SVM & SCC TUTORIAL

Thomas Huining Feng

April 5, 2004

Contents

1 INTRODUCTION

This document serves as a user manual for SVM (Statechart Virtual Machine) and SCC (StateChart Compiler) developed at the MSDL (Modeling, Simulation and Design Lab) of McGill University, Canada. SVM is a realtime simulator for DCharts (Design Charts), a formalism extended from David Harel's statecharts. SCC is a code synthesizer for DCharts.

For a detailed definition and description of the DCharts formalism and the general idea of SVM and SCC, the readers are referred to my master's thesis *DCharts, a Formalism for Modeling and Simulation Based Design of Reactive Software Systems*, finished under the supervision of Prof. Hans Vangheluwe at School of Computer Science, McGill University:

`http://msdl.cs.mcgill.ca/people/tfeng/thesis/`

The online version of this document is available at:

`http://msdl.cs.mcgill.ca/people/tfeng/svmsccdoc/`

1.1 Overview of DCharts

DCharts have their roots in David Harel's statecharts, which have been a part of UML now (with some modification). There are a number of variants of statecharts, mostly because the semantics of statecharts is not rigorously defined. This situation limits the reusability of statecharts models.

DCharts reuse all the concepts presented in statecharts. Extensions are added so that DCharts models are able to express models of all the statecharts variants. They can even be used to model the DCharts formalism itself. (This capability of meta-modeling is extremely interesting and cool!)

Some of the important extensions to original statecharts are summarized below. A complete description can be found in my master's thesis.

1.1.1 Transition Priorities

It is possible to assign different transition priorities to every transition in a DCharts model. Unlike the common approach of allowing for arbitrary priority numbers for transitions (which is not modular), DCharts support another scheme based on inner-first and outer-first transition ordering. The meaning of several related keywords is explained below:

- `ITF` (Inner Transition First). If this keyword is specified as a property of a state, transitions within the *scope* of that state are ordered by the inner-transition-first manner. A transition ordered before another

has higher *overall priority*.

Note: A transition is in the *scope* of a state iff the source state of the transition is that state or a substate of that state.

Note: In case of a conflict caused at run-time (i.e., two or more transitions are able to handle a single event and compete for it), the transition of the highest *overall priority* (or one of those transitions of the highest overall priority) is fired, while the other transitions are simply ignored.

- `OTF` (Outer Transition First). The inverse of `ITF`.
- `RTO` (Reverse Transition Ordering). A state with this property has reversed transition ordering compared to its parent state. Suppose states `A` is the parent of state `B`. If transitions in the scope of state `A` is ordered in an inner-transition-first way, transitions in the scope of state `B` is ordered outer-transition first. Vice versa.

  Note that transitions in the scope of state `B` are also in the scope of state `A`. The transition ordering of state `A` is considered before the ordering of state `B`, while the ordering of state `B overrides` the ordering of state `A` within the scope of `B`.

If none of the above properties is assigned to a state (the default case), the state inherits the transition ordering of its parent if any, or follows the `true` or `false` setting of the `InnerTransitionFirst` global option if it is a top-level state.

The `ITF` and `OTF` scheme solves most of the run-time conflicts, but not all those conflicts. For example, the conflict between two transitions with the same source state cannot be solved in this way. To solve this conflict, the designer is allowed to assign priority numbers (integers that can be negative) to the two transitions. For transitions ordered in the same place with the `ITF` and `OTF` scheme, smaller priority number always means higher overall priority. Priority numbers are not considered for conflicts that can be solved with the `ITF` and `OTF` scheme. By default, each transition has a priority number of 0.

If conflicts still exist at run-time, for example, two transitions with the same source state and the same priority number, the transition that is actually triggered is implementation-dependent. Designers should always avoid this circumstance.

1.1.2 Importation

A model designed in DCharts can also be used as a *component* (or *submodel*) of another larger model. When this happens, the designer of the larger model explicitly imports the component by placing it in one of the leaf states of the larger model (so that the leaf state becomes non-leaf). All the states in the component become substates of that leaf state (known as *importation state*), and transitions between those states defined in the component are preserved.

Importation is done dynamically. An imported model is loaded only when information about it is needed. For example, a transition from state `A` to state `B` is fired, where `B` is an importation state. If the imported model has not been loaded at that time, the simulator/executor loads it and does the importation dynamically. Once imported, the importation state becomes non-leaf and non-importation state. There is no means to delete the imported model from the model where it is imported, unless the user rolls the whole model back to a previous snapshot taken at the time when the imported model has not been loaded yet. Rollback support is optionally supported by the SVMDCP (SVM Distributed CheckPointing) sub-project, which currently only supports rollback of the Java code generated by SCC.

