model grows, it becomes less manageable. The chances of undetectable errors increase dramatically.

5.6 Conclusion

The expressive power of DCharts is formally shown in this chapter. It is proved that non-recursive DCharts are at most as powerful as statecharts with variables (section 5.1), and non-recursive (the theorem in section 5.3 does not use recursive importation) DCharts are at least as powerful as statecharts with variables. From these results, it can be inferred that non-recursive DCharts are equivalent to statecharts with variables in terms of expressiveness. Similarly, sections 5.2 and section 5.4 prove that non-recursive DCharts are equivalent to DEVS in terms of expressiveness.

Obviously, DCharts are more powerful than statecharts with variables and DEVS. Recursive importation and the parametrized importation introduced by SVM cannot be modeled with non-recursive DCharts. [1] Because of the equivalence of expressiveness, recursive importation and parametrized importation cannot be modeled with statecharts with variables or DEVS, either. The following inequation shows the comparison of expressiveness of the above-mentioned formalisms:

$$DCharts > non\text{-}recursive\ DCharts = statecharts\ with\ variables = DEVS$$

The expressiveness of DCharts is further shown by using them to model the constructs in the C programming language. This proves that DCharts are able to model a complete system in place of modern programming languages such as C. The explicitly modeled parts of a system can be formally checked, analyzed, optimized and simulated with DCharts modeling and simulation tools. Code can be generated from the well-developed parts for efficiency. This development process strongly emphasizes the use of automated tools and saves human labor.

[1] The SVM simulator itself can be modeled with non-recursive DCharts. In this sense, the execution of recursive DCharts can be modeled with non-recursive DCharts. This issue is highly implementation-oriented. It is not considered here.

The examples in section 5.5 illustrate several *design patterns* [38]. Those patterns point out a way in which C constructs can be transformed into DCharts submodels. Designers may model those patterns in separate submodels, and import them into their systems. Tools can also be implemented for this transformation.

For a relatively large system, there may be a lot of those design patterns. Optimization tools and code generation tools may reverse them. They locate every appearance of known patterns, and transform it into equivalent but much simplified C code. This code generation produces much more efficient code than the classes generated in a normal way, which manage all the states and transitions in those patterns.

6

SVM – A DCHARTS SIMULATOR

A valid DCharts model contains all the necessary information for a simulation. SVM (Statechart Virtual Machine) is the simulation environment that runs textual DCharts models.

6.1 An Introduction to SVM

SVM is originally a statecharts simulator implemented in Python (`http://www.python.org`), but now it supports the complete DCharts semantics and the textual syntax, including the syntactic extensions.

SVM is a project developed in the MSDL (Modeling, Simulation and Design Lab) of SOCS (School of Computer Science) of McGill University. The lab is headed by Prof. Hans Vangheluwe.

SVM has multiple sub-projects. One of its sub-projects, SCC (StateChart Compiler), aims at a tool that synthesizes source code from DCharts models. Multiple target-languages are supported. This code synthesizer is discussed in chapter 8.

SVM and its sub-projects are provided for public use under the terms of GNU GPL (General Public License) version 2. There is absolutely no warranty for these tools. The text of the license can be obtained from:

`http://www.gnu.org/licenses/gpl.html`

The SVM homepage is at:

`http://msdl.cs.mcgill.ca/people/tfeng/?research=svm`

All the necessary information for obtaining and installing SVM and SCC can be found at the homepage. In particular, a tutorial on SVM and SCC, which contains the installation instructions and several interesting examples, is available [39].

6.2 The Design of SVM

The class design of SVM is shown in Figure 6.1. (This class diagram only shows the important attributes and methods.) Class `EventHandler` is the main class that loads the model from a text file, builds internal data structures for the model, and simulates the model on demand. It can be used with different user interfaces: the `TextualInterface` class defines the default textual interface that accepts input and produces output on the console; the `GraphicalInterface` class defines the default graphical interface; and the `CursesInterface` class defines the default curses interface (for UNIX only) to be used in the text mode with colors. Designers may define model-specific interfaces. Examples of model-specific interfaces are discussed in later chapters.

Class `SVMFrontEnd` provides a front end of the simulation environment for the end users. It accepts command-line parameters and initializes an instance of `EventHandler` with the model description. It also interacts with the model user through one of the user interfaces.

Class `Generator` uses `EventHandler` to parse DCharts models. It generates source code in different target-languages from the internal structures created by `EventHandler`. Class `SCCFrontEnd` provides a command-line front end for the code generator.

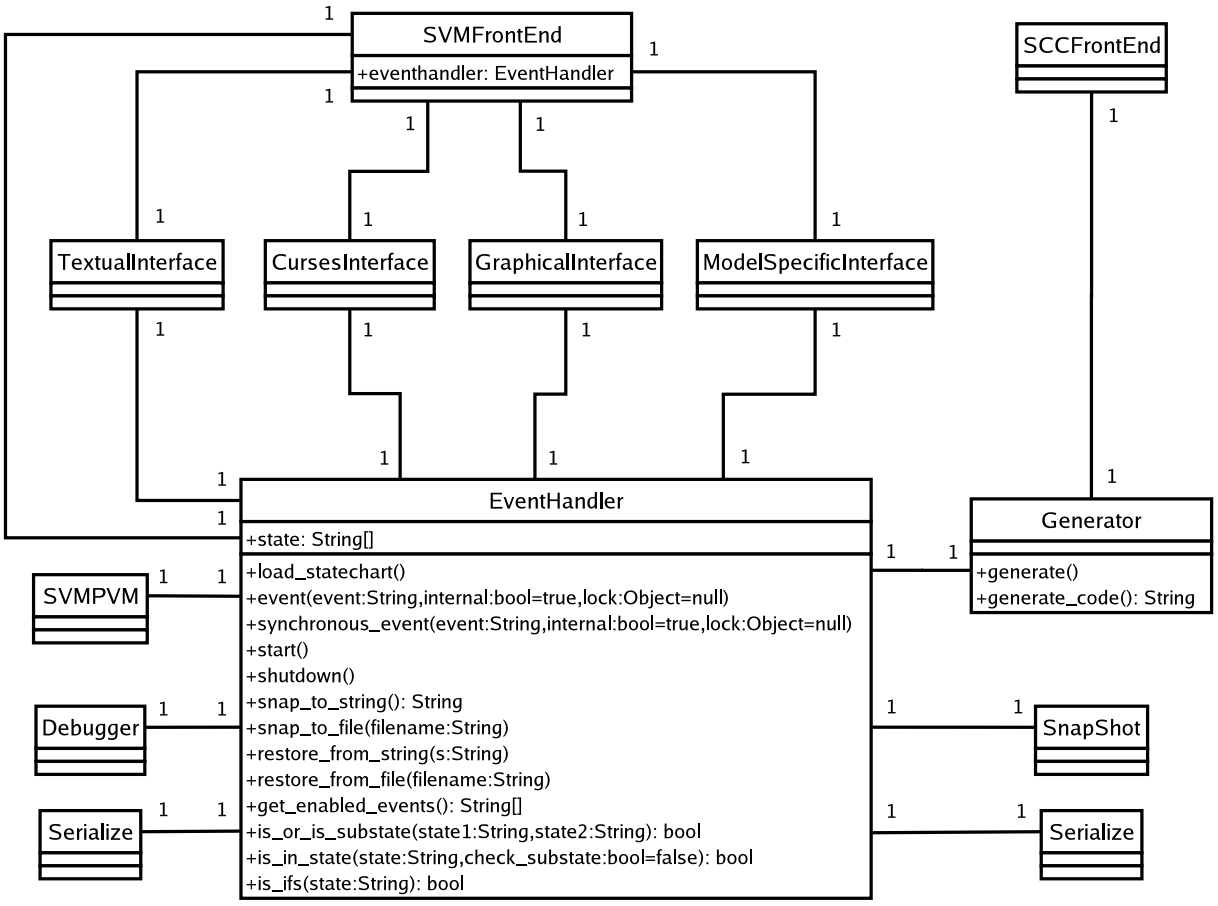

Figure 6.1: SVM class design

As `EventHandler` is the core of the parser and the simulator, it is necessary to introduce some of its methods and attributes here:

- `event(event, params, internal, lock)`

  Appends an event to the global event list. The event will be handled after all the events before it in the list are consumed. Parameter `event` is the event name. `params` is a parameter or a Python list of parameters for the event. `internal` is a boolean that denotes whether the event is generated by the model itself or is received as a message from a port. `lock` is a semaphore. If it is non-null, the simulator will release this lock once this event is handled. The following piece of code uses a semaphore to schedule an event and waits until it is handled:

  ```
  ...
  import thread  // import the Python threading library
  lock = thread.allocate_lock()  // allocate a lock
  lock.acquire()  // acquire the lock for the first time
  eventhandler.event("e", [], 1, lock)  // raise event
  lock.acquire()  // acquire the lock again; block until the event is handled
  del lock  // destroy the lock
  ...
  ```

- `synchronous_event(event, params, internal, lock)`

  Raises an event and waits until the event is handled (as the code segment above). Its parameters have exactly the same meaning as the `event` method.

- `start()`

  Starts the simulation. It executes the initializer of the model and places the model in its initial default state.

- `shutdown()`

  Ends the simulation. If the model is not in a final state (the finalizer has not been executed yet), the finalizer is executed; otherwise (the finalizer has been executed), the SVM simulator simply exits.

- `snap_to_string():String`

  Takes a snapshot of the current state of the model. The snapshot is returned as a string.

- `snap_to_file(filename)`

  Takes a snapshot and save the snapshot in the file named by the `filename` parameter. The file is a plain text file. The user may manually edit it. (Note: it is the modifier's responsibility to make sure that the file is still meaningful.)

- `restore_from_string(s)`

  Restores a previously taken snapshot (saved in a string) and resumes the simulation. Information about the current simulation is completely lost.

- `restore_from_file(filename)`

  Restores a previously taken snapshot stored in the file named by the `filename` parameter.

- `get_enabled_events()`

  Returns a list of the names of enabled events. The result depends on the current state of the simulation.

- `is_or_is_substate(state1, state2):bool`

  Returns `true` if `state1` is equal to `state2` or `state1` is a substate of `state2`.

- `is_in_state(state, check_substate):bool`

  Returns whether the model is currently in `state`. If `check_substate` is `false`, the simulator only checks leaf states. Hence, the result is `true` if and only if the model is in `state` and `state` is a leaf

state. If `check_substate` is `true`, the function returns `true` if and only if the model is in `state`, whether it is a leaf state or not.

- `is_ifs(state):bool`

  Returns whether `state` is inner-transition-first.

- Attribute `state`

  A list of strings enumerating all the leaf states that the model is currently in.

To simulate DCharts models, `EventHandler` requires the support of other classes. These classes are not necessary for code generation in SCC:

- Class `SVMPVM` is an interface to PVM (Parallel Virtual Machine) for distributed simulation in SVM.
- Class `Debugger` provides the functions for model debugging. It allows the testers to define callback functions that are invoked when certain criteria are satisfied during a simulation. Those callback functions are similar to the breakpoints of modern IDEs (Integrated Development Environments).
- Class `Serialize` provides serialization facilities for SVM. With this class, the global `eventhandler` object can be serialized as a string that contains all the information needed to reconstruct the object.
- Class `SnapShot` makes use of the functions provided by class `Serialize` and provides snapshotting facilities for SVM.

The `EventHandler` class is a parser and a simulator. It can be reused in other applications. For example, the SCC code synthesizer uses this class to parse textual model descriptions; an application that needs a DCharts simulator (such as AToM^3 with the DCharts plugin) may use it to simulate models.

The command-line to invoke SVM is discussed in [39]. It includes a complete description on how to start the simulation of a model, how to choose among the default interfaces, and how to redefine macros for the model.

6.3 Default Interfaces

This section discusses the default graphical interface and the default textual interface. The default curses interface is similar to the default textual interface.

6.3.1 Default Graphical Interface

Figure 6.2 shows the default graphical interface. The window on the right is the main window. The enabled events are displayed in the "Enabled Events" list. This list is refreshed whenever the state of the model changes. The "Output" box displays output from the model or the commands entered by the user. The model sends output to this box with the `DUMP` macro. Model description (if defined) is also displayed in this box at the time a model is loaded or imported. The "Command" box accepts commands from the user. Three kinds of commands are accepted:

- Events. To raise an event, the user may enter the event name in the "Command" box and press `ENTER`, or double-click the event name in the "Enabled Events" list.
- "`debug`". The user may enter a special event "`debug`" to change to the debug mode.
- "`exit`". This special event terminates the simulator and closes the SVM windows. It has the same effect as pressing the "Exit" button in the main window.

Any other command not recognized by SVM is simply ignored.

By pressing the "Snapshot" button, the user takes a snapshot of the current state of the model. The snapshot is saved to a `.snp` file with the same name (excluding the postfix) as the file name of the model description.