A *recursive DCharts model* imports itself directly of indirectly. A theoretically infinite state hierarchy is created in this way. In practice, dynamically importation is always done finite times in a single simulation or execution, so the state hierarchy of the run-time model is manageable.

Because of the closure under importation property of DCharts, it is always possible to flatten a non-recursive model with importation by statically importing all its components.

1.1.3 Macros and Importation Parameters

Macros are a textual feature of the DCharts description format supported by SVM and SCC. They are comparable to macros in programming languages like C. Once defined, the left-hand side of a macro can be used (between square brackets) in the textual description, which is literally substituted by the right-hand side before simulation/execution of the model.

The right-hand sides of some macros may use other macros, provided that the macros that they use are defined before them.

A model may specify parameters for its components by redefining their macros. For example, suppose macro `DEST` is defined to be `StateA` in component `C`. The textual description of `C` may contain such a segment:

```
MACRO:
    DEST = StateA
```

(Note: `MACRO` is a descriptor here, which starts the macro segment. Under this descriptor, one or more macros can be specified.)

In other parts of the description of `C`, `[DEST]` can be used to substitute `StateA`.

When `C` is imported into an importation state of model `M`, the macros defined in `C` may be redefined. A macro redefinition is written as a property of the importation state. It serves as the parameter given to the submodel at run-time before it is loaded. For example, the following is a part of the textual description of `M`, which imports `C` into importation state `S` and redefines the `DEST` macro of `C` to be `StateB`:

```
IMPORTATION:
    c_submodel = C.des

STATECHART:
    ...
    S [c_submodel] [DEST=StateB]
    ...
```

(Note: DCharts model descriptions are usually written in text files ending with postfix `.des`. `IMPORTATION` is the descriptor to start an importation definition segment. In this example, submodel `C.des` is imported and given ID `c_submodel`. This ID is used in the state hierarchy to refer to the submodel. The same ID may be used for multiple importation states, in which case the same component is imported into different states. `S` is defined to be an importation state because component ID `c_submodel` is given to it as a property. The macro redefinition is written as another property between square brackets.)

Macro redefinitions are not supported by SCC but only SVM. This is because SCC statically synthesizes code in a specific programming language for all the models and components. This is no means to dynamically modify their behavior.

1.2 Overview of SVM

SVM simulates DCharts models in real time.

SVM takes the following steps to process a model description, specified on the command-line:

1. Read in the model description;
2. 1st parse: search for all the macros defined in the model and store them in the memory (macros with the same names as the parameters specified by the importing model or by the user on the command-line are ignored);

3. 2nd parse: construct all the necessary data structures in the memory according to the model description under all the other descriptors (for example, the state hierarchy is built by literally combining and understanding the description under all the `STATECHART` descriptors, and a transition is constructed for each `TRANSITION` descriptor);

4. initialize the model by executing its *initializer* (an arbitrary piece of Python code written under the `INITIALIZER` descriptor, which is empty by default);

5. simulate the model by running its *interactor* (an arbitrary piece of Python code running in a separate thread to accept user inputs and interact with the model, as discussed later);

6. wait until the interactor finishes, which means the user stops the simulation, or the model itself ends the simulation;

7. finalize the model by executing its *finalizer* ((an arbitrary piece of Python code written under the `FINALIZER` descriptor, which is empty by default).

More information and releases of SVM can be found at its homepage:

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

1.3 Overview of SCC

SCC reuses the first parse and the second parse of SVM to construct the same internal structures. It synthesize code in multiple target languages from those internal structures. Currently it supports the following target languages: Java, C++, C# and Python.

The models are optimized before code synthesis. For example, an unique integer number is assigned to each event, which is used as the internal representation of the event instead of its string name. The names of states are also translated into integer numbers. This speeds up the execution by transforming string comparison to integer comparison.

Compared to the simulation in SVM, execution of the code generated by SCC is much more efficient. However, the code must be compiled by appropriate compiler for the target language before running. Because the code is statically generated, the flexibility of dynamically modifying its behavior by means of macro redefinition is lost.

1.4 License

The sources of SVM and SCC are provided to the public for free. However, on using these tools or redistributing them, the users must agree to the GNU GPL (General Public License) version 2 or later:

> *SVM and SCC are free software; you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.*
>
> *SVM and SCC are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.*
>
> *You should have received a copy of the GNU General Public License along with SVM or SCC; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA*

A complete description of the GPL can be acquired from:

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

2

SIMULATION IN SVM WITH EXAMPLES

This chapter provides a full explanation on the use of SVM. Example are given to demonstrate several important descriptors.