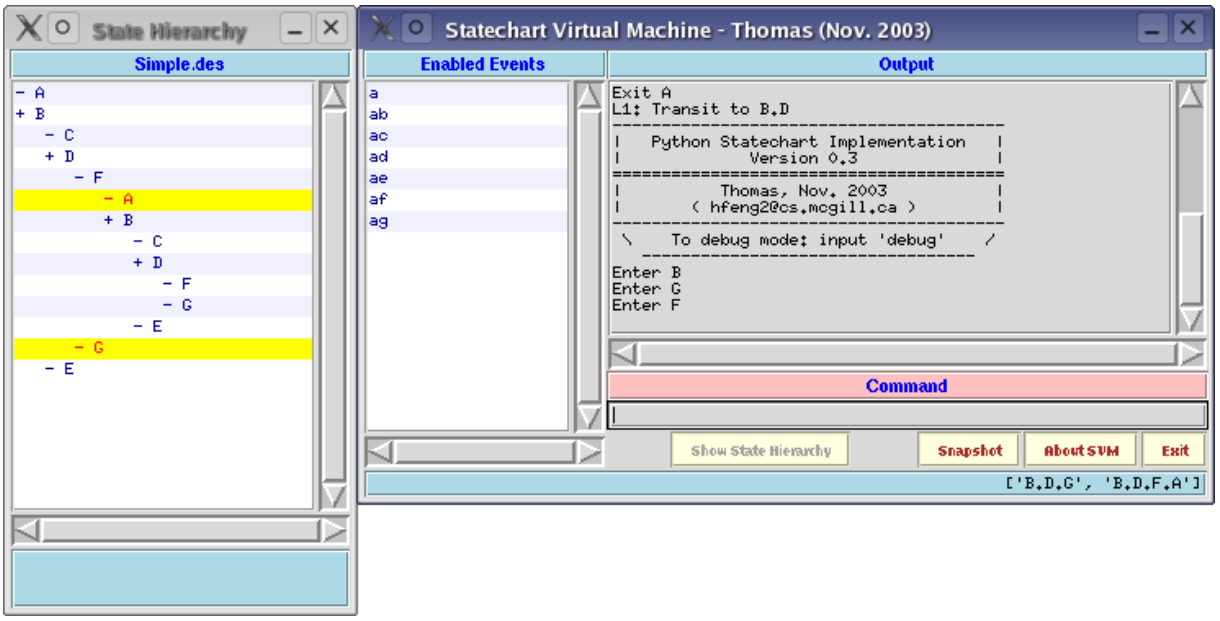

Figure 6.2: SVM default graphical interface

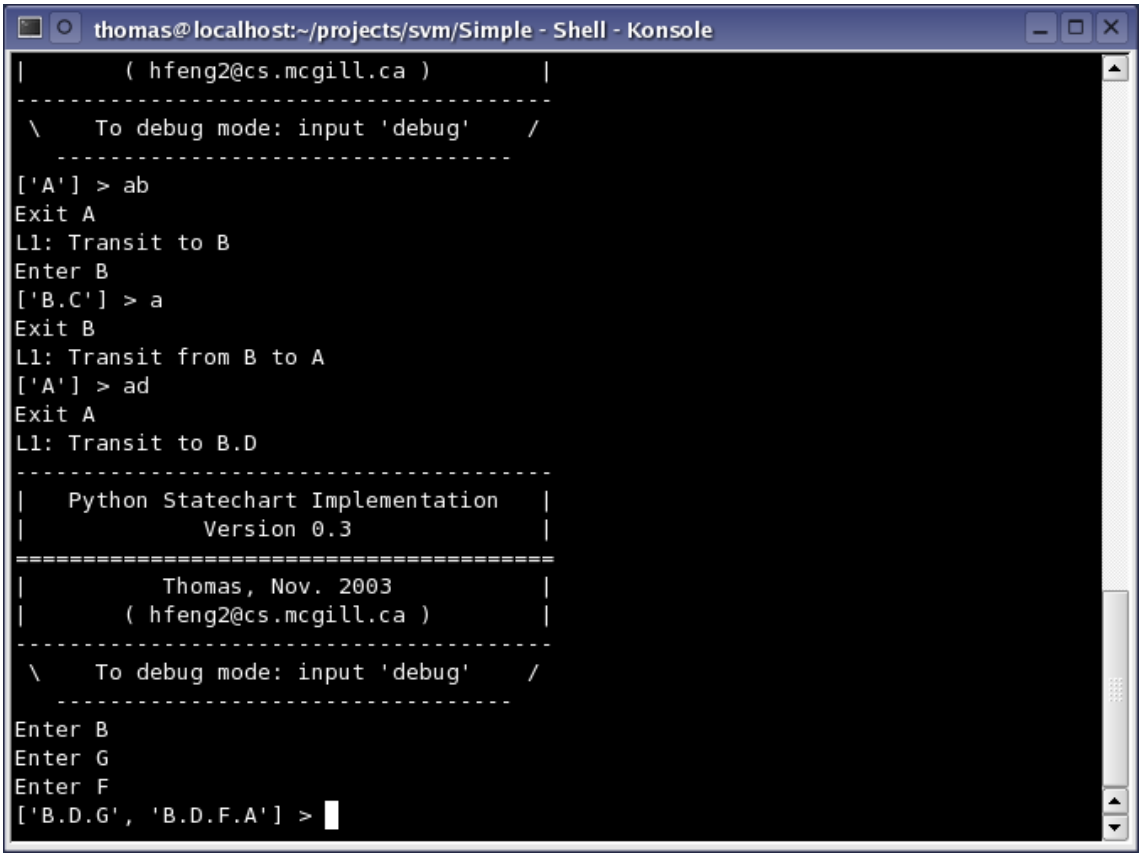

Figure 6.3: SVM default textual interface

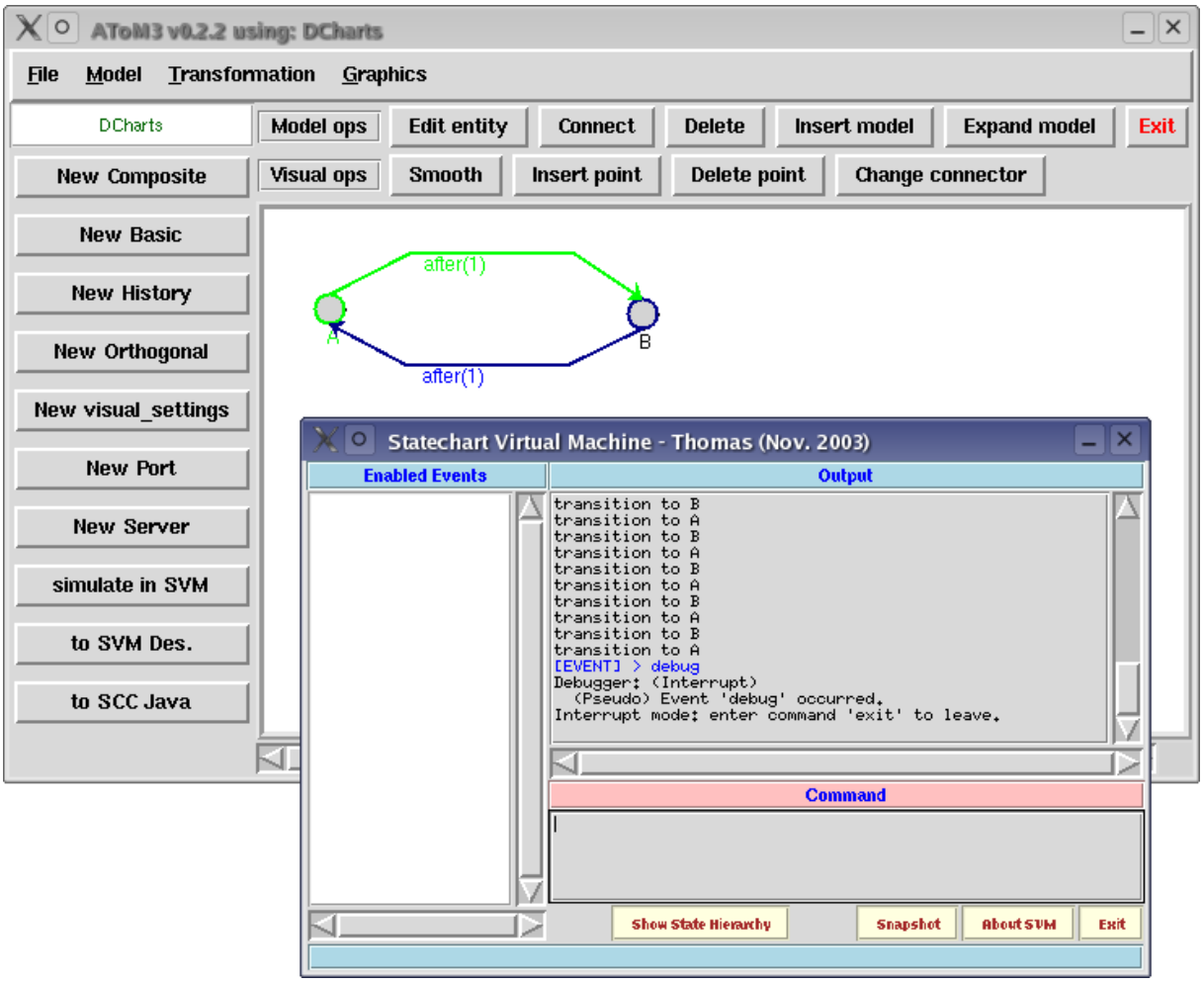

Figure 6.4: AToM^3 modeling environment with SVM plugin

6.3.2 Default Textual Interface

As opposed to the graphical interface, textual interface is suitable for most systems, even if they do not have any graphical device or they are too slow to support Tkinter (the graphical widget library for Python). The default textual interface is shown in Figure 6.3. The state of the model is printed before the prompt. Similar to the graphical interface, the user is allowed to enter event names and the "`debug`" and "`exit`" special events. The `DUMP` macro prints messages to the console.

6.4 Modeling and Simulating DCharts in AToM^3

SVM is a stand-alone simulator that does not depend on any other modeling and simulation tool. However, it can be seamlessly integrated with AToM^3. A plugin for AToM^3 generates DCharts model descriptions from its graphical representation in AToM^3. The user may then save the descriptions in text files to be simulated by SVM. Alternatively, the generated model descriptions can also be stored in memory and be simulated by SVM immediately without being saved. In the latter case, SVM highlights the current states and enabled transitions in AToM^3 during the simulation (Figure 6.4).

The SVM plugin adds a DCharts meta-model to AToM^3. It is developed on the basis of Spencer Borland's statecharts meta-model for AToM^3 [9]. Three buttons are available to simulate the model in the current canvas immediately, generate `.des` model description to a text file, and generate Java source code from the current model with SCC (discussed later). The designer is thus able to design the model in the AToM^3 visual

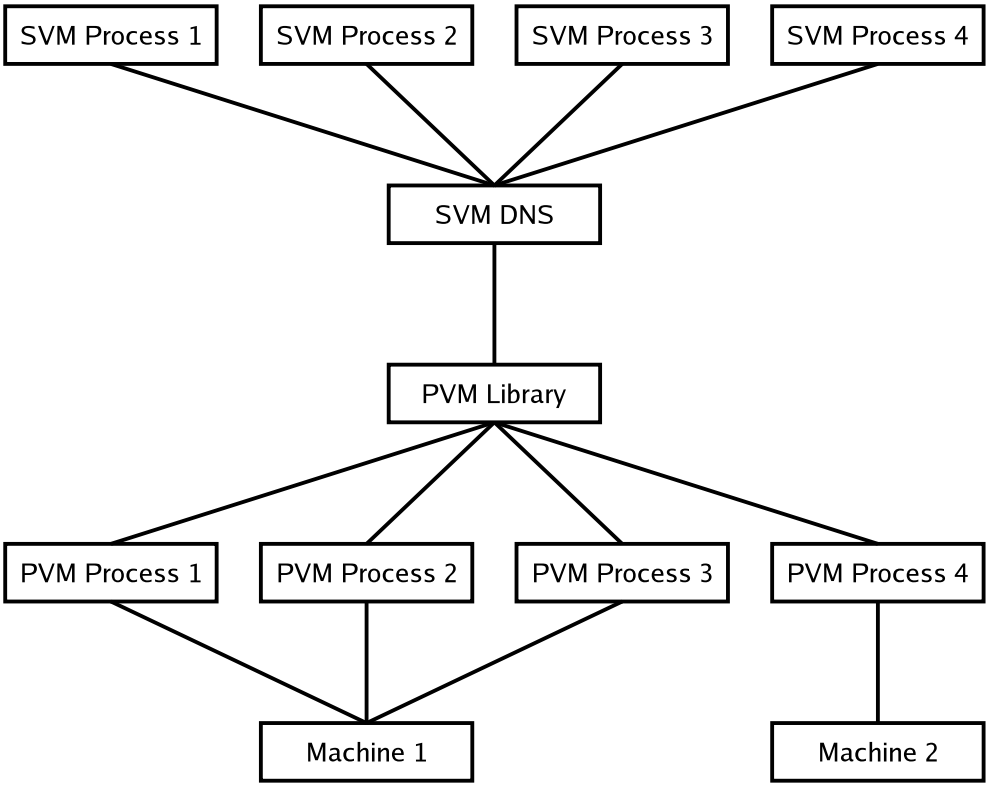

Figure 6.5: Multiple layers for distributed simulation in SVM

environment, and access to these functions simply by clicking on the corresponding buttons.

6.5 Distributed Simulation

SVM supports distributed simulation with PVM (Parallel Virtual Machine) [10]. A *distributed model* is divided into several components conceptually running on multiple machines. PVM hides the configuration of those machines. Each PVM process is regarded as a conceptual machine that has its unique ID and is able to communicate with other PVM processes. Multiple PVM processes may run on the same machine. Multiple machines may be involved in a distributed simulation enabled by PVM, after they are added to the PVM daemon.

6.5.1 The SVMDNS daemon

SVMDNS (SVM Dynamic Naming Service) is another daemon built on top of the PVM library. It provides a higher level of interface to SVM processes. For example, in Figure 6.5 there are 4 SVM processes, each of which has a DCharts component running on it. Those DCharts components communicate with each other via ports. The SVM processes register themselves to a single SVMDNS daemon. The SVMDNS daemon invokes functions in the PVM library to create 4 PVM processes. Each of them corresponds to an SVM process. The location of those PVM processes depends on the configuration of the PVM daemon. In this case, PVM processes 1, 2 and 3 are located on machine 1, while PVM process 4 is located on machine 2. The PVM library hides details of this configuration, but provides a uniform API to SVMDNS.

SVMDNS provides the following functionality to each SVM process:

- Registration. Each SVM process that interacts with remote components must register itself to SVMDNS. By default, the SVM simulator attempts to register itself to SVMDNS if and only if a model with at least one port is running in it.

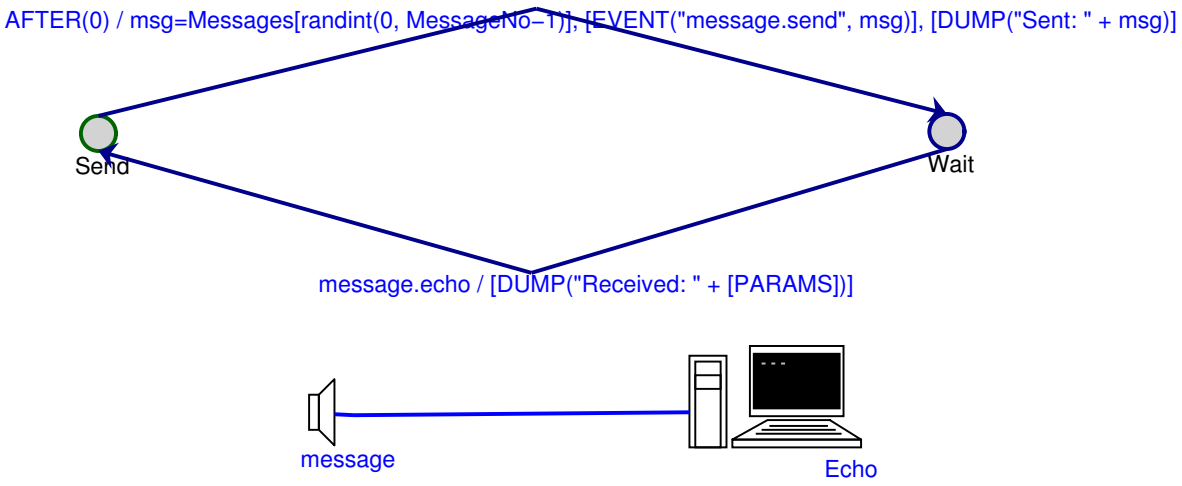

Figure 6.6: `Sender` of the `Echo` example

- Life-time. Each SVM process registered to SVMDNS must periodically sends a keep-alive message to the SVM daemon. If the daemon does not receive such a message from an SVM process within a certain period of time (known as *life-time*), information about the SVM process is removed from SVMDNS' registry. The life-time can be customized in `PVMUtil.py`. By default it is 30 seconds. Each SVM simulator, after it registers itself, sends the keep-alive message to the SVMDNS every half life-time period.
- Component lookup. SVM processes send the name patterns or types of required components to the SVMDNS. SVMDNS then locates the registered components with those name patterns and types. It establishes the connections between components.
- SVMDNS also maintains the connections between components. SVM processes are ignorant of this information. They simply use ports to identify groups of connected components in SVMDNS. SVMDNS acts as a router in this inter-component communication.

Detailed information about the setup of the SVM daemon and the PVM daemon can be found in [39].

6.5.2 Example

A simple `Echo` example is studied in the section. There are two components in the system: `Sender` and `Echo`. The `Sender` randomly generates a message and sends it to the `message` port of the `Echo`. The `Echo` sends back this message to the `Sender` after 1 second. When the `Sender` receives the message, it sends another random message to the `Echo`. This loop continues forever.

These components are designed in AToM^3 as shown in Figure 6.6 and Figure 6.7. In Figure 6.7, an input/output port named `message` is defined for the `Echo` component. The `Sender` component in Figure 6.6 also has a port called `message`. The port of the `Sender` is connected to the port of the `Echo`. The name pattern of the server is `Echo` (Figure 6.8). This matches the `Echo` component only. The link between the `Sender` port and the server has a property that specifies the server port `message` (Figure 6.9). The enter actions of the `Send` state of the `Sender` component is hidden. Those actions import necessary Python libraries and initialize a list of random messages.

When the `Sender` component is loaded into AToM^3, the user may press the "to SVM Des." button to generate a `.des` file. Here is the `Sender.des` generated by the SVM plugin:

```
# DCharts description generated by SVM—AToM3—plugin, written by Thomas Feng
#  Source: /home/thomas/Backup/Atom3_2.2/DCharts/models/SimpleEcho/Sender.py
#  Date: January 15, 2004
```

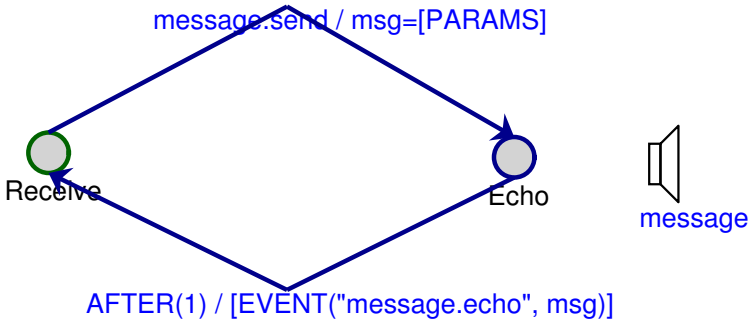

Figure 6.7: `Echo` of the `Echo` example

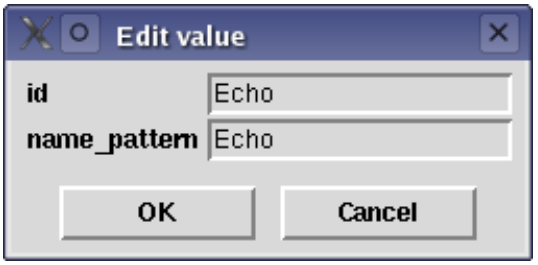

Figure 6.8: Name pattern of the `Echo` server

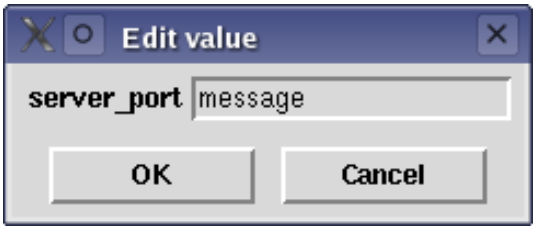

Figure 6.9: Port name of the `Echo` server

```
#  Time: 21:29:44

COMPONENT:
 id = Echo
 name = Echo

PORT:
 name = message
 type = inout

CONNECTIONS:
 message —— Echo.message

STATECHART:
 Send [DS]
 Wait

ENTER:
 N: Send
 O: from random import randint
   Messages=["Hello, everyone!", "Have a nice day!", "How are you today?", "I feel very well \
today!", "The same to you!"]
   MessageNo=len(Messages)


TRANSITION:
 S: Send
 N: Wait
 T: 0 [RTT]
 C: 1
 O: msg=Messages[randint(0, MessageNo—1)]
   [EVENT("message.send", msg)]
   [DUMP("Sent: " + msg)]

TRANSITION:
 S: Wait
 N: Send
 E: message.echo
 C: 1
 O: [DUMP("Received: " + [PARAMS])]
```

Here is `Echo.des`:

```
# DCharts description generated by SVM—AToM3—plugin, written by Thomas Feng
#  Source: /home/thomas/Backup/Atom3_2.2/DCharts/models/SimpleEcho/Echo.py
#  Date: January 15, 2004
#  Time: 21:31:4

PORT:
 name = message
 type = inout

CONNECTIONS:

STATECHART:
 Receive [DS]
```

```
  Echo


TRANSITION:
  S: Receive
  N: Echo
  E: message.send
  C: 1
  O: msg=[PARAMS]

TRANSITION:
  S: Echo
  N: Receive
  T: 1 [RTT]
  C: 1
  O: [EVENT("message.echo", msg)]
```

This example can also be found in [39].