2.1 System Requirements

SVM is written in Python. It should be possible to use SVM on all the platforms and operating systems that support the Python language. However, some features of SVM requires the support from platform-dependent libraries. They will be highlighted when encountered in the following text. Despite this limitation, all the features are tested on RedHat Linux 9.0 and the Windows family.

Other requirements:

- Python (`http://www.python.org`) 2.2 or higher. Because Python is an interpreted language, the Python environment is required for the execution of SVM.
- PyGame (`http://www.pygame.org/`), the gaming library for the Python language, is required for the CD player demo and the MP3 player demo, which are distributed with SVM (found in the `CDPlayer` directory and the `MP3Player` directory, respectively). It is not a requirement for SVM itself.
- PVM (Parallel Virtual Machine, `http://www.netlib.org/pvm3/`) is required for distributed simulations with SVM. However, it is not required for local simulations.
- Cygwin (`http://www.cygwin.com/`), "a Linux-like environment for Windows", is required if the user uses Windows and wants to receive the support for distributed simulations. Tools in Cygwin, including GNU Make and GCC, are required to build the PYPVM (`http://pypvm.sourceforge.net/`) that interfaces Python code with the native PVM library for C.

2.2 Installation

The first step is to obtain SVM. The latest release can be downloaded from its website:

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

The `.tar.gz` packages are for Linux. Users may use the following command to extract those packages:

```
tar zxvf svm-xxxxx.tar.gz
```

(Here, xxxxx is the version number followed by the architecture, or src if the package is a source package.) A new directory will be created with SVM in it.

The .zip packages are for Windows. They can be unpacked with WinZip (http://www.winzip.com/) or WinRAR (http://www.rarlab.com/).

The users who want to experience the most cutting-edge features of SVM may download it from the CVS (Concurrent Versions System) server with the following commands (executed in Linux systems or in Cygwin in Windows):

```
export CVS_RSH="ssh"
cvs -z3 -d:ext:anoncvs@savannah.nongnu.org:/cvsroot/svm co svm
```

The CVS server is hosted by Savannah (https://savannah.nongnu.org/). SVM's project home on Savannah is at:

https://savannah.nongnu.org/projects/svm

Windows users are strongly advised to use binary distributions. Those distributions build in all the necessary libraries and do not require the Python environment. The users simply unpack the packages and execute svm.exe or scc.exe in it.

SVM developers or users of other operating systems (Linux, FreeBSD, SunOS, etc.) may download the source packages. svm.py and scc.py in the packages contain the main program of SVM and SCC, respectively. They are executable in the Python environment. For example, the following command starts SVM with no parameter. The command-line help will be printed to the console.

```
python svm.py
```

(Support svm.py is in the current directory.)

Start-up scripts svm and scc in the same directory as svm.py and scc.py are written for Linux users. By adding the SVM directory to the PATH environment variable, the users can invoke SVM and SCC simply by running svm and scc from any directory.

Similarly, scripts svm.bat and scc.bat are written for Windows users to invoke SVM and SCC from any directory.

If source packages or CVS versions are used, the users must manually build the PYPVM library before distributed simulation can be performed:

- Linux users.

  Execute "make pypvm" on the command-line in the SVM directory. This will build the PYPVM library with SCC and copy it to the SVM directory. Before executing the command, the users MUST confirm the following environment variables are set correctly:

  1. PVM_ROOT should be set to the path where PVM is installed. For RedHat Linux 9.0, this path is /usr/share/pvm3. It may be different for other Linux distributions.
  2. PVM_ARCH should be set to the ID of the architecture. This ID is equal to the name of the sub-directory of $PVM_ROOT/lib. Use the following command to retrieve this ID:

     ```
     ls -F $PVM_ROOT/lib | grep '/' | gawk 'sub(/\//, "", $0)'
     ```

- Windows users.

  The PYPVM library is prebuilt for Windows users and saved in the lib\win32\Release sub-directory of the SVM directory. Command "make -f Makefile.win32 pypvm" copies the library to the SVM directory.

2.3 HelloWorld Example

The design and simulation of a HelloWorld example is discussed in this section. This model has two states A (default state) and B (final state). When simulation is started, only event e is accepted, which changes the model from state A to state B, and prints a "Hello World!" message.

To design a textual model manually, the designer writes the description according to the DCharts textual syntax (see my master's thesis) and saves it in a .des file. For this example, the textual description is listed below:

```
DESCRIPTION: # description of the model
    Hello World Example

STATECHART:  # definition of the state hierarchy
    A [DS]
    B [FS]

TRANSITION:  # one single transition
    S: A
    N: B
    E: e
    O: [DUMP("Hello World!")]
```

This description is saved as HelloWorld.des. Several important points about this description are explained below:

- Three descriptors are used in this example. Descriptor DESCRIPTION starts a segment of description of the model (usually in natural language). Empty lines or lines with only spaces or tabs are removed automatically. If a model has multiple description segments, they are literally combined to become one single description. The default SVM interface prints this description on the screen when the simulation is started.

- Descriptor STATECHART starts the definition of the state hierarchy. If a model has several such segments, they are literally combined before the simulator understands the state hierarchy of the model. In this example, A and B are the two top-level states. They are defined in separate lines and aligned left. Substates of a state (if any) should appear in lines immediately under its definition, with more leading spaces. Sibling states are always aligned left. (Examples of more complex state hierarchies are discussed later.)

  Property [DS] is assigned to state A, which means it is a default state. Similarly, property [FS] of state B means it is a final state.

- Descriptor TRANSITION starts the definition of one single transition. In this example, the transition has the following properties:

  1. S property specifies the source state of the transition.
  2. N property specifies the destination state of the transition.
  3. E property specifies the event that triggers the transition.
  4. O property specifies a list of commands to be executed as the output of the transition. [DUMP(...)] is a predefined macro that prints a message.

- Comments are placed to the right of a "#" mark until the end of the line. They are similar to the C-style comments behind "//".

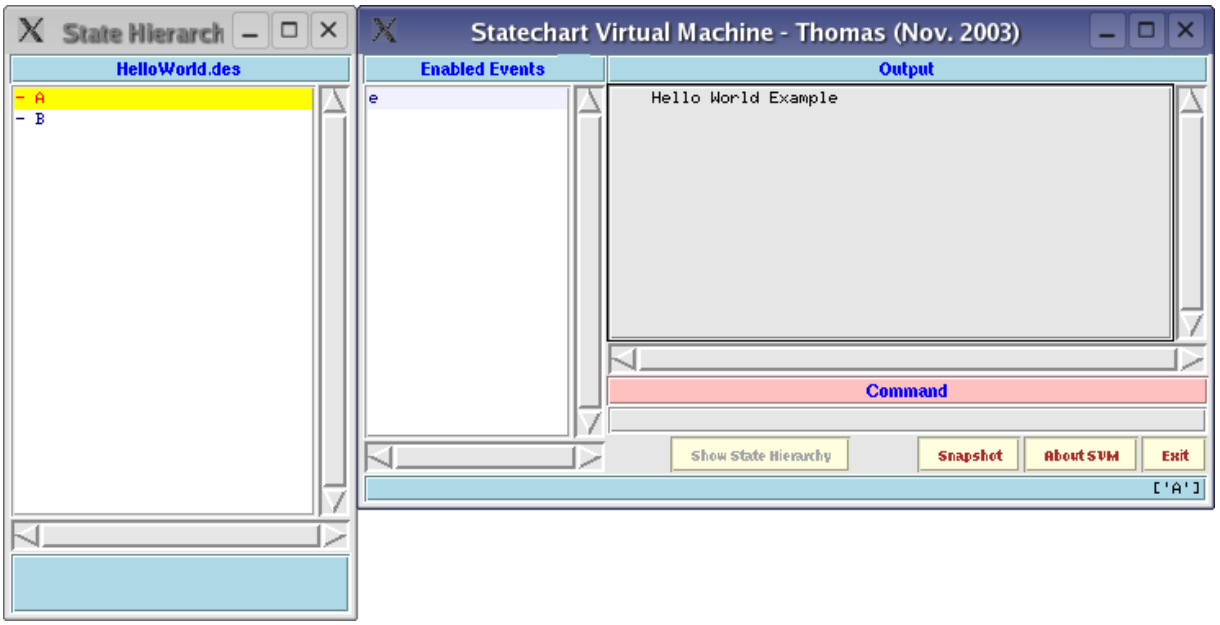

Figure 2.1: Screenshot 1 of the `HelloWorld` example

To run the model, first make sure that `HelloWorld.des` is in the current directory and the SVM directory is added to the `PATH` environment variable. Execute the following command:

```
svm HelloWorld.des
```

The default graphical interface (Figure 2.1) will be shown, and the simulation is started.

First-time users may not be familiar with this default graphical interface, and brief explanation may be required. There are two separate windows: the right one is the main window, and the left one is the state hierarchy window.