6.6 Debugging

The SVM simulator supports low-level debugging. Its debug mode is entered whenever the user inputs the "`debug`" special event. If the simulation makes use of a model-specific interface, the debug mode may be entered in a different way. For example, the `CDPlayer` example in the SVM distribution provides a "Debug" button that switches to the debug mode.

When the debug mode is entered, the simulation is suspended. The user is allowed to execute arbitrary Python code. If the default curses interface or the default graphical interface is used, the Python code entered by the user is highlighted according to a combined syntax of Python and SVM model description.

The Python code executed in the debug mode may inspect the status of the simulation and the model running in it, as well as modify their variables. `eventhandler` is an important object that contains most information concerning the simulation. Its following attributes are useful for debugging:

- `eventhandler.state` contains the current leaf states (a Python list of strings). Modifying this list changes the state of the model.
- `eventhandler.trans` contains the definition of all the transitions. It is a Python dictionary.
- `eventhandler.stateH` is another Python dictionary that contains the definition of the state hierarchy.
- `eventhandler.enter` and `eventhandler.exit` are the two Python dictionaries that contain the enter actions and the exit actions, respectively.
- `eventhandler.ports` is a Python dictionary that contains all the ports and their properties.
- `eventhandler.connections` is a Python dictionary that contains all the required connections between this model and other components.

7 MODEL VERIFICATION

Model checking, model verification and debugging are the three methods to improve the correctness of a model or to find out potential errors in it.

Model checking checks the correctness of a model by means of formal property proving. This checking does not depend on individual experiments. The properties, such as reachable states and acceptable event lists, are always true for the model. *Model verification* checks the correctness of a model by means of multiple simulations. The common properties of those simulations are summarized and are regarded as properties of the model. For example, the states that are not entered in all those simulations are considered unreachable states of the model. However, this conclusion may not be correct because of the non-exhaustive sampling of all possible behaviors. In fact, to achieve 100% certainty for a single property, an infinite number of simulations are usually required. Those simulations exhaust all possible traces of the model simulation. This is, of course, impossible. As a result, model verification is less formal than model checking. *Debugging* is the least formal in this comparison, since it is responsible for only one simulation or one group of simulations. When an error occurs in a simulation, the debugger usually looks inside the faulty part of the model to locate the error. When a design error is discovered, the debugger tries to fix it without affecting the other parts. However, this is usually impossible, and the result of this modification becomes unpredictable.

This chapter mainly discusses model verification, as it is the most well-studied approach. Our paper on consistency checking [40] contains a general discussion and an example of model verification with SVM. Formal model checking of DCharts is interesting and useful for many applications. It will be studied in the future. Debugging of DCharts models in the SVM simulator has been discussed in the previous chapter.

7.1 Simulation Trace

A *simulation trace* records the evolution of the state (and possibly, the messages being sent) over time. In SVM it is obtained as text output of a simulation. This output is sent by the model with the `DUMP` macro. Different models may have different output formats. However, if the models conform to a single standard and provide enough information in the output, the output trace is useful to check the correctness of those models.

An example of chat rooms and clients is introduced in [40]. Chat rooms and clients are the two different types of DCharts components. The communication between them conforms to the following simplified protocol:

- There are 5 clients and 2 chat rooms in the system. Initially, the clients are not connected. They try to connect to a random chat room every 1 to 3 seconds (uniformly distributed). The requested chat room instantaneously receives the request (zero network delay and reliable communication are assumed).
- A chat room accepts at most 3 clients. It accepts a connection request if and only if its capacity is not exceeded.
- The requesting client receives an acceptance or rejection notice from the requested server immediately.
- A client must be accepted by a chat room before it may send chat messages.
- When connected, a client sends random messages to the chat room that it is connected to every 1 to 5 seconds (uniformly distributed). The chat room immediately receives the messages. It takes 1 second to process a message and broadcast it to all the clients connected to it, except the sender.

- The clients instantaneously receive the broadcast.

The following is a part of the output trace generated by a simulation of the whole system with 2 chat rooms and 5 clients:

```
......
CLOCK: (10.5s,0)
Client 0
Says "Hello!" to ChatRoom 1
......
CLOCK: (11.5s,0)
ChatRoom 1
Broadcasts "Hello!" to all clients except Client 0
......
CLOCK: (11.5s,2)
Client 1
Receives "Hello!" from Client 0
......
```

For more insight into this example, the readers are referred to the paper published at the UML 2003 conference [40]. This example is cited here only to demonstrate model verification. This output trace consists of a number of output segments, each of which is composed of three lines: the time when the output is produced, the component that produces the output and a brief message from the component.

The time is written in an enhanced format. A *time tuple* consists of a float number that denotes the number of seconds elapsed since the simulation was started, and an integer that denotes the sequence of the multiple events received at that time. For example, time `(10.5s,0)` means 10.5 seconds have elapsed since the simulation is started and the event is the first $(0+1)$ to occur at that time. Time `(11.5s,2)` means 11.5 seconds after the simulation is started and that the event is the third $(2+1)$ to occur at that time.

This extended time format allows the specification of multiple events that occur at exactly the same time, while their order is still important. For example, according to the protocol, *as soon as* the server broadcasts a message, the clients receive it. The two events occur at exactly the same time, but in the output trace, the message sent from the chat room must appear before the message received by the clients (a causality constraint).

7.2 Extended Regular Expressions

Checking the consistency between the protocol and the output trace is a kind of model verification. An automatic approach is taken to check this consistency for each simulation. If a large number of checks are successful, confidence in model correctness increases.

Before automatic checking can be done, the protocol must be translated into a formal description to be processed by computer programs. Here, a rule-based approach is employed. A rule file to be processed contains several rules that the output trace must conform to. The rules are written with *extended regular expressions*, an extended form of UNIX regular expressions. Each rule consists of 4 parts: pre-condition, post-condition, guard (optional) and counter-rule property (optional). *Pre-condition* is a regular expression used to match a part of the output trace. It, combined with the *guard* (a boolean expression), defines when and where the rule is applicable. If it is applicable and the *counter-rule property* is `false`, the *post-condition* (another regular expression) must be found in the output; on the contrary, if counter-rule is `true`, the post-condition must *not* be found.

For example, the rule in Table 7.1 expresses the requirement that the sender of a message does NOT receive the broadcast after 1 second. (However, it does not address whether it can receive the message after 0.9999 second or 1.0001 seconds.)

In the pre-condition, five *groups* are defined between parentheses. They are numbered 1 to 5 in the order

pre-condition	`CLOCK: \((\d+\.{0,1}\d*)s,(\d+\.{0,1}\d*)\)\n\Client (\d+)\nSays "(.*?)" to ChatRoom (\d+)\n`
post-condition	`CLOCK: \([(\1+1)]s,(\d+\.{0,1}\d*)\)\nClient [(\3)]\n Receives "[(\4)]" from Client [(\3)]\n`
guard	`[(\1+1)]<50`
counter-rule	`true`

Table 7.1: An example of an extended regular expression

of their appearance. Group 1 matches the floating-point time. Group 2 matches the sequence number. They constitute a time tuple. Group 3 matches the integer ID of the sender. Group 4 matches the message, which is an arbitrary string. Group 5 matches the ID of the chat room that the sender is connected to.

In the post-condition, `[(...)]` contains an expression, where values of groups can be cited with their index numbers behind "\". Thus, `[(\1+1)]` is the value of the first group plus 1. `[(\3)]` is equal to group 3. More about regular expressions can be found in [41].

Suppose the execution stops at simulated time 50. The checking should not exceed time 50. Without additional conditions, if a message is sent to a chat room at time 49.5, the checker would expect a corresponding broadcast at time 50.5. To cope with this, a guard `[(\1+1)]<50` is added. This tells the checker that the rule is applicable only when the value of group 1 (floating-point time) plus 1 is less than 50.

Since a client should not receive its own message, the counter-rule property is set to `true`.

7.3 Rule Checker

A rule checker is implemented to read in a text file with rules defined in it, and check the correctness of the output trace saved in another text file.

The algorithm of the rule checker is summarized below (suppose that the rule file is read into *rules* and the output trace is read into *outtrace*):

```
function check(rules, outtrace)
    for each rule r in the rules
        pre = the pre-condition
        post = the post-condition
        cond = the condition
        counter = the counter-rule property
        pos = 0
        while true
            match = search(pre, outtrace, pos)
            if match is empty then
                break
            else
                pos = the last position of the match in outtrace
            if cond is not empty
                replace(cond, match)
                if cond is not satisfied then
                    continue
            replace(post, match)
            if (counter and search(post, outtrace, 0) is not empty) or
                    (not counter and search(post, outtrace, 0) is empty) then
                output an error and exit
    return successful
```

```
function search(re, text, pos)
    search regular expression re in text starting from position pos
    if the pattern is found then
        return the matching with the value of all the groups in the pattern
    else
        return empty

function replace(text, match)
    i = 0
    while i < number of groups in match
        replace all the citations of group i in text with the actual value of group i
        i = i + 1
```

7.4 Limitation and Future Work

Model verification with extended regular expressions is very useful. In theory, most properties concerning the behavior of a model can be expressed with rules and be written in text files as the input to the rule checker. However, it is not an easy task to write such a rule file with extended regular expressions. The rules in the file may contain errors themselves. As a consequence, the result of this largely manual verification process is unreliable.

This approach can be greatly improved by developing a method to automatically generate rules from other formalisms such as UML sequence diagrams. (However, there is a large gap between the protocol specified in natural language and formal specifications.) The future work in this area will mostly focus on making this approach practical by developing more tools and reducing human intervention.

Model checking, since it is much more formal than model verification, overcomes some of the vulnerabilities of model verification. For example, if a property is formally proved to exist in a model, it always holds no matter how many simulations are made. However, for model verification to reach this certainty, an infinite number of simulations are usually required.

Model checking of DCharts is not easy. This is mainly because DCharts support variables and arbitrary actions that modify those variables. The result of this modification is hardly predictable statically. A promising approach of model checking is to transform DCharts into other formalisms such as PetriNets [42], and formally check the properties of the new models. Graph grammars [43] [44] [45] [33] are useful for model transformation, because of their well-developed theory. In view of this, the future work in the area of model checking will mainly focus on possible transformations from DCharts models to other formalisms by means of graph grammars.

8

SCC – A DCHARTS COMPILER

SCC (StateChart Compiler) is a command-line tool to synthesize executable code from DCharts models. It optimizes the models and produces efficient code. The code is independent of the SVM simulator.

SCC is able to synthesize code in Java, C++, C# and Python. Those target languages can be chosen on the command-line when the user invokes SCC.

SCC is distributed with SVM. It is started with the `scc` script (or `scc.bat` for DOS). A command-line parameter specifies the `.des` file name of a model description. The code is written to a file with the same name as the model description (with its extension changed according to the target language).

This chapter mainly discusses code synthesis in Java. Several classes are defined in a single Java source file, so when it is compiled with JDK (Java Development Kit), multiple `.class` files are produced. The class with the same name as the Java source file and the model description is the main public class. A `main` function is defined in this class, which provides the default textual interface.

More information about SCC can be found at its homepage:

`http://msdl.cs.mcgill.ca/people/tfeng/?research=scc`

Usage and several examples on SCC can be found in the *SVM and SCC Tutorial* [39].

8.1 Java Code Design

SCC invokes the functions in the Python module `JavaGenerator.py` to generate source code from DCharts. The code is included as a template in the module. SVM-style macros are defined in the template. For different models, the template is the same, but the macros are given different values. The code synthesizer simply substitutes those macros in the template with their values, and writes the result to a text file.

The Java code design refers to the design of the template for Java code generation.

8.1.1 Class Hierarchy

In the class hierarchy of the generated Java source code (Figure 8.1), class `StateMachine` is the common superclass of all the DCharts models. It defines the common interface for the models, so that one model may invoke methods in another without explicitly specifying its concrete type. For each DCharts model (or submodel to be imported), a Java class with the same name is synthesized. The class with the same name as the `.des` file specified on the command-line is the *main class*.

SCC searches for the submodels to be imported into the main model. Those submodels are converted into corresponding Java classes, written in the same Java source file. In addition, it also generates code for the subsubmodels (if any) imported into those submodels. This search repeats until no new model is found under the `IMPORTATION` descriptor of all those model descriptions. If a model is imported by more than one model, or a model is imported by itself directly or indirectly, it is converted into only one class, which can then be reused in different importing models.

For example, in Figure 8.1, class `MainModel` is generated from a DCharts model in `MainModel.des`. It imports other models, and those imported models also import more models in their own right. Classes

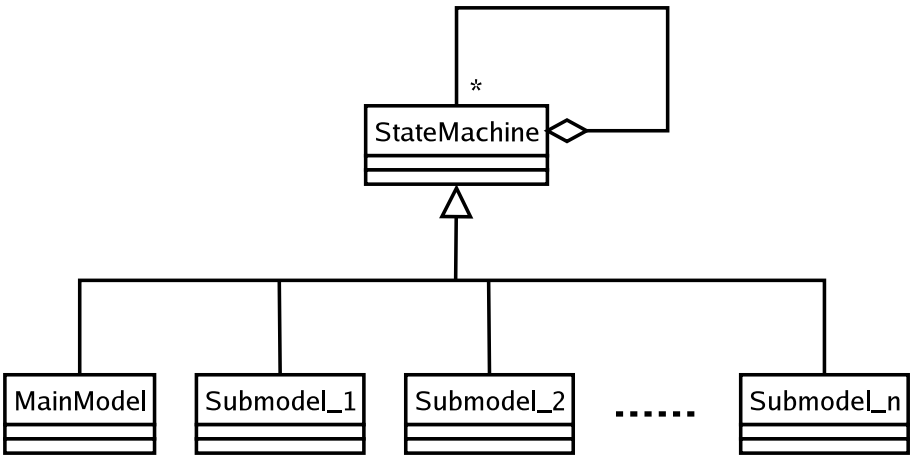

Figure 8.1: Java class hierarchy of state machines

`Submodel_1` to `Submodel_n` are generated from those imported models. All the above classes inherit class `StateMachine`.

8.1.2 Numbering

A unique integer number is assigned to each state of a DCharts model. Internally, the state number is used instead of the name or full path of the state. This has two effects:

- The execution becomes more efficient because string comparisons between state names is reduced to integer comparisons.
- The state hierarchy is partly flattened because those integer IDs do not contain any hierarchical information as can be found in the full paths of the states.

A linked list of current leaf states is maintained in each model. It may contain more than one state ID for a model with orthogonal components.

Two states of different models may have the same ID, whether there is an importation relation between those models or not. The Java function to get the current state first checks the leaf states that the main model is currently in. If any of those leaf states is originally (before importation) an importation state, it then further checks the current leaf states of the submodel imported in that state. This is because, unlike SVM, SCC does not merge the imported model with the importing model but it records the imported model as an attribute of the importation state. This lookup process repeats until the bottom of the state hierarchy is reached.

Similarly, all the events are numbered. Those event IDs are the internal representation of the events that trigger transitions. The synthesized code uses a switch-case structure to test acceptable events. Events handled by different models may have the same ID, even if they have different event names in their model descriptions.

8.1.3 Members of Model Classes

The following constants are defined in each model class. They contain information about the model structure.

- `private static final int StateNum`. The number of states in the model, not including the states in its submodels.
- `private static final String[] EventNames`. The names of the events to be handled by the model, not including the events handled by its submodels. The indexes of those event names represent their IDs. Those IDs start from 0.

- `private static final String[] StateNames`. The full paths of the states in the model, not including the states in its submodels. The indexes of those full paths represent the IDs of the states. Those IDs start from 0.
- `private static final int[] ParentTable`. The table of parent-children relations in the model. `ParentTable[i]` contains the ID of the parent of state `i`. This is the inverse of the *C* function in the abstract syntax.
- `private static final boolean[][] Hierarchy`. The children function of the model (the same as the *C* function in the abstract syntax). `Hierarchy[i]` is an array over all the states. `Hierarchy[i][j]` is `true` if and only if state `j` is a child of state `i`.
- `private static final int[] HistoryStateTable`. The definition of history states in the model. Each element in this array has the following meaning:

$$HistoryStateTable[i] = \begin{cases} 0, & if\ state\ i\ has\ no\ history \\ 1, & if\ state\ i\ has\ a\ normal\ history \\ 2, & if\ state\ i\ has\ a\ deep\ history \end{cases}$$

- `private static final String[] LeafStateTable`. The definition of leaf states in the model, including importation states if any. If state `i` is a leaf state, `LeafStateTable[i]` is equal to the full path of state `i`; otherwise, `LeafStateTable[i]` is equal to `null`.