- Main window:

  1. The left panel of the main window lists all the currently enabled events. When simulation of the above example is started, event `e` is enabled because, if it is received, it triggers the only transition in the model. There are no other enabled events.
  2. The output box is the area where messages from the model is printed. In this case, when the simulation is started, the description is printed automatically to this area.
  3. The command box receive interactive commands from the user. During a simulation, three types of commands are acceptable:
     (a) Enabled events (as are listed in the left panel);
     (b) `debug` command, which switches the simulator to the debug mode (discussed later);
     (c) `exit` command, which forces the simulation to stop immediately and closes the SVM windows.

     An unaccepted command entered by the user is simply ignored.
  4. The “Show State Hierarchy” button is initially disabled. If the user closes the state hierarchy window, it becomes enabled and clicking on it brings up the state hierarchy window again. (Closing the main window itself has the same effect as typing the `exit` command in the command box.)

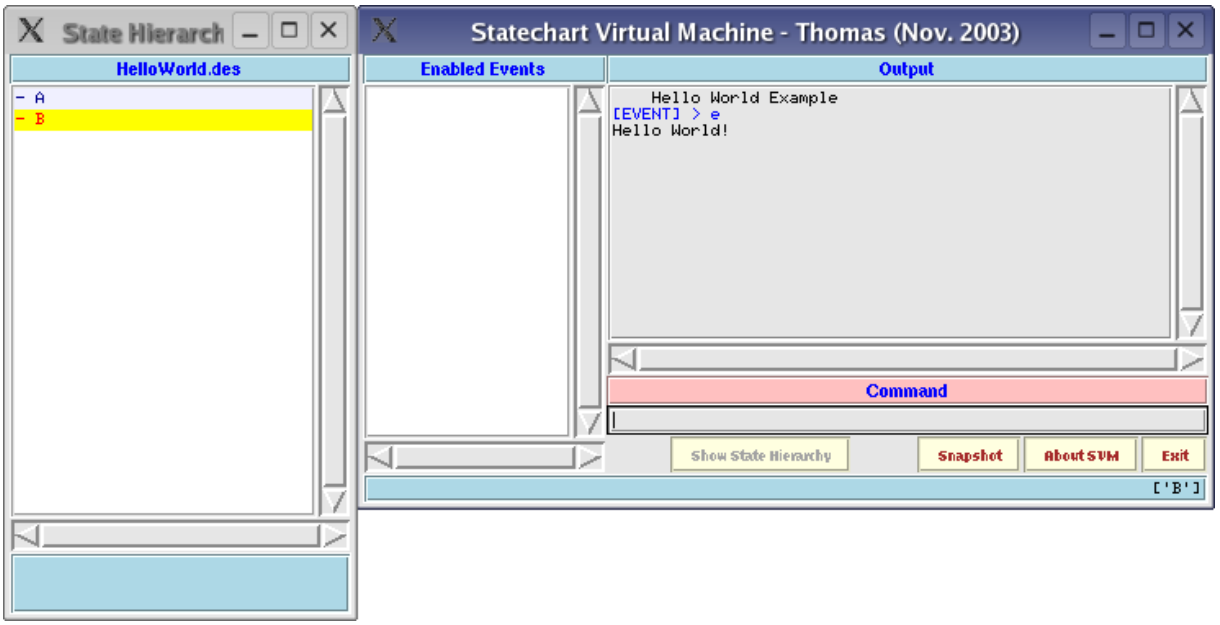

Figure 2.2: Screenshot 2 of the `HelloWorld` example

5. The “Snapshot” button takes a snapshot of the current state of the simulation. The snapshot is saved in a text file, which can be used to resume the simulation at the current state. Snapshotting is discussed later.
6. The “About SVM” button brings up the about dialog.
7. The “Exit” button provides the same functionality as the `exit` command.
8. The status bar at the bottom always shows the current state of the model (in the format of Python list).

- State hierarchy window:

1. The top bar shows the file name of the current model.
2. The state list shows the state hierarchy in a tree style. Leaf states which the model is current in are highlighted with yellow background.
3. The bottom box shows the *full path* of the currently selected state in the state list. The full path of a state consists of the names of all its superstates and its own name in the order of their levels. Those names are separated by a dot. For example, the full path of state `A` in this example is `A`. If `A1` is a child of `A`, the full path of `A1` is `A.A1`.

   If no state is selected in the state list (as in Figure 2.1), this box is empty.

To send event `e` to the `HelloWorld` model, the user should either use the left mouse button to double click the event name in the enabled event list, or type in `e` in the command box and then enter. This event triggers the only transition defined in the model, and message “Hello World!” is printed in the output box (Figure 2.2).

Now, because state `B` (the current state) is a final state, there is no enabled event any more, and the enabled event list becomes empty. To close SVM, the user should either enter command `exit` in the command box, or click the “exit” button with the left mouse button, or close the main window.

2.4 SVM Command-line Parameters

When executed without any command-line parameter, SVM prints out the following messages to the console:

```
============================================================
|  Python Implementation of Statechart Virtual Machine  |
|                          Version 0.3                   |
|              Presented by Thomas Feng, Nov. 2003       |
============================================================

Usage: svm [options...] <.des|.snp file> [parameters...]
       options:
           -c: force curses interface (Linux)
           -t: force textual interface
           -i <file>: include a file (to the head)
           -I <file>: include a file (to the tail)
       parameters:
           "name=value"
```

The meaning of the accepted command-line parameters is explained below:

- The `-c` parameter enforces the curses interface (if the simulated model does not provide a model-specific interface). Curses is a colorful textual library for Unix-based systems. It is not supported in Windows.

  For example, the users may run the `HelloWorld` example with the following command:

  ```
  svm -c HelloWorld.des
  ```

- The `-t` parameter enforces the textual interface (if the simulated model does not provide a model-specific interface). This interface is suitable for most systems and machines, even if the Tk library is not supported or no graphical device is available. It is also more efficient than the default graphical interface.

  For example, the users may run the `HelloWorld` example with the following command:

  ```
  svm -t HelloWorld.des
  ```

- The `-i` parameter literally inserts a text file to the head of the model description. This option is very useful when the designer designs a group of models and wants them to share a common part of description (for example, the interactor or the initializer). In that case, the designer writes the common part in a separate text file, and inserts the file to the model descriptions at the time of simulation.

  Multiple text files can be inserted with multiple `-i` parameters on the command-line. For example, `a1.des` contains the following content:

  ```
  STATECHART:
  ```

  `a2.des` contains the following content:

  ```
      A [DS]
  ```

  And `a3.des` contains the following content:

```
    B
```

To combine those descriptions and simulate the model in SVM, the user should execute the following command:

```
svm -i a1.des -i a2.des a3.des
```

(Note: there must be exactly one *main file* on the command-line which is not preceded by the `-i` parameter, in this case, `a3.des`.) The resulting model is equivalent to the following:

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

(It is a model with two states but without transition.)

- The `-I` parameter is similar to `-i`, except that the file is appended to the end of the model description. As a result, the model discussed above can also be simulated with the following command:

  ```
  svm -I a2.des -I a3.des a1.des
  ```

- The main file follows all the `-i` parameters and `-I` parameters (if any). After the main file, the user may specify one or more macro redefinitions. Those macro redefinitions override the macros defined in the model, but they do not affect the submodels that are imported into the main model (whose description is the combination of all the text files that appear in the command-line).

  For example, `DUMP` is a predefined macro (discussed later) used in the `HelloWorld` example. The example prints "Hello World!" when event `e` is received. To override this macro on the SVM command-line, the user may execute the following command:

  ```
  svm -t HelloWorld.des 'DUMP(msg)=print "example:", [msg]'
  ```

  When event `e` is received, instead of printing "Hello World!" the model prints "example: Hello World!"

2.5 More Examples

Several examples are discussed in this section. Each of them is designed to introduce one or more descriptors. For a complete list and definition of these description, the readers are referred to my master's thesis.

2.5.1 An example of macros

Macros are defined under the `MACRO` descriptor. The descriptor is followed by the macros, each of which is written in a separate line. There may be multiple `MACRO` descriptors in the same text file.

Sometimes the definition of a macro is too long to be understandable in a single line. In that case, the designer may use a line break "\" at the end of a line to connect it with the next line (preceding spaces and tabs are preserved in the next line).

`macro1.des` demonstrates the use of macros:

```
MACRO:
    NEWSTATE = B
    DUMP_STATE(state) = print "Current state:", "[state]"
```

```
STATECHART:
    A [DS]
    B
    C

TRANSITION:
    S: A
    N: [NEWSTATE]
    E: e
    O: [DUMP_STATE([NEWSTATE])]
```

In this example, two macros are defined: `NEWSTATE` is defined to be `B` and `DUMP_STATE` is defined to dump the current state to the console. Once defined, the macros can be used throughout the model description, with their names placed in square brackets.