The following are the Java data structures to store the state of the model at run-time:

- `private State state`. The current leaf state list of the model. Class `State` is a linked list of state IDs.
- `private StateMachine[] Submodels`. The submodels of the model. `Submodels[i]` is not `null` if and only if state `i` is an importation state and the submodel in it has been loaded. Once the submodel is loaded, it is never deleted even if the model leaves the importation state. The history recorded in the submodel object is the history of the importation state.
- `private History[] history`. The history of each state in the model. `history[i]` is not `null` if and only if a history is recorded for state `i`. Class `History` is an internal structure that records a single history. Its value is changed as the model enters the state again and leaves it from another substate.

As all of the attributes of a model class are private members, the users can only access them by means of public methods. The following list includes some of the important methods:

- `public modelname()`. Constructor of the model class. (`modelname` is the name of the model.) The required data structures are initialized. However, the model is not initialized. Its current state is illegal.
- `public void initModel()`. Initializes the model. This means to place the model in its default state(s). The initializer of the model is executed.
- `public boolean isInState(String s)` and `public boolean isInState(int s)`. These two functions check whether the model is in a certain state. The state can be specified with its full path or with its integer ID. The second method is more efficient since it does not require string comparison. The method with a string parameter is kept only for interaction with the model users, who do not know the internal state IDs.
- `public boolean handleEvent(String se)`. The handler of any event. The event is given as a string, and this function automatically converts the string into its integer ID for internal use. It tests the event ID with a switch-case control structure. It checks the source states of the transitions and their guards. If any enabled transition is found, the state variables are changed according to the original design of the DCharts model. This method also executes the output and the enter/exit actions (if any) as the transition is fired. If the model changes to a final state, it also executes the finalizer before returning.

- `public void changeState(int s1, int s2, boolean check_history)`. Changes the model from state `s1` to `s2`. This method implements the state change triggered by a transition from state `s1` to `s2`. If `check_history` is `true`, the history of s2 or any state in the path from state *Common*$(s1, s2)$ to state `s2` is considered (such a transition has a `[HS]` property in the original DCharts model); otherwise, history is ignored even if it is recorded.

  `public void changeState(int s1, int s2)` is equivalent to `public void changeState(int s1, int s2, false)`.
- `protected int eventStr2Int(String event)`. Converts a string event name into its integer ID.
- `protected StringList getCurrentStateList()`. Retrieves the current leaf state(s) of the model. The result is stored in a linked list of strings. Its elements are instances of class `StringList`. This method looks up all the current leaf states in the main model, as well as the current leaf states in all the submodels. It is used internally in the Java class. To retrieve the current states in a more understandable format, the designers should use function `public String getCurrentState()`.
- `public String getCurrentState()`. This method invokes method `protected StringList getCurrentStateList()`, and returns the current leaf state(s) as a string enclosed by a pair of square brackets. Multiple states are separated by comma ",".
- `public int getParentState(int state)`. Returns the parent state of the specified state.
- `public boolean isHistoryState(int state)`. Tests if the specified state has a history.
- `public boolean isLeafState(String state)`. Tests if the specified state is a leaf state.
- `public Hierarchy getHierarchy(int start_level, String state_prefix)`. Returns the state hierarchy structure of the model. The hierarchy structure is a linked list. The `Next` attribute of each element (if it is non-null) gives access to the next element. Each element in this linked list has the following attributes:

  - `public String StateName`. The name of a state in the model.
  - `public String PathName`. The full path of the state.
  - `public int StateNum`. The integer ID of the state.
  - `public int Level`. An integer that denotes the level which the state is at. The larger this number is, the deeper the state is in the state hierarchy.

  Parameter `start_level` is an integer to be added to the `Level` attributes of all the elements in the returned list. This is useful for importation states. Suppose an importation state is at level 5. It calls the `getHierarchy` method of the imported model with `start_level=5`. The generated hierarchy starts from level 6.

  Parameter `state_prefix` is the prefix to be added to the head of the `PathName` attributes of all the elements in the returned list. For example, importation state "`A.B`" calls the `getHierarchy` method of the imported model with `state_prefix="A.B"`. The full paths in the generated hierarchy start with "`A.B.`".

8.1.4 Default Textual Interface

The `main` function of a model class provides a default textual interface. The user inputs events to the model from this interface, and every time the current states of the model are changed, the new states are displayed as a list between square brackets before the "$>$" prompt.

Figure 8.2 shows this default textual interface. It is started by executing the class of the main model in the JVM (Java Virtual Machine). For each model or submodel, this textual interface is defined in a `main` function. By default, JVM starts the `main` function of the main model. The `main` functions of submodel classes, if necessary, can be invoked by user-defined classes.

The user may reuse the Java code of a model with a customized interface instead of the default textual

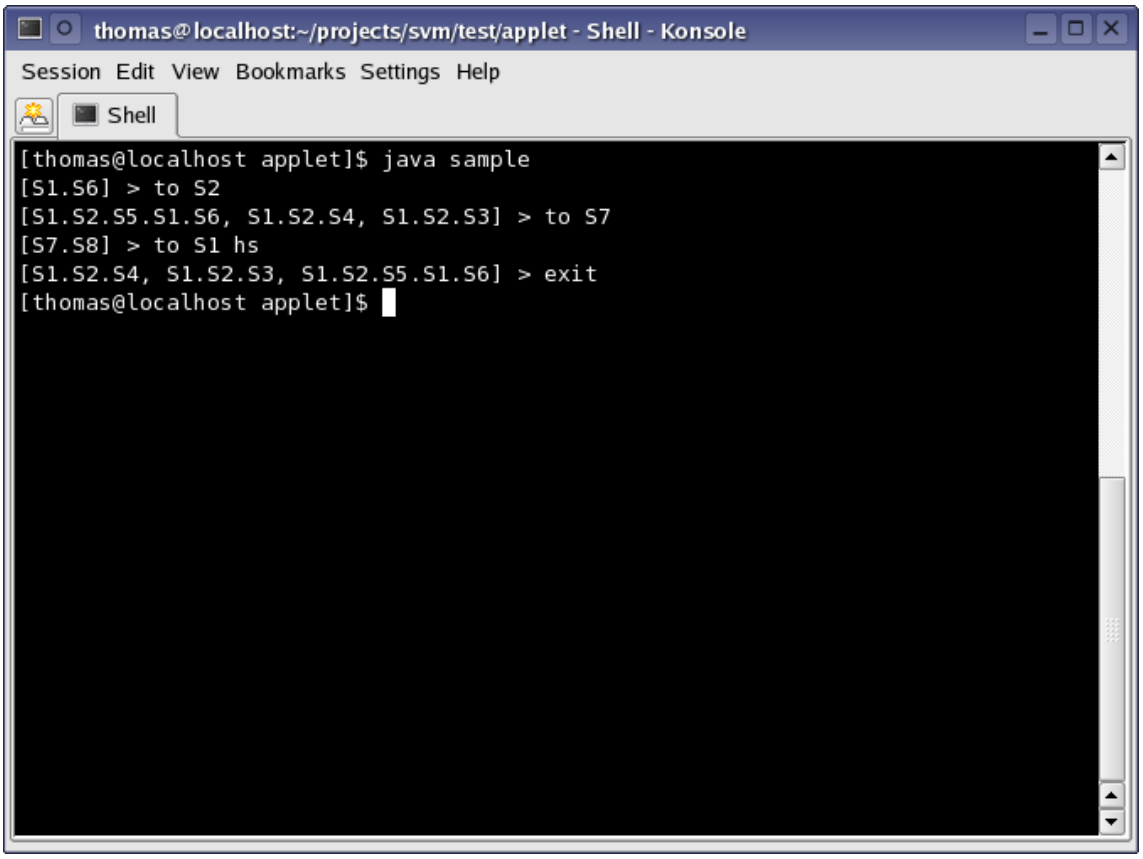

Figure 8.2: An example of the default textual interface of the Java code synthesized by SCC

interface. This is discussed in section 8.5.

8.2 Transformation Strategies

The strategies used to transform different parts of a DCharts model into source code are discussed in this section.

8.2.1 State Hierarchy

As was discussed previously, each state in a model is given an integer ID. Those IDs are unique within a single model but may be duplicated across different models. This is a flattening of the state hierarchy, which causes a loss of information, such as the parent-children relations and information about orthogonal components. Auxiliary functions and arrays are generated to preserve information like this.

Constant attributes `ParentTable` and `Hierarchy` of the model class record the parent-children relations. `ParentTable[i]` contains the state ID (≥ 0) of the parent of state `i`. If state `i` is at the top level, `ParentTable[i]` is equal to `-1`. `Hierarchy[i][j]` is a boolean specifying whether state `j` is a child of state `i`. With these data structures, the following simple Java function tests if a state is the parent of another state (suppose that a state with ID less than 0 is parent of any state):

```
private boolean isParent(int sp, int sc) {
    return sc>=0 && (sp<0 || Hierarchy[sp][sc]);
}
```

Each model class has a state hierarchy defined in it. The hierarchy of a submodel is not visible from the model that imports it. However, the importation states are statically decided in the importing model. A model calls the member functions of its submodels to access to their states.

8.2.2 State Properties

Most of the state properties are statically coded in multiple parts of the Java classes. For example, the property of default states are implemented in these functions:

- The `initModel` function changes the state of the model to its default leaf states. Those default leaf states are computed statically. For example, if states number `4` and `6` are the default leaf states, the following statements are statically coded in function `initModel`: `addInState(4); addInState(6)`. Method `private boolean addInState(int s)` simply adds a state to the current state list of the model. If the model is already in state `s`, the function returns `false`. Since the current state list of the model is empty when it is being initialized, this function always returns `true` when invoked by `initModel`.
- The default states are statically coded in function `changeState`, which changes the model from one state to another. If the new state is not a leaf state or it is orthogonal to other states in the path from the $Common(SRC, DES)$ to the DES of the transition, default leaf states are generated according to the model structure. SCC invokes the SVM simulator to decide those default leaf states as if the model were being simulated. The default leaf states which need to be added for each transition do not vary in different simulations or executions.

State properties concerning transition priorities (`[ITF]`, `OTF` and `RTO`) are statically interpreted. The transitions are sorted according to the algorithm in section 2.2.5. Those transitions are coded in the Java classes in the same order with a switch-case structure. The first enabled transition at run-time is always the one with the highest total priority. This sorting of a submodel's transitions does not vary, because the importing model is not allowed to modify its global option `InnerTransitionFirst` (which is default to 0).

Orthogonal components are also statically coded in the classes. For a transition going out of an orthogonal component, code is generated to eliminate all other orthogonal components of the same parent. For a transition going into an orthogonal component, the default leaf states of other orthogonal components of the

same parent are added to the current state list. SCC hard-codes this information in the Java code to improve performance.

8.2.3 History

History is the most complex part in the Java code, because it largely depends on the state of the model execution, and cannot be decided statically. The `history` attribute of a model class keeps track of its histories. Its value changes at run-time. The computation of this part is among the most expensive in the execution of a hierarchical model with history.

Method `private void recordHistory(int top_state)` records the history of state `top_state` (suppose it has a history or deep history defined in it) in the `history` attribute.

`history` is an array over the state IDs. `history[i]` for state `i` is an instance of class `History`, which has the following attributes:

- `public int[] States`. The history of all the states when state `i` is exited. `States[j]` contains the ID of the child state of state `j`, which the model is currently in. If the children of state `j` are orthogonal components, `States[j]` is meaningless.
- `public long[] Times`. The time when the history is recorded for each state. When a state with history is entered after its history is recorded, the record with the latest time-stamp is considered most recent and will be restored as the current state(s).
- `public StateMachine Submodel`. The submodel imported in state `i`, or `null` if state `i` is not an importation state. When the model leaves an importation state, the imported model remains, because its history recorded in its own `history` attribute may be useful in the future.

Calls to the `recordHistory` method is statically coded in the `changeState` method. When a model leaves state `i` with a history in it, `recordHistory(i)` is called before the state changes.

When the model enters a state with a history recorded in the `history` attribute, the model dynamically decides the destination states with the history record, and changes the model to them. For normal history, this computation is complex.

Note that history recording is necessary even for non-history states. This is because the compiled model may be imported into such an importation state that it is a history state itself, or some of its superstates are deep history states. In those cases, history manipulation is required for every state.

8.2.4 Event Handling

Method `handleEvent` handles events by comparing their IDs with accepted event IDs. It uses a switch-case structure to test those events, and invokes the `changeState` method to change the current state.

Method `changeState` usually makes the following three calls:

1. `recordHistory(`*com*`)` records the necessary history in the path from $com = Common(SRC, DES)$ to the bottom of the state hierarchy. $Common(SRC, DES)$ is computed statically and stored in a 2-dimensional array. If $Common(SRC, DES)$ does not exist (because *SRC* and *DES* are or belong to two different top-level states), `recordHistory(-1)` is called.
2. `removeOutStates(`*com*`)` causes the model to leave state $com = Common(SRC, DES)$ by removing that state and all its substates from the current state list. If $Common(SRC, DES)$ does not exist, the code generator simply writes statement `state=null;` to clean up the current state list.
3. `generateStates(`*com*`,` *DES*`)` generates new states in the path from $com = Common(SRC, DES)$ to *DES*, and adds them to the current state list.

Tool	Achieve	Sacrifice
SVM	functionality and extensibility	space and speed
SCC	speed	space, modularity and functionality

Table 8.1: Trade-offs between SVM and SCC

8.2.5 Importation

Importation is transformed into instantiation in the synthesized code. A class is generated for each model. When a submodel is imported, the importing model instantiates an object of the submodel class, and associates the new object with the ID of the importation state.

According to the DCharts semantics, an imported model is conceptually a part of the importing model. Once imported, it remains until the simulation or execution finishes. All the states and transitions of the imported model are copied to the inside of the importation state. This semantics is implemented as instantiation as follows:

- Once an object of a submodel class is instantiated and associated with a state ID, it is not deleted until the execution of the top-level model finishes (or, usually until the program exits).
- When the model leaves an importation state, the object associated with it is kept as its history.
- If history is recorded for an importation state, it is restored when the model goes to that state because of the firing of a transition with the `[HS]` property. If the state has a history but the transition does not have the `[HS]` property, the submodel is re-initialized to its default states. In the latter case, for simplicity, the importing model instantiates a new object of the submodel class, and replaces the old one with it. The old submodel object is recycled by the Java garbage collector.
- For each model, the configuration of submodels, including their number and the states that import them, is fixed and can be decided statically. The array `Submodels` of a model class keeps track of all those submodels.

8.3 Space Efficiency and Speed Efficiency

Several aspects and concerns affect the design of SVM and SCC (Table 8.1). There are different emphases of these tools:

- SVM sacrifices *space* and *speed* to achieve *functionality* and *extensibility*. Here, space refers to the memory space required for a simulation. Because SVM is a simulator, speed and space usage is not the most important. However, it must provide a suitable experimental platform for a complete DCharts syntax and semantics. It must also be extensible so that new features can be easily added to the simulator, as DCharts are improved over time.
- SCC sacrifices *space*, *modularity* and *functionality* for *speed*. The purpose of code synthesis is to produce highly efficient code that can be used in practical applications. Hence, speed becomes the most important factor. SCC guarantees high performance for most of the implemented features, but sacrifices the features that are not practical, or warns the users about the implemented but inefficient ones. Modularity is not important either. When a model is transformed into Java code, it does not tend to change any more. The code for different DCharts features is usually mixed in an uninterpretable way to achieve better performance. For example, the model is flattened and its hierarchy information is encoded in its transitions. The code to fire transitions and to change the state of the model is optimized with static state properties such as default states, history and orthogonal components.

The design of SCC reflects the above concerns. The numbering of states and events reduces string comparison to integer comparison. Tables are statically generated, which records the parent-children relations and leaf states.