In the definition of `DUMP_STATE`, `print` is a Python command and prints a message to the console. This macro has a parameter `state`. In the left-hand side, the parameter is placed between parentheses. In the right-hand side, the value of the parameter can be referred to with the name of the parameter between square brackets. Designers must keep in mind that parameters are literally substituted by their values when they are used in other parts of the model description. For example, in this case, `[states]` is substituted with `[NEWSTATE]` in the output of the transition. `[NEWSTATE]` refers to the value of macro `NEWSTATE`, which is `B`. Because `[state]` is substituted by `B` in the right-hand side, to print out this current state as a Python string, `[state]` must be placed between quotes.

To execute this example in textual mode, use the following command:

```
svm -t macro1.des
```

Here is the result:

```
['A'] > e
Current state: B
['B'] > exit
```

2.5.2 An example of history states

History states have `[HS]` or `[HS*]` properties following their names in the state hierarchy. A state with `[HS*]` property is a deep history state. A state with `[HS]` property but no `[HS*]` property has a normal history (or shallow history). When both `[HS*]` and `[HS]` are given to a state, `[HS]` is ignored.

A transition to the (deep or normal) history of a state has `[HS]` property to distinguish with a transition to the state itself. The property is written behind the `TRANSITION` descriptor of that transition.

The following is the description of a model with a history state and a deep history state (`history1.des`):

```
STATECHART:
    A [DS] [HS]
      A1 [DS]
        A11 [DS]
        A12
      A2
    B [HS*]
      B1 [DS]
        B11 [DS]
        B12
```

```
      B2

TRANSITION:
    S: A.A1.A11
    N: A.A1.A12
    E: A11-A12

TRANSITION: [HS]
    S: A
    N: B
    E: A-B

TRANSITION:
    S: B.B1.B11
    N: B.B1.B12
    E: B11-B12

TRANSITION: [HS]
    S: B
    N: A
    E: B-A
```

The result of its simulation is:

```
['A.A1.A11'] > A11-A12
['A.A1.A12'] > A-B
['B.B1.B11'] > B11-B12
['B.B1.B12'] > B-A
['A.A1.A11'] > A-B
['B.B1.B12'] > exit
```

Here, `A` has a normal history while `B` has a deep history. The first event `A11-A12` changes the model to state `A.A1.A12`, which is not its default state. The transition triggered by event `A-B` has `[HS]` property, so it changes the model to the history of state `B`. Initially, `B`'s history is empty, so the model transitions to the default substate of `B` (which is `B.B1.B11`). Event `B11-B12` changes the model to state `B.B1.B12`. When event `B-A` is received, the model goes back to the history of `A`. The history was recorded at the time when the model left state `A`. At that time, the model was in state `A.A1.A12`. However, because the history of `A` is normal history, it only records the child of `A` that the model was in (`A.A1`). As a result, the model changes the default substate of `A.A1`, which is `A.A1.A11`. When event `A-B` is received at this time, the model goes to the deep history of `B`, so the new state before exiting is `B.B1.B12`.

2.5.3 An example of orthogonal components

Orthogonal components are states with `[CS]` property (short for Concurrent State). According to the definition of orthogonal components, if a state is an orthogonal component, all its siblings (other states of the same parent) must also be orthogonal components. Orthogonal components must also be default children of their parent, so they must have `[DS]` property.

An example of orthogonal components (`orthogonal1.des`) is included below:

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

```
      B1 [CS] [DS]
        B11 [DS]
        B12
          B121 [CS] [DS]
          B122 [CS] [DS]
      B2 [CS] [DS]
      B3 [CS] [DS]

TRANSITION:
    S: A
    N: B
    E: e

TRANSITION:
    S: B.B1.B11
    N: B.B1.B12
    E: f
```

The result of the simulation of this model in text model is:

```
['A'] > e
['B.B1.B11', 'B.B2', 'B.B3'] > f
['B.B2', 'B.B3', 'B.B1.B12.B121', 'B.B1.B12.B122'] > exit
```

In this example, `B1`, `B2` and `B3` are siblings because they are all children of state `B`. They are orthogonal components with `[CS]` property and `[DS]` property. When the model goes to state `B` because of event `e`, it is concurrently in the three states: `B.B1.B11` (the default substate of `B.B1`), `B.B2` and `B.B3`. The current state is printed to the screen as a Python list: `['B.B1.B11', 'B.B2', 'B.B3']`. (The sequence of the components in this list is unimportant.) When event `f` is received, the model goes to state `B.B1.B12`. Orthogonal components `B.B2` and `B.B3` are unchanged. `B.B1.B12` has two orthogonal components as its children. As a result, there are 4 components in the current state: `['B.B2', 'B.B3', 'B.B1.B12.B121', 'B.B1.B12.B122']`.