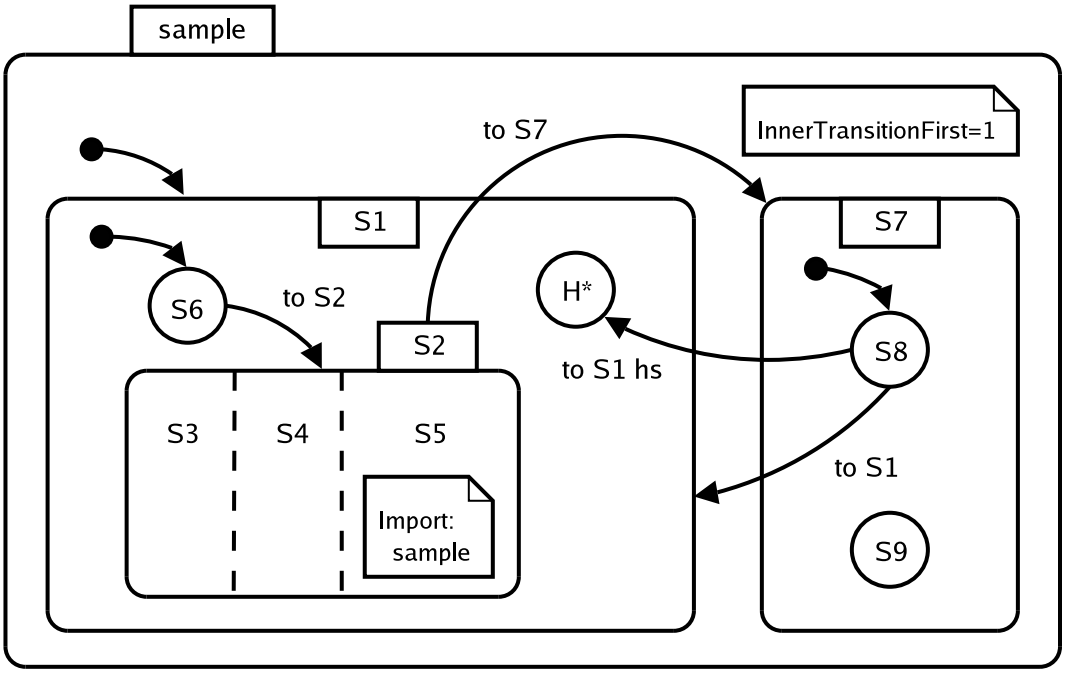

Figure 8.3: The graphical representation of a sample model for SCC

However, history still requires complex computation at run-time. This is because the behavior of models with history is statically unpredictable. For this reason, the use of history (whether it is common history or deep history) is discouraged, if the designer intends to synthesize really efficient code for his/her model.

8.4 Example

A sample model is provided in the `test/applet/` subdirectory of SVM. It demonstrates the use of SCC and the applet interface discussed in section .

The graphical design of the model is in Figure 8.3. This model uses the following features of DCharts:

- default states;
- orthogonal components;
- recursive importation;
- deep history; and
- inner-first transition priorities.

`sample.des`, the textual model description is included below:

```
IMPORTATION:
 sample = sample.des

OPTIONS:
 InnerTransitionFirst = 1

STATECHART:
 S1 [DS] [HS*]
  S2
   S3 [DS] [CS]
   S4 [DS] [CS]
```

```
    S5 [DS] [CS] [sample]
   S6 [DS]
  S7
   S8 [DS]
   S9

TRANSITION:
 S: S1.S6
 N: S1.S2
 E: to S2

TRANSITION:
 S: S1.S2
 N: S7
 E: to S7

TRANSITION: [HS]
 S: S7.S8
 N: S1
 E: to S1 hs

TRANSITION:
 S: S7.S8
 N: S1
 E: to S1
```

When the code (`sample.java`) is synthesized by SCC, the following classes are defined:

- `StateMachine`. The common superclass of all DCharts models.
- `sample`. The main class in the source file. Because `sample.des` only imports itself, no other model class is generated.
- `State`. Data structure of the elements in the current state list.
- `History`. Data structure to record the history of a single state.
- `EventList`. Linked list for returning the enabled event list.
- `StringList`. Linked list over strings.
- `Hierarchy`. Linked list for returning the DCharts hierarchy.

Among the above classes, `sample` is the public class that can be accessed by user classes or from the Java command-line. The user may execute "`java sample`" to run the model.

8.5 Applet Interface

The user may provide a customized interface for the model. This is done by manually writing a Java application, which instantiates the model class and provides input/output channels to it.

A general applet interface is written to be used with any DCharts model. It is embedded in webpages and executed in a JVM, as shown in Figure 8.4. This interface has a similar look as the SVM simulator. The state hierarchy is shown as a tree in the left panel. All the enabled events are listed in the "Events" list. The "Output" box displays the output from the model. The "Command" box accepts commands from the user. Accepted commands are limited to enabled events. Debugging is not supported. Exiting the program is not applicable for an applet.

This applet is written in Java source file `svmapplet.java` in the `test/applet/` subdirectory of the SVM directory. It supports a `model` parameter. Its value is the name of the model class to be loaded. The applet

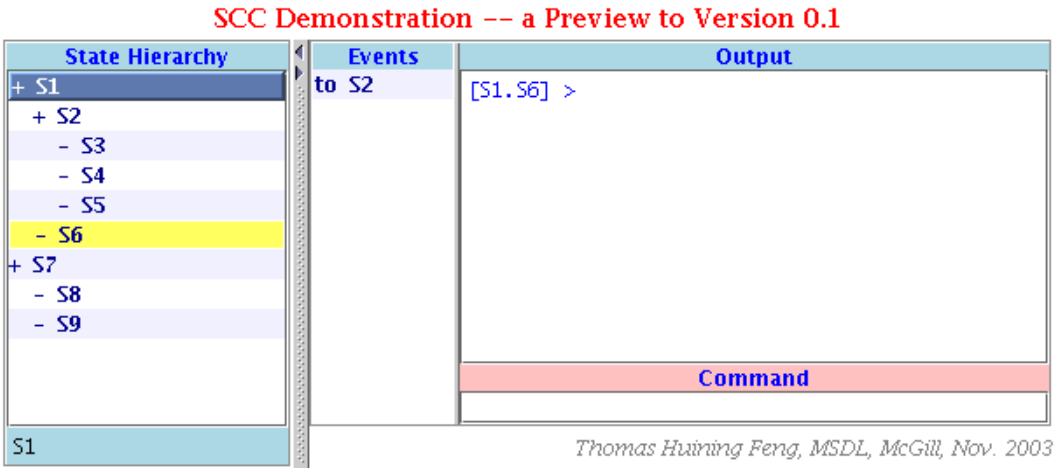

Figure 8.4: Applet interface for the Java code synthesized from a DCharts model

automatically looks for the model class and instantiates an instance of it. If an error occurs, an error message is displayed.

8.6 Limitations

The following DCharts features are not supported. They will be studied in future research:

- Currently, actions and guards are not supported for target-languages Java and C#. If Python is chosen as the target-language, actions and guards are optionally included in the synthesized code (if parameter `--ext` is given on the SCC command-line). The behavior of this Python code with actions and guards is the same as the simulation in SVM. If C++ is chosen instead, the actions and guards may also be included (with the same `--ext` parameter). The code must then be linked to a Python run-time library. In an execution, the binary code automatically loads the Python library, and executes the actions and evaluates the guards. With this, the behavior of the model is also preserved.

  There are many other choices for the implementation of actions and guards. One possibility is to use languages that are independent of specific target-languages, such as action semantics [46] [47] and Modelica [48] [49]. Action semantics is not yet standardized. There is no mature library for it until now. Modelica is a powerful language capable of specifying non-causal equation sets. Actions "`a=b+c, d=a/2`" can thus be written as "`a=2*d, a-b-c=0`". The Modelica compiler symbolically and automatically determines the unknown variables and sorts the equations in an order in which all the equations can be solved sequentially. For example, suppose `a` and `d` are unknown before the actions are executed. Modelica changes the order of the equations and symbolically transforms them. As a result, `a` is solved with "`a=b+c`" first, and then `d` is solved with "`d=a/2`". Though, there is no non-commercial Modelica solver until now, its has a bright future as both an action language and a constraint language.
- Transition parameters are supported only for Python and C++. The `--ext` parameter must be explicitly given on the command-line to invoke SCC.
- Timed transitions are supported only for Python and C++. The `--ext` parameter must be explicitly given on the command-line to invoke SCC. Scheduling events in an execution requires extra threads. This limits the portability and predictability of the model.
- Macros are statically substituted with their values by SCC. The generated code does not contain macros any more. This also implies that macro redefinition is no longer allowed in the synthesized code. When a model imports a submodel, it instantiates the submodel class, which cannot be modified

at run-time. Because of this, the behavior of a model is fixed when code is generated.

- Distributed simulation is not supported. Ports and connections defined in a model description are simply ignored by SCC. This feature is left as future work. It is important and meaningful for SCC to automatically generate distributed systems, where different components (objects in the target language) communicate via a network. If they conform to the same communication protocols as PYPVM and SVMDNS, those components may coexist in a system with some DCharts components simulated by SVM. This makes the system extremely flexible.
- As discussed a previous section, the implementation of history is not efficient.

9 APPLICATIONS

There have been a number of practical applications for SVM and SCC. Some of them are introduced in this chapter.

9.1 Simple Data Types

Data types such as boolean and integer are explicitly modeled with DCharts. Though variables of those types are internally supported, modeling them explicitly allows symbolic checking and analysis.

9.1.1 Boolean

The boolean data type is one of the simplest DCharts models. Its textual description is saved in file `Boolean.des` in the `DataTypes/` subdirectory of SVM.

```
MACRO:
  INIT = true

STATECHART:
  initiate [DS]
  true
  false

TRANSITION:
  S:initiate
  T:0
  N:[INIT]

TRANSITION:
  S:true
  E:chg
  N:false

TRANSITION:
  S:false
  E:chg
  N:true

TRANSITION:
  S:true
  E:get
  N:true
  O:[EVENT('true')]

TRANSITION:
  S:false
  E:get
```

```
 N:false
 O:[EVENT('false')]

ENTER:
 N:true
 O:[DUMP('Current value is true.')]

ENTER:
 N:false
 O:[DUMP('Current value is false.')]
```

This model simulates a boolean data cell, whose value is either `true` or `false`. Macro `INIT` can be redefined in the command-line to give an initial value. By default, it is `true`.

The `chg` event inverts the value in the cell. The `get` event reveals its value to the user by dumping it out.

9.1.2 Integer Counter

An *integer counter* is a cell that stores an integer in it. The only operations on its value are "increase" by 1 and "decrease" by 1.

The model saved in `Counter.des` in the `DataTypes/` subdirectory of SVM models such an integer counter.

```
MACRO:
 INIT = 0
 CURRENT = [INIT]

OPTIONS:
 InnerTransitionFirst = 1

IMPORTATION:
 myself = Counter.des

STATECHART:
 STABLE [DS]
 SMALLER [myself] [INIT = [INIT]] [CURRENT = [EVAL([CURRENT]-1)]]
 LARGER [myself] [INIT = [INIT]] [CURRENT = [EVAL([CURRENT]+1)]]

TRANSITION:
 S:LARGER
 C:[CURRENT] >= [INIT]
 E:dec
 N:STABLE

TRANSITION:
 S:STABLE
 C:[CURRENT] <= [INIT]
 E:dec
 N:SMALLER

TRANSITION:
 S:SMALLER
 C:[CURRENT] <= [INIT]
 E:inc
 N:STABLE

TRANSITION:
 S:STABLE
```

```
 C:[CURRENT] >= [INIT]
 E:inc
 N:LARGER

TRANSITION:
 S:STABLE
 E:get
 N:STABLE
 O:[EVENT('CURRENT')]

ENTER:
 N:STABLE
 O:[DUMP('Current value is [CURRENT].')]
```

The `INIT` macro can be redefined to give a different initial integer value to the cell.

In this model, recursive importation is extensively used. When the model receives the `inc` event after it is initialized, a submodel with the same structure is imported into the `LARGER` state (because its value becomes [INIT]+1, which is larger than [INIT]). The value in the submodel is redefined as `[CURRENT]+1` in the submodel. Since transitions in this model are inner-first, if the `get` event is received at this time, a transition in the submodel instead of the importing model is triggered. The model returns the new value. When `dec` is received at this time, the model goes out of the submodel. The `[CURRENT]` value of the model at the higher level is 1 less than the `[CURRENT]` value of its submodel in the `LARGER` state. A `get` event received at this time is handled by the importing model itself.

If the value of the cell becomes less than `[INIT]` because of `dec` events, submodels are imported into its `SMALLER` state.

The cell has a theoretically infinite capacity, which only depends on the available memory of the system.

9.1.3 Integer

The integer model [26] is similar to the counter. However, it allows the user to directly set its value to an arbitrary number. It also has an upper bound and a lower bound. At any time during a simulation, any number between the lower bound (inclusively) and the upper bound (exclusively) are accepted as an event. The value of the cell is set accordingly.

The integer model is saved in `Integer.des` in the `DataTypes/` subdirectory of SVM.

```
MACRO:
 MIN = 0
 MAX = 9
 INIT = [MIN]
 FIRST = 1

IMPORTATION:
 myself = Integer.des

OPTIONS:
 InnerTransitionFirst = 1

STATECHART:
 STABLE [DS]
 TEMP
 LEFT [myself] [MIN = [EVAL([MIN]+1)]] [INIT = [INIT]] [FIRST = [FIRST]] [MAX = [MAX]]
 RIGHT [myself] [MIN = [EVAL([MIN]+1)]] [INIT = [INIT]] [FIRST = 0] [MAX = [MAX]]

TRANSITION:
```

```
  S:STABLE
  T:0
  C:[MIN] <= [MAX]
  N:LEFT

TRANSITION:
  S:STABLE
  T:0
  C:[MIN] > [MAX] and [FIRST]==1
  N:TEMP
  O:[EVENT('[INIT]')]
# When the bottom is reached and it is initiating,
# sent an event of the [INIT] character

TRANSITION:
  S:LEFT
  E:[MIN]
  N:RIGHT
  O:[DUMP('Current value is [MIN].')]

TRANSITION:
  S:RIGHT
  E:[MIN]
  N:RIGHT
  O:[DUMP('Current value is [MIN].')]

TRANSITION:
  S:RIGHT
  E:get
  N:RIGHT
  O:[EVENT('[MIN]')]
```

By default, the lower bound (`[MIN]`) of the cell is 0, and the upper bound (`[MAX]`) is 10. The idea is to structure all the possibilities in a bi-tree. Valid states in a model execution include `LEFT.LEFT.RIGHT...STABLE` and `RIGHT.LEFT.LEFT.RIGHT.LEFT.RIGHT...STABLE`. (There are 11 levels in total, with the last one named `STABLE`.) The rightmost `RIGHT` represents the current value. Suppose the name components in "`...`" are all `LEFT`, then the first state represents integer 2, and the second represents 5.

When initiated, the model nests deep enough so that the transitions at the first level are duplicated (with only the event names changed) 10 times. When the innermost `STABLE` state is reached, events of all those states are accepted. For the `get` event to return the current value from the deepest `RIGHT` state, the transitions in this model must be inner-first ordered.

Having nested deep enough (`[MIN]>[MAX]`) and the model is being initialized (`[FIRST]=1`), the state changes to `TEMP` – a dummy state, and at the same time event `[INIT]` is broadcast. The cell immediately changes to the initial value. Whenever the first `RIGHT` state is entered, the model is no longer being initialized and is able to accept events from the user (possibly input from the SVM graphical interface). The `[FIRST]` is then set to 0.

When an event between 0 (the `[MIN]` value) and 9 (the `[MAX]` value minus 1) is received, the state in the appropriate level changes to `RIGHT`. If it is already in `RIGHT`, a self-loop is triggered. The self-loop eliminates the `RIGHT` states at all the lower levels, so it becomes the deepest `RIGHT` state.

9.2 The Clock Component for Virtual-Time Simulation

SVM only supports real-time execution. However, virtual-time simulation is required sometimes. The clock component makes it possible to simulate DCharts models in an as-fast-as-possible way.

The textual description of the clock component is included below:

```
# Clock component for tight coupling

MACRO:
 CHECKINTERVAL = 0
 STARTTIME = 0

INITIALIZER:
 sched=[]
 global_time=[STARTTIME]
 def sched_cmp(a, b):
  return cmp(a[1], b[1])

STATECHART:
 NORMAL [DS]

TRANSITION:
# schedule event
#  param 1: scheduler ID
#  param 2: schedule time
 S: NORMAL
 N: NORMAL
 E: schedule
 O: sched.append([PARAMS])

TRANSITION:
# idle checker
#  notifies the earliest scheduled event
 S: NORMAL
 N: NORMAL
 T: [CHECKINTERVAL] [RTT]
 C: len(eventhandler.event_list)==1 and len(sched)>0
 O: sched.sort(sched_cmp)
   s=sched[0]
   del sched[0]
   global_time=s[1]
   [EVENT("notify", s)]
   # param 1: scheduler ID
   # param 2: schedule time

TRANSITION:
# time retrieval
 S: NORMAL
 N: NORMAL
 E: gettime
 O: [EVENT("timereturn", [global_time])
   # param 1: current global time
```

The clock component uses variables to explicitly model the scheduler of an as-fast-as-possible simulation. (This clock component is not functional in distributed simulation. Timewarp [31] technology is needed for as-fast-as-possible distributed simulation.) To use this component, the designer designs a real-time model as usual, but imports the clock as a top-level orthogonal component. Some transitions in the model need to be modified to interact with the scheduler. After this, the real-time model is converted into a virtual-time model. In a virtual-time model, there must be exactly one clock component.

In a real-time simulation, a model schedules transitions simply with the *after* special event. The transitions with this event are triggered after the specified number of seconds. In as-fast-as-possible simulation, scheduling becomes different. The simulator does not really wait. When no event is scheduled at the current time, the virtual time counter is immediately increased to be the next scheduled time, and the transitions scheduled at that time are fired without delay.

When the clock component is used, the model schedules transitions with the `schedule` event. This event is broadcast by other parts of the model, and it is handled by the clock component. Two parameters must be sent with this event:

1. The first parameter is an arbitrary ID. This ID can be any Python variable. When the virtual time becomes equal to the scheduled time, the clock broadcasts a notifier with this ID as a parameter. Transitions in the model that react to the notifier test this ID in their guards to determine whether or not the event is scheduled by themselves.
2. The second parameter is a float number of the difference between the scheduled time and the current time. It must be positive or 0. If its value is 0, the notifier will be received before the clock component advances the time counter.

For example, if an orthogonal component in the model has ID "`o1`" (arbitrarily determined by the designer), and it schedules a transition after 5.3 seconds, it may send the `schedule` event with action "`[EVENT("schedule", ["o1", 5.3])]`", which is then handled and recorded by the clock component.

The clock component increases the virtual time automatically when no more events are scheduled at the current time. At that time, all the orthogonal components are considered idle because they are waiting for notifiers from the clock. This condition is expressed with the following guard:

```
len(eventhandler.event_list)==1 and len(sched)>0
```

Here, `eventhandler.event_list` is the internal list of scheduled events in SVM. If its length is equal to 1, no event other than the one that the clock component itself schedules is in the event list. This means all other orthogonal components are idle. This guard also checks whether there is any event scheduled in `sched` (the list of schedule requests maintained by the clock component).

When all the other orthogonal components are idle, the clock component increases the time counter to the smallest scheduled time. It then broadcasts notifiers. A notifier is a `notify` event with the same parameters as the `schedule` event that schedules it. If multiple events are scheduled at exactly the same time, the clock component broadcasts multiple notifiers with different parameters. The transitions in other orthogonal components reacting to the `notify` event use guards to test whether they are the ones to be notified. To continue with the last example, "`[PARAMS][0]="o1"`" is the guard of the transition that reacts to the `notify` event.

In real-time simulation, the current time can be retrieved by calling the `time` function in the `time` Python library. This function returns the current time according to the hardware clock. However, as-fast-as-possible simulation uses a different concept of time. The current time is maintained in a time counter. To retrieve the current time from the clock component, a `gettime` event should be sent without parameter. When the clock component receives this event, it immediately replies with a `timereturn` event. The current time (a float number) is the only parameter with the event. The receiver retrieves the current time with "`[PARAMS][0]`".

The following are several rules for the current time broadcast by the clock component:

- It is impossible to request and retrieve the current time in the output of a transition. At least 2 transitions are required for this purpose: one sends the `gettime` event, and the other reacts to the immediate `timereturn` event. (As a trick, the model may directly access the `global_time` variable in the clock component, since all the variables in a model, including those of the clock component, share the same name space. However, this method is not modular.)

Figure 9.1: The MP3 player

- Usually, there is no need to retrieve the current time in a scheduled transition, because the current time is always equal to the time when it is scheduled, received as the second parameter of the `notify` event.
- Using the clock component and the *after* special event (with *t* larger than 0) in combination produces unpredictable result and is strongly discouraged.
- Multiple clock components in the same model conflict with each other. For as-fast-as-possible simulation, there must be exactly one clock component.

9.3 An MP3 Player

An MP3 player is developed according to the division of the 3 parts of a system in Figure 5.8. It is included in the `MP3Player/` subdirectory of SVM.

The model consists of the following files:

- `MP3Player.des`. The main DCharts model of the control logic between the user interface and the hardware driver.
- `MP3PlayerGUI.py`. The model-specific user interface. It is a Python library, where classes and functions concerning the graphical interface are defined. The user interface is instantiated under the `INTERACTOR` descriptor in the main model. Transitions in the main model control the interface by means of the functions defined in the library.
- `MP3Library.py`. The hardware driver library. In this example, the hardware is the PyGame (`http://www.pygame.org/`) MP3 library that provides playback functions. The hardware driver accommodates this conceptual hardware to the main model. Because the hardware is not event-based, the driver starts an extra thread to periodically test the status of the hardware, and generate events to be handled by the main model. The main model also controls the hardware by means of the functions provided by the driver.
- Files `Fwd.gif`, `KsCD.gif`, `MP3GUI.gif`, `PlayPause.gif`, `Rew.gif` and `Stop.gif` are the images to be displayed on the buttons in the graphical interface.

The `FILE` macro in the main model specifies the name of the MP3 file to be played. By default, it is empty. It must be redefined by the user on the command-line. The following statement under the `INITIALIZER` tests the validity of its value. The simulation halts if no file name is given:

```
if "[FILE]"=="":
    print 'usage: svm MP3Player.des "FILE=[.mp3]"'
    exit(1)
```

Figure 9.1 shows the graphical interface of the MP3 player. It is initialized by the following statements under the `INTERACTOR` descriptor:

```
from MP3PlayerGUI import MP3PlayerGUI      # import the GUI class MP3PlayerGUI
root = Tk()
```

Round	Task	Hours
1	developing code	12
	developing tests	8
	running tests	1
	analyzing problems	3
2	developing code	6
	developing tests	4
	running tests	1
	analyzing problems	2
3	developing code	3
	running tests	1
	– passed –	

Table 9.1: Rounds and tasks in a software development process

root.title("MP3 player")
gui = MP3PlayerGUI(root, eventhandler) # instantiate the GUI with the global eventhandler
eventhandler.start() # start the simulation of the model
root.mainloop() # loop infinitely to receive GUI events, until the window is closed

Because SCC supports actions and guards for the Python and C++ target languages, the user may synthesize code for this MP3 player in those languages. This produces a stand-alone application, which does not depend on the Python environment. The `FILE` macro must be explicitly redefined on the SCC command-line, since it is not possible to redefine it in the synthesized code. The user may use the following command to generate `MP3Player.py`, which encodes the complete behavior of the MP3 player (assuming that MP3 file `music.mp3` exists):

```
scc -lpython --ext MP3Player.des "FILE=music.mp3"
```

The command to synthesize code in C++ is similar:

```
scc -lcpp --ext MP3Player.des "FILE=music.mp3"
```

Note that the user need not compile Python source, as Python is an interpreted language. `MP3Player.py` can be directly executed with Python and it plays `music.mp3`. However, the C++ source needs to be compiled and linked with the Python shared library. The need and the command for this compilation is printed to the console when SCC synthesizes the code.

CDPlayer is another meaningful model included in the SVM distribution. It is in the `CDPlayer/` subdirectory. It models a CD player similar to the MP3 player. It is more complex because debugging and snapshotting are supported.

9.4 Simulation of Software Process

Sadaf Mustafiz has built a software process model [50] for SVM. The development process is modeled as several tasks, each of which is "an entity (a real object that exists and has an extended lifetime)" (Sadaf Mustafiz).

To model those tasks with DCharts, each of them corresponds to an active state. For example, a software development process consists of three rounds, as shown in Table 9.1. In the first round, the following tasks are scheduled sequentially: developing code, developing tests, running tests and analyzing problems. The distribution of hours among them is: developing code takes 12 hours, developing tests takes 8 hours, running tests takes 1 hour, and analyzing problems takes 3 hours.

Because problems are discovered in round 1 during the running tests task, another round must be added to fix those problems. Before round 1 finishes, the task of analyzing problems is undertaken to analyze the problems that are to be fixed and the cost to fix them in the next round.

Round 2 is based on the results of round 1. It strives to fix the problems discovered in round 1, as well as to improve the functionality of the system. The development becomes faster: developing code takes 6 hours, developing tests takes 4 hours, running tests takes 1 hour, and analyzing problems takes 2 hours. Because there are still problems found in the running tests task, round 3 is required, which only aims at fixing those remaining problems.

In round 3, the problems are fixed, and all the tests are passed. The development process successfully finishes.

Sadaf Mustafiz has modeled this process with DCharts. The model is simulated with SVM. The output trace is written in text files. The plots of the output trace is shown in Figure 9.2.

More information about the above software process model can be found at the Modeling and Simulation Based Design course homepage:

`http://moncs.cs.mcgill.ca/people/hv/teaching/COMP762B2003/`

9.5 Simulation of TCP

Shah Asaduzzaman and Zaki Hasnain Patel have built a TCP model for SVM. This model simulates communication via the TCP network protocol.

The communication process of the system is shown in Figure 9.3. There are 6 parts in the system:

- The client application generates data with a data generator. The data enter a buffer (FIFO queue). They are sent by the application controller one by one. The client application also listens to the data coming from the TCP driver.
- The TCP driver on the client side accepts data packets from the application controller. It sends messages via a network. It has no buffer. The messages must be sent immediately. It also listens to the incoming data channel.
- The data channel transfers data packets from the client side to the server side.
- The TCP driver on the server side accepts data from the data channel from the client side to the server side. It sends control information to the outgoing data channel.
- The server application computes with the received data packets. It generates control packets. Because the control packets are generated one at a time, buffer is not necessary for the server. The generated control packets are sent to the TCP driver on the server side.
- The data channel transfers control packets from the server side to the client side. Those packets are received by the client TCP driver.

Each part of the system is modeled with a DCharts orthogonal component. The whole system is a combination of those orthogonal components by means of importation (Figure 9.4).

The 6 parts of the system is modeled with submodels imported into the total system:

- The client application is modeled in submodel `ClientApp` (Figure 9.5).
- The TCP driver is modeled in submodel `TCPDriver`. It is for both the client side and the server side, because the API of the TCP protocol on both sides is exactly the same. The `ActiveClose`, `PassiveClose` and `Established` states of the submodel are abstracted. Figures 9.7, 9.8 and 9.9 show the internal structure of those states, respectively.
- The data channels are modeled in submodel `Channel` (Figure 9.10). Both channels in the system are implemented with the same submodel.
- The server application is modeled in submodel `ServerApp` (Figure 9.11).

The channel in Figure 9.10 uses the $after$ special event to simulate delay in the network and the time interval between two subsequent inquiries to the buffer. As a result, this model is a real-time model. To convert it