2.5.4 An example of enter/exit actions

Enter actions of a state are written under the `ENTER` descriptor. Exit actions are written under the `EXIT` descriptor. There may be multiple `ENTER` or `EXIT` descriptors.

The `N` (new state) property of an `ENTER` descriptor specifies the state that is entered. The `S` (source state) property of an `EXIT` descriptor specifies the state that is exited.

The following example (`enterexit1.des`) demonstrates the use of enter actions and exit actions.

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

ENTER:
    N: A
    O: [DUMP("enter A")]

EXIT:
    S: A
    O: [DUMP("exit A")]
```

```
ENTER:
    N: B
    O: [DUMP("enter B")]

TRANSITION:
    S: A
    N: B
    E: e
    O: [DUMP("transition from A to B")]
```

The result of the simulation is:

```
enter A
['A'] > e
transition from A to B
exit A
enter B
['B'] > exit
```

In this example, enter actions are defined for states `A` and `B` to print out simple messages. An exit action is also defined for state `A`. A transition changes the model from state `A` to state `B`. In its output (`O` property), a message is also printed.

From the output, one may notice that output of the transition that causes the state change is executed before exit actions, and exit actions are before enter actions. This semantics of DCharts conforms to the original statecharts semantics proposed by David Harel. In addition to this fact, in one single transition, exit actions of a state are always executed before the exit actions of superstates of that state. Conversely, enter actions of a state are always executed after the enter actions of superstates of that state.

A global option "`Harel`" specifies whether the semantics of a model strictly conforms to the David Harel's semantics. By default it is set to `true`. A designer may override this option to `false` to use the alternative semantics. (This option is discussed later.)

2.5.5 An example of guards

Designers may optionally specify guards (boolean conditions) for transitions and enter/exit actions. The guard of a transition is tested when the event of the transition is received. The transition is triggered only if the result of this testing is `true`. Similarly, when the model enters or exits a state and if corresponding enter/exit actions are defined, they are executed only if the guards are evaluated to `true`.

The following example (`guard1.des`) demonstrates the use of guards:

```
STATECHART:
    A [DS]
      A1 [DS]
      A2
      A3
    B

ENTER:
    N: A
    C: 1==1
    O: [DUMP("enter A")]
```

```
EXIT:
    S: A
    C: 1==0
    O: [DUMP("exit A")]

TRANSITION:
    S: A
    N: B
    E: e
    C: [INSTATE("A.A1")] or [INSTATE("A.A2")]
    O: [DUMP("transition to B")]
```

The result of the simulation is:

```
enter A
['A.A1'] > e
transition to B
['B'] > exit
```

When state `A` is entered initially, the guard (`1==1`) of its enter action is tested. Since the result is `true`, the action is executed and "enter A" is printed. When the transition is triggered, its guard is tested. Predefined macro `INSTATE` is used to test whether the model is in state `A.A1` of `A.A2`. The result is `true`, so the output of the transition is also executed. The exit action of state `A` is not executed because its guard is evaluated to `false`.

2.5.6 An example of an initializer and a finalizer

Each model may have an initializer defined under the `INITIALIZER`. Designers are allowed to write arbitrary Python code in the initializer. This code is executed at the very beginning of the simulation, even before the simulator finds the default states of the model or the enter actions of those default states are executed. If a model has multiple `INITIALIZER` descriptors, the code under those descriptors is literally combined.

Similarly, the finalizer is defined under the `FINALIZER` descriptor. It is a piece of arbitrary Python code. If multiple `FINALIZER` descriptors are defined in the model description, they are literally combined. The finalizer is executed at the very end of the simulation, after the execution of enter actions of the last entered state. The end of a simulation is defined as one of the following two events:

1. A final state of the model is entered and the final state is a leaf state. If it is not a leaf state, the simulation ends when all the default leaf substates of that state are entered. (I.e., the simulation of the model finishes a macro-step.)
2. The user stops the simulation manually. This is usually by closing the SVM main window or entering the `exit` command in the command box (or text console).

An example of an initializer and a finalizer (`initfinal1.des`) is included below:

```
INITIALIZER:
    def print_numbers():
      for i in range(10):
        print i,
      print
    [DUMP("initialized")]
```

```
FINALIZER:
    [DUMP("nothing to do in finalizer")]

STATECHART:
    A [DS]

TRANSITION:
    S: A
    N: A
    E: e
    O: print_numbers()
```

Here is the simulation result:

```
initialized
['A'] > e
0 1 2 3 4 5 6 7 8 9
['A'] > exit
nothing