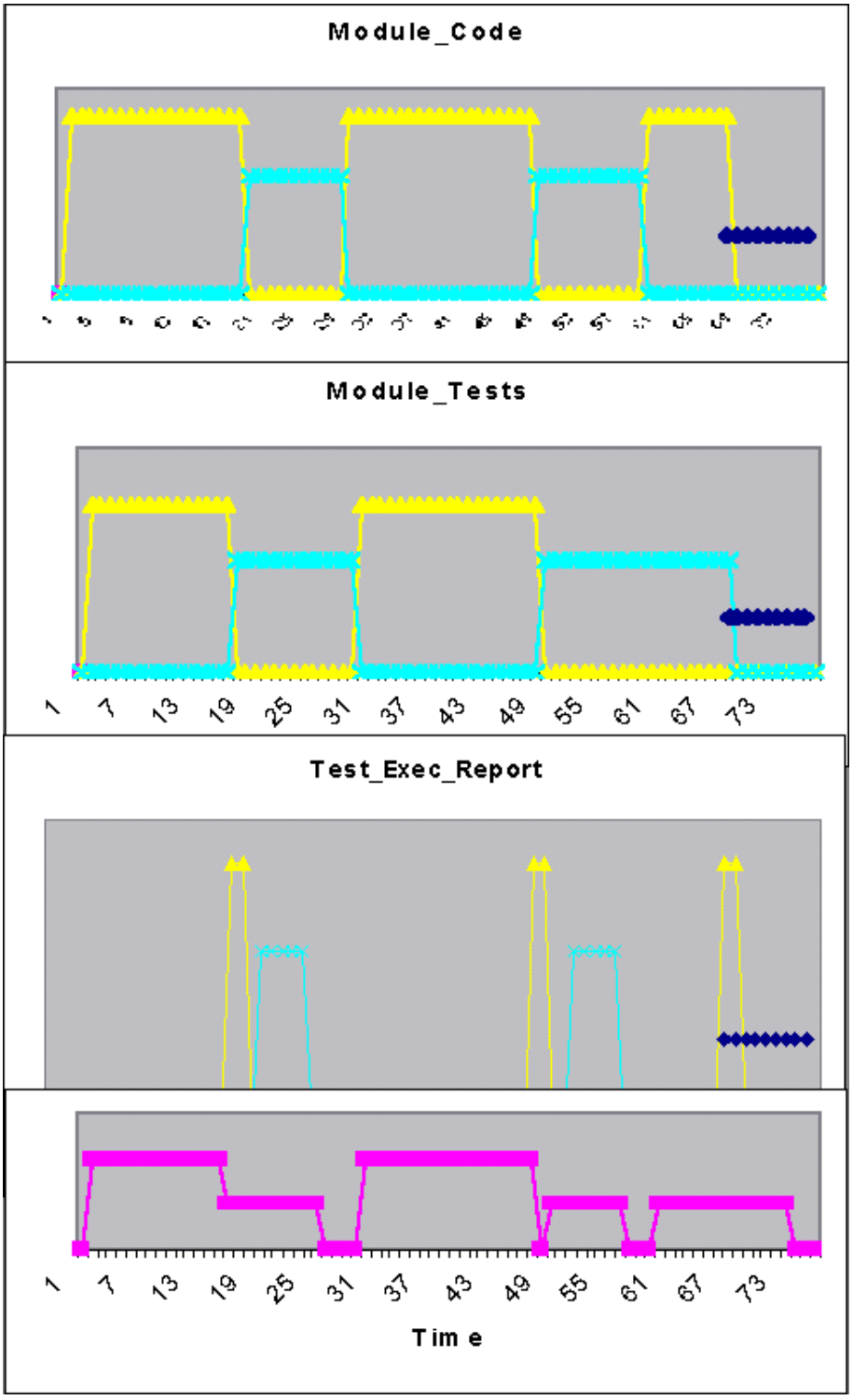

Figure 9.2: Traces of the software process model simulation

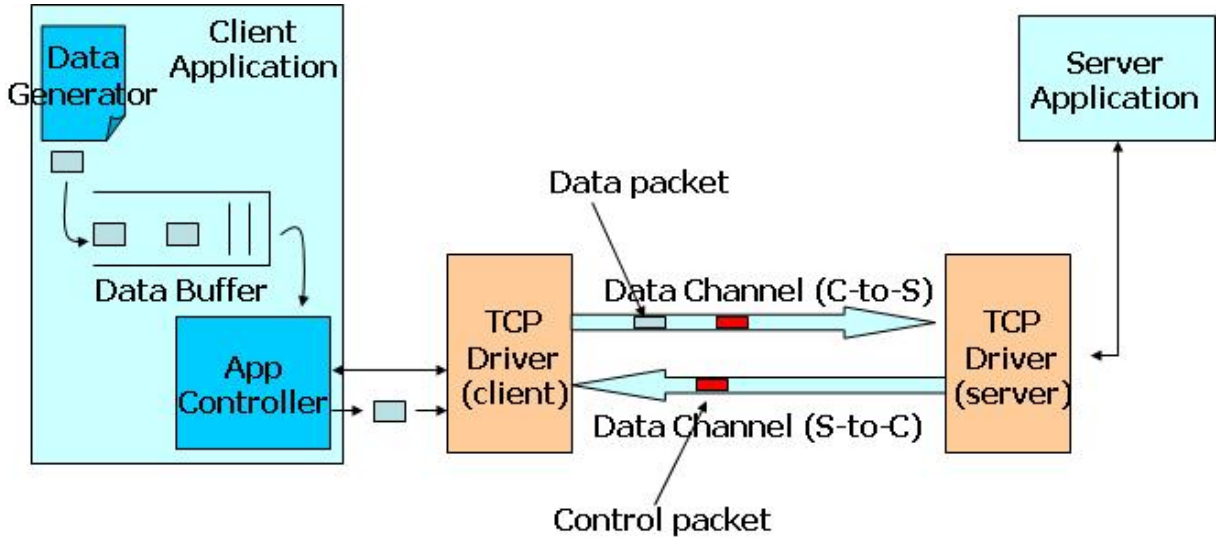

Figure 9.3: The TCP system

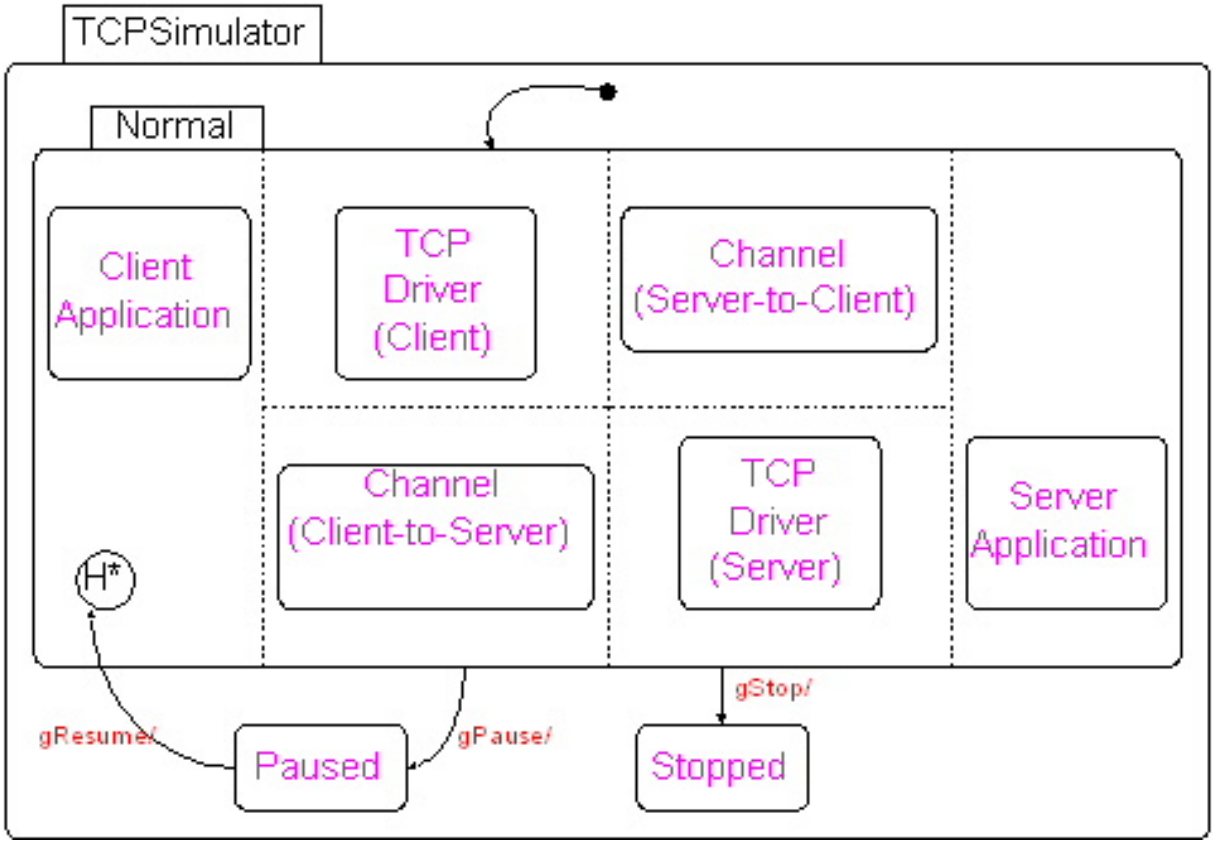

Figure 9.4: Overview of the TCP simulator

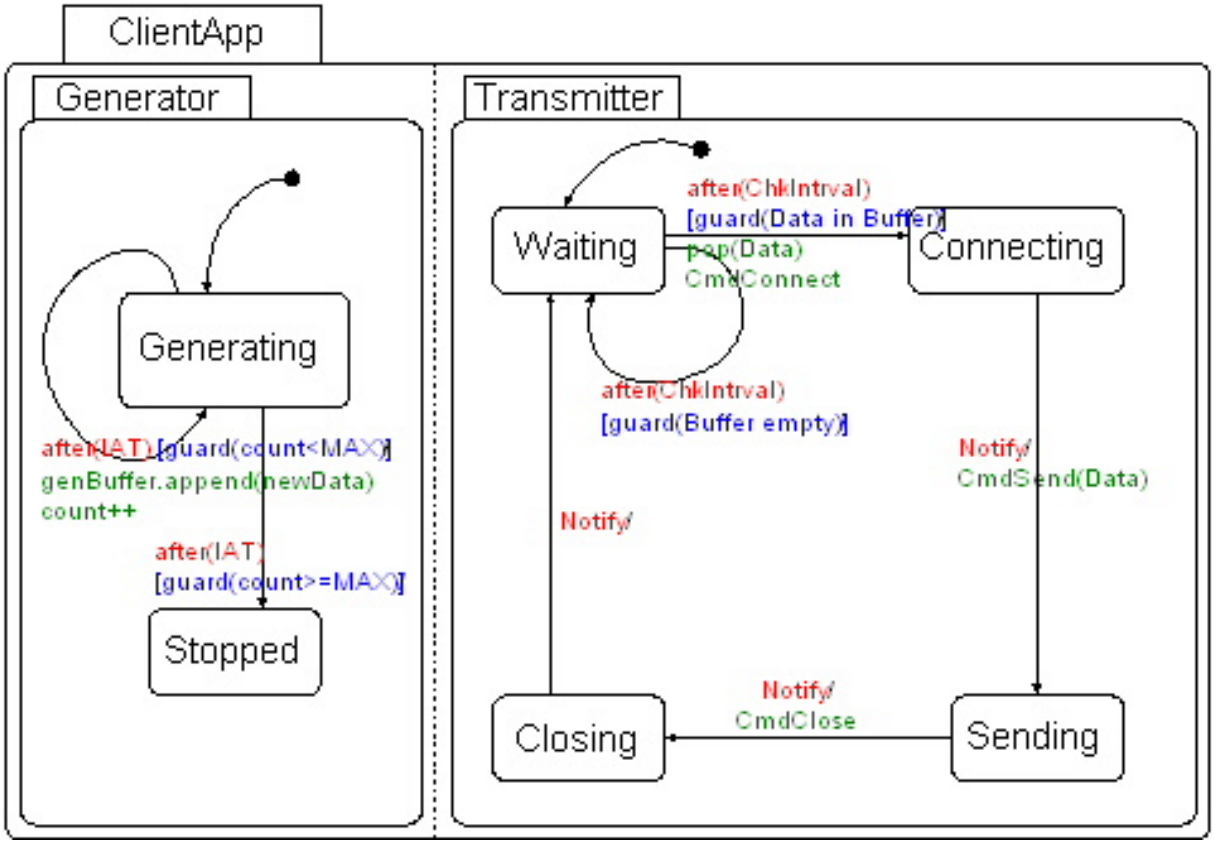

Figure 9.5: The submodel of the client application

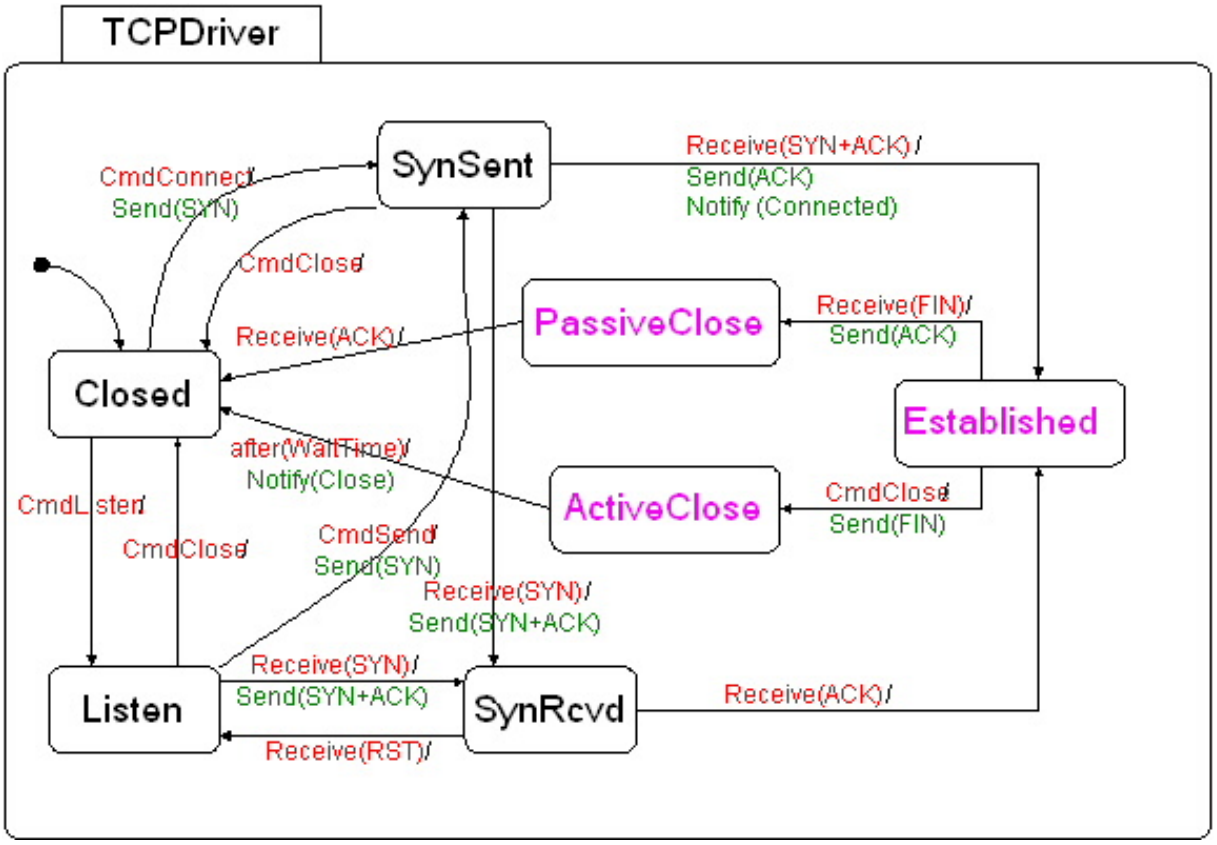

Figure 9.6: The submodel of the TCP driver (for both client side and server side)

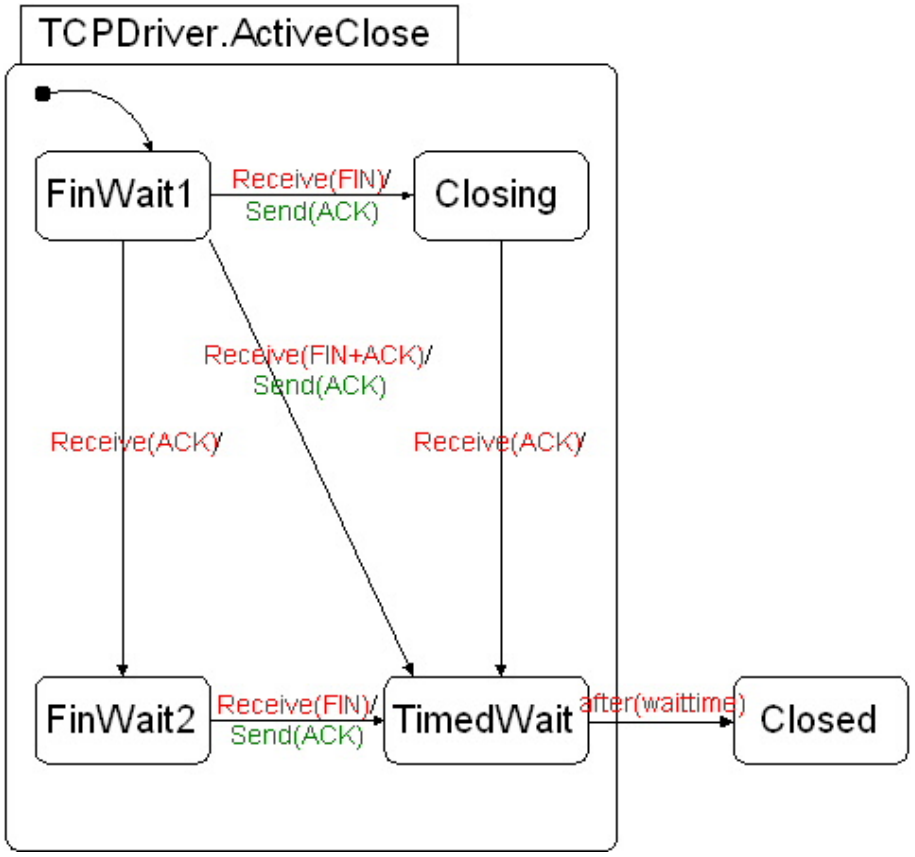

Figure 9.7: The `ActiveClose` state of the TCP driver

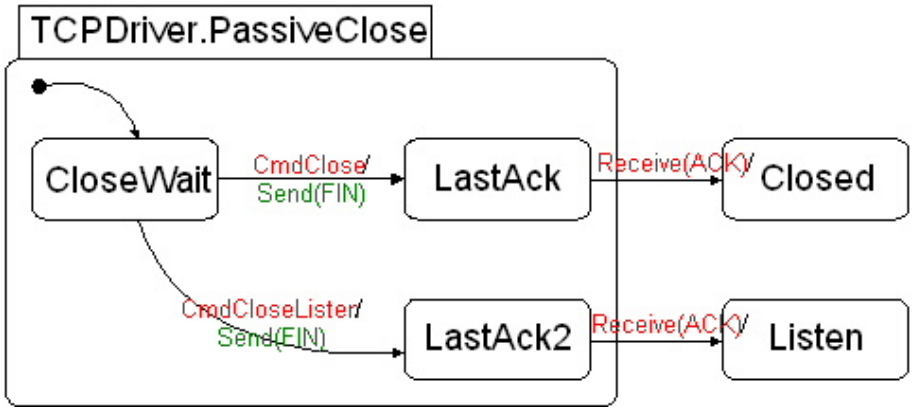

Figure 9.8: The `PassiveClose` state of the TCP driver

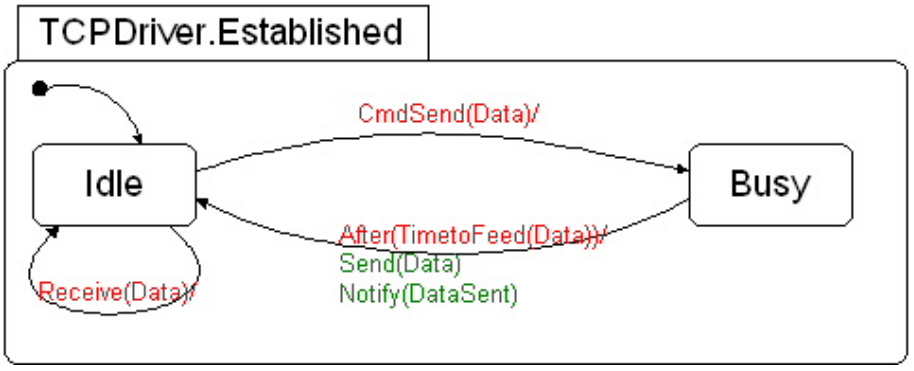

Figure 9.9: The `Established` state of the TCP driver

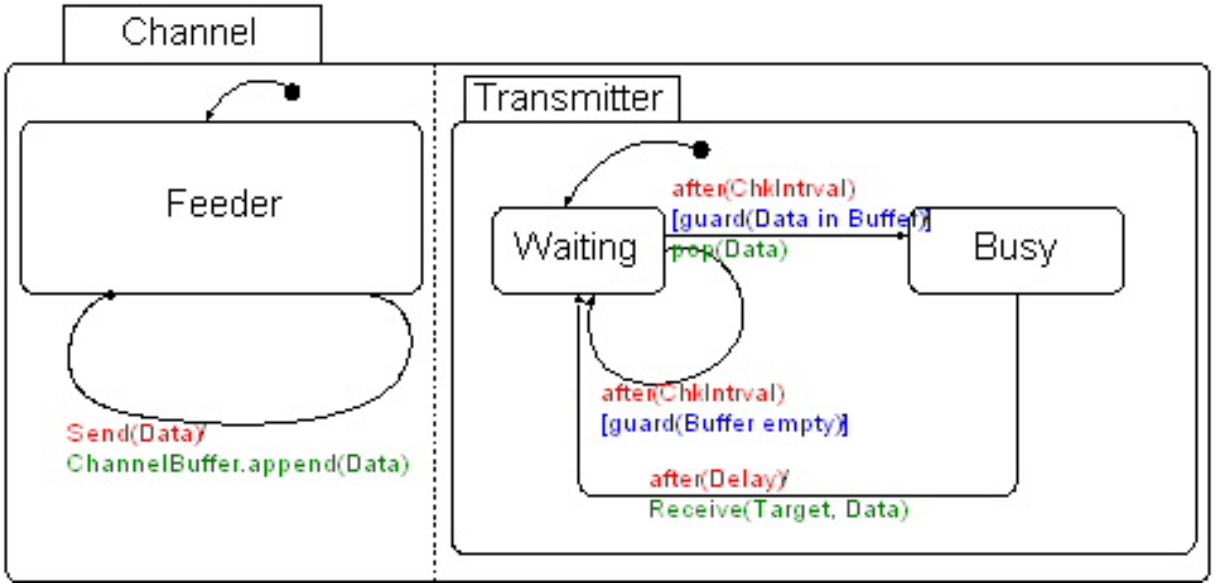

Figure 9.10: The submodel of the communication channel

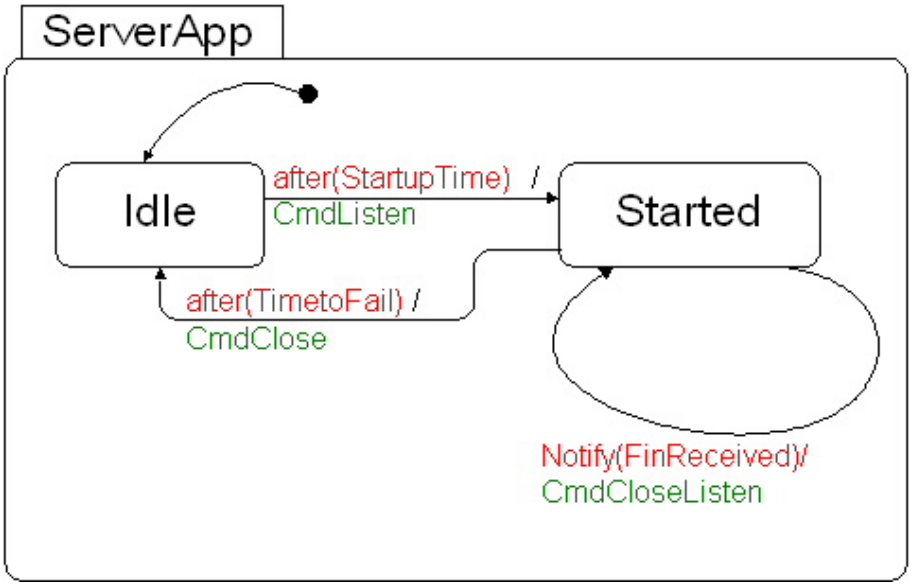

Figure 9.11: The submodel of the server application

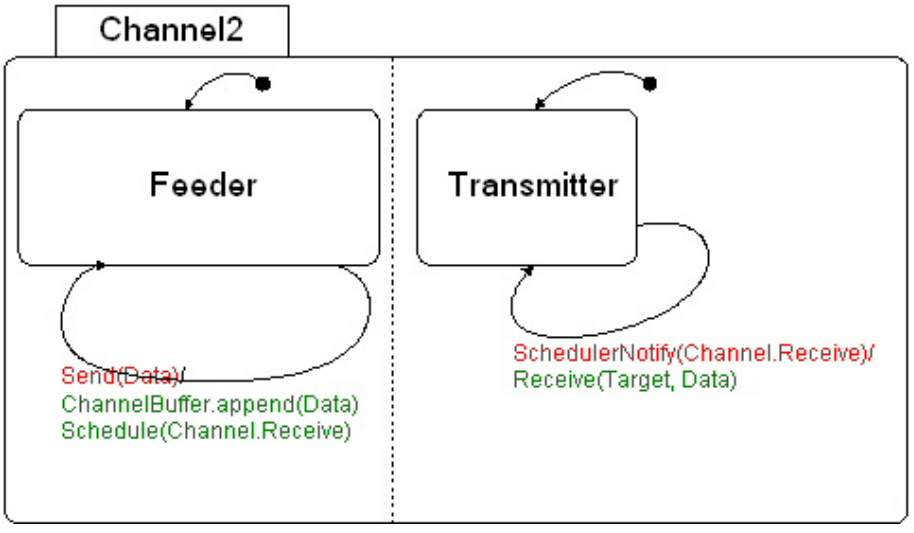

Figure 9.12: The virtual-time version of the communication channel

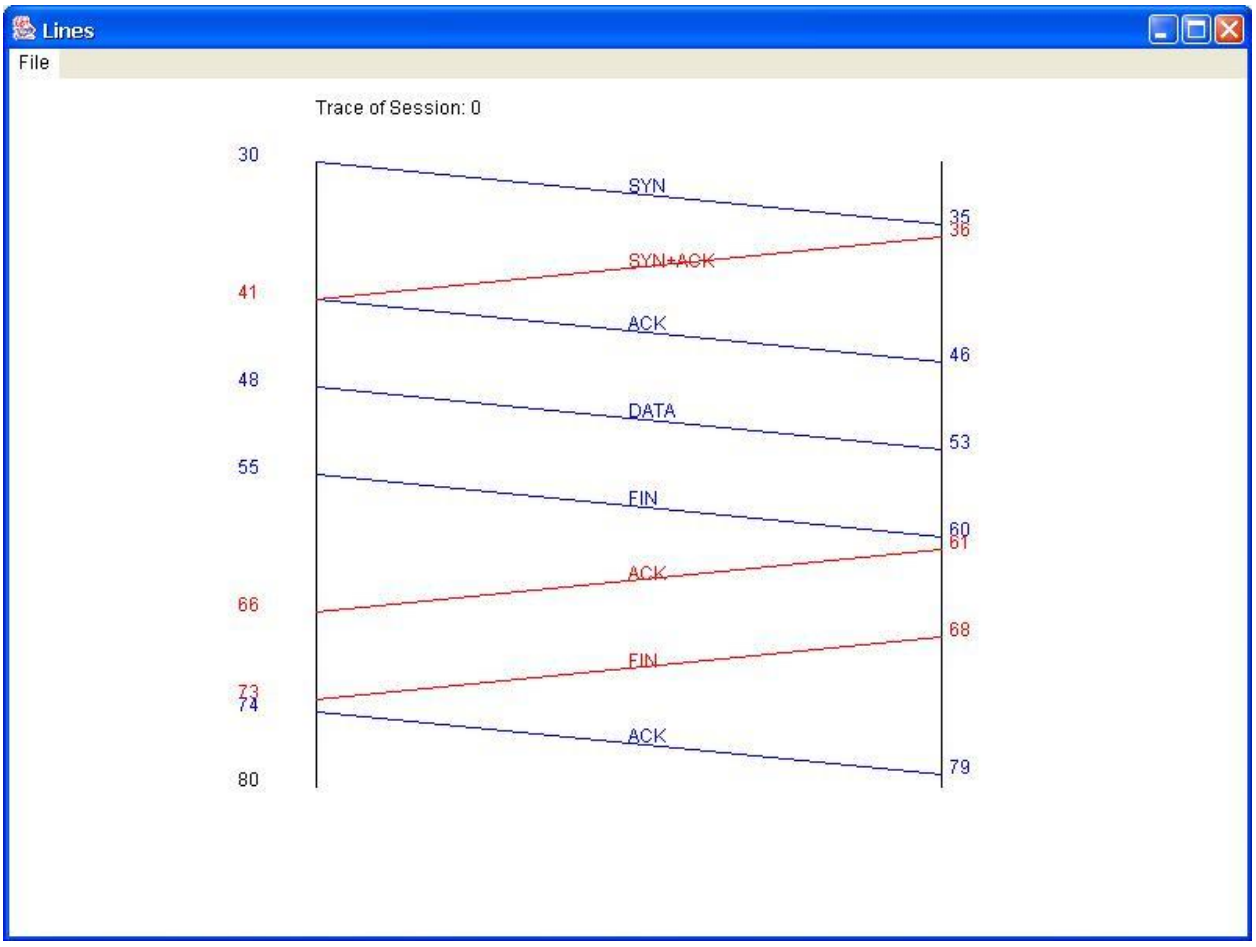

Figure 9.13: The plot of the simulation result of the TCP model

into a virtual-time model, Shah Asaduzzaman and Zaki Hasnain Patel have provided another version of the channel submodel `Channel2` (Figure 9.12). The clock component is imported as a top-level orthogonal component in the system. The new data channel schedules events by sending the `Schedule` event to the clock component. When the virtual time becomes equal to the scheduled time, the clock component sends back a `SchedulerNotify` event. (The `schedule` event and the `notify` event discussed in section 9.2 are renamed to `Schedule` and `SchedulerNotify`, respectively.)

The results of the simulation are gathered and plotted in Figure 9.13. For more information about the TCP model, the readers are referred to Shah Asaduzzaman's on-line report for the Modeling and Simulation course at McGill University:

`http://www.cs.mcgill.ca/~asad/archive/project-522/`

10 CONCLUSION

DCharts are a new formalism that combines the benefits of statecharts and DEVS for the design of complex physical systems and software systems. It has the following advantages:

- A visual syntax is designed for the DCharts formalism. There is graphical representation for every entity or feature of DCharts.
- DCharts are powerful. They support statecharts-like hierarchical model design with variables. Variables help keep infinite and innumerable states. Recursive DCharts models are much more expressive than statecharts and DEVS in that they are able to specify infinite states and transitions.
- DCharts are modular. Importation is also known as *tight coupling*. Submodels are copied to the inside of a state of the importing model. The behavior of the submodels may not be modified by the importing model, except that macros can be redefined as parameters.

  Connections between multiple models via ports are also known as *loose coupling*. In that case, a model may affect other models only by means of messages sent via the established connections.
- DCharts are independent of simulation strategies. Though its definition only addresses real-time simulation, it is shown that virtual-time simulation can be easily accomplished by means of a clock component.
- DCharts are highly practical. SVM is a simulator for DCharts, which supports a complete semantics of DCharts 1.0. Many of the algorithms implemented in SVM can be reused by other simulators or applications. SVM itself is reusable (for example, by AToM^3 and SCC).

  SCC is a code synthesizer for DCharts capable of generating source code in multiple target languages. The synthesized code is efficient and suitable for practical purposes.
- Besides these, non-recursive DCharts can be transformed into statecharts with variables or DEVS models. Statecharts and DEVS models can also be transformed into DCharts. This property is useful for model simulation and model checking.

Three types of syntaxes are discussed: abstract syntax, graphical/visual syntax and textual syntax. The mathematical syntax provides a means by which DCharts models can be formally specified. The graphical syntax represents DCharts models visually, which is much more easily understood by human beings. The textual syntax is accepted and processed by computer programs, while at the same time designers can still easily write DCharts models with the textual syntax. A few extensions to the basic syntax are proposed by the textual syntax. Those extensions allow designers to specify their models with more flexibility. They are supported by SVM and SCC.

The future work on DCharts includes:

- Do more research on model checking and verification of DCharts. There are two possible approaches:
  - Build tools that directly check DCharts models, or verify them by means of simulations.
  - Transform DCharts models into models in other well-studied formalisms, and check/verify the new models with the tools available for those formalisms.

- The performance of some DCharts features (such as history) in the code generated by SCC must be improved. Above this, an important hurdle to cross is the need for a target-language-independent action language.
- Extend the concept of ports and connections to tight coupling, so that an importing model may only send events to its submodels via ports and connections established between them. This mechanism further protects the internal behavior of submodels. It makes DCharts more modular.
- Implement the support of more target languages in SCC. Users will be able to integrate the code generated by SCC with the code generated by other code generators for other formalisms. This integration allows the users to model a system with different formalisms and tools, and finally combine different parts to get a complete system.

11 ACKNOWLEDGMENT

I have given beginning to the research on modeling and simulation with DCharts. The completion of this formalism requires and will require the collaboration and support from many researchers friends. At the very end of this thesis work, I would like to especially and sincerely thank these people:

- Professor Hans Vangheluwe of the MSDL (Modeling, Simulation and Design Lab) of McGill University, who has earnestly supervised my research from the very beginning, and who is still ardently supporting me with his learning, research equipments and morality;
- My parents and Ms. Wanmei Huang, my girlfriend, who have been supporting me by understanding the importance of my work and not asking for more time from me;
- Mr. Spencer Borland, whose research results, including the theory of transformation from statecharts to DEVS and the statecharts plugin for AToM^3 (which has been enhanced by me to become a DCharts plugin for AToM^3), have been the basis for my research;
- Professor Jörg Kienzle in the SEL (Software Engineering Lab) of McGill University, who always jokes with me and teaches me with his unique humorous tone;
- Ms. Sadaf Mustafiz, who has built the software process model for SVM, one of the most cited applications of my research result;
- Mr. Shah Asaduzzaman and Mr. Zaki Hasnain Patel, who have built the TCP model for SVM, another one of the most cited applications of my research result; and
- any one else who has unselfishly supported my research.

Index

Bibliography

[1] Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE)*, pages 174–188, April 2002. Grenoble, France.

[2] Juan de Lara and Hans Vangheluwe. Using atom^3 as a meta-case tool. In *4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649, 2002. Ciudad Real, Spain.

[3] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation*, 12(4):1–7, 2002. Special Issue Guest Editorial.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[5] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[6] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[7] Bernard P. Zeigler. *Multifacetted modelling and discrete event simulation*. Academic Press Professional, Inc., 1984.

[8] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Krieger Publishing Co., Inc., 1984.

[9] Spencer Borland. Transforming statechart models to DEVS. Master's thesis, School of Computer Science, McGill University, Montréal, Canada, August 2003.

[10] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[11] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package PythonDEVS for classical hierarchical DEVS. Technical report, MSDL, McGill University, June 2001. technical report MSDL-TR-2001-01.

[12] Ernesto Posse and Bolduc Jean-Sébastien. Generation of DEVS simulators by graph-transformation. In *Summer Computer Simulation Conference (Student Workshop)*, pages S139–S146. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[13] Spencer Borland and Hans Vangheluwe. Transforming statecharts to DEVS. In *Summer Computer Simulation Conference (Student Workshop)*, pages S154–S159. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[14] Alison Stewart. Modelling and simulation based design of GUI behaviour. Technical report, MSDL, McGill University, December 2003. `http://msdl.cs.mcgill.ca/people/astewa5/report.dtml`.

[15] C. Hylands, E. A. Lee, and et al. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical report, University of California, Berkeley, CA USA 94720, July 2003. Technical Memorandum UCB/ERL M03/27.

[16] C. Hylands, E. A. Lee, and et al. Heterogeneous concurrent modeling and design in java (volume 2: Ptolemy ii software architecture). Technical report, University of California, Berkeley, CA USA 94720, July 2003. Technical Memorandum UCB/ERL M03/27.

[17] C. Hylands, E. A. Lee, and et al. Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains). Technical report, University of California, Berkeley, CA USA 94720, July 2003. Technical Memorandum UCB/ERL M03/27.

[18] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights*, volume 131. Springer-Verlag, May 1981.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[20] Joanne M. Atlee and John Gannon. State-based model checking of event-driven systems requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[21] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Understanding and comparing model-based specification notations. In *IEEE International Requirements Engineering Conference (RE 2003)*, September 2003.

[22] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Composable semantics for model-based notations. In *Proceedings of the 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2002)*, 2002.

[23] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, pages 866–882, October 2002.

[24] Ivan Porres. Model refactorings as rule-based update transformations. In *Proceedings of the <<UML>> 2003 Conference*. LNCS 2863, Springer, October 2003. San Francisco, California, USA.

[25] Ivan Porres. A toolkit for model manipulation. *Journal on Software and System Modeling*, 2, 2003.

[26] Thomas Huining Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S147–S166. The Society for Computer Modelling and Simulation, July 2003. Montréal, Canada.

[27] Dániel Varró. A formal semantics of UML Statecharts by model transition systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 378–392, Barcelona, Spain, October 2002. Springer-Verlag.

[28] Foldoc (free on-line dictionary of computing), November 1997. `http://wombat.doc.ic.ac.uk/foldoc/`.

[29] Webopedia, 2004. `http://www.pcwebopedia.com/`.

[30] Msn encarta (online encyclopedia, dictionary, atlas, and homework), 2004. `http://encarta.msn.com/`.

[31] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley-Interscience, 2000.

[32] Stanley B. Lippman and Josée Lajoie. *C++ Primer*. EPUBCN.COM, 1998.

[33] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom^3. *Software and Systems Modeling (SoSyM)*, 2003.

[34] David M. Beazley. *Python Essential Reference (2nd Edition)*. New Riders Publishing, 2001.

[35] Mark Lutz. *Programming Python*. nh. O'Reilly & Associates, Inc., 1996.

[36] Mark Lutz and David Ascher. *Learning Python*. nh. O'Reilly & Associates, Inc., 1999.

[37] Fredirk Lundh. *The Standard Python Library*. PythonWare, 2000.

[38] Erich Gamma, Richard Helm, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[39] Thomas Huining Feng. SVM and SCC tutorial, March 2004. `http://msdl.cs.mcgill.ca/people/tfeng/svmsccdoc/`.

[40] Thomas Huining Feng and Hans Vangheluwe. Case study: Consistency problems in a UML model of a chat room. In *Sixth International Conference on the Unified Modelling Language (UML 2003), Workshop on Consistency Problems in UML-based Software Development II*, October 2003. San Francisco, USA. `http://msdl.cs.mcgill.ca/people/tfeng/docs/con03.pdf`.

[41] Python 2.2.3 documentation, May 2003. `http://www.python.org/doc/2.2.3/`.

[42] Juan de Lara and Hans Vangheluwe. Computer aided multi-paradigm modelling to process petri-nets and statecharts. In *International Conference on Graph Transformations (ICGT)*, volume 2505, pages 239–253. Springer-Verlag, October 2002. Barcelona, Spain.

[43] Juan de Lara Jaramillo, Hans Vangheluwe, and Manuel Alfonseca Moreno. Using meta-modelling and graph grammars to create modelling environments. In Paolo Bottoni and Mark Minas, editors, *Electronic Notes in Theoretical Computer Science*, volume 72, February 2003.

[44] Juan de Lara and Hans Vangheluwe. Using meta-modelling and graph grammars to process gpss models. In Hermann Meuth, editor, *16th European Simulation Multi-conference (ESM)*, pages 100–107, June 2002. Darmstadt, Germany.

[45] Ernesto Posse, Juan de Lara, and Hans Vangheluw. Processing causal block diagrams with graph-grammars in atom^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23–34, April 2002. Grenoble, France.

[46] Peter D. Mosses. Theory and practice of action semantics". In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113, pages 37–61. Springer-Verlag, 1996.

[47] Peter D. Mosses. Action semantics and asf+sdf. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[48] Peter Fritzson and Vadim Engelson. Modelica — A unified object-oriented language for system modeling and simulation. *Lecture Notes in Computer Science*, 1445, 1998.

[49] Hilding Elmqvist et. al. Modelica – A unified object-oriented language for physical systems modeling: Tutorial and rationale. Technical report, The Modelica Design Group, December 1999. `http://www.modelica.org/`.

[50] Watts S. Humphrey and Marc I. Kellner. Software process modeling: principles of entity process models. In *Proceedings of the 11th international conference on Software engineering*, pages 331–342. ACM Press, 1989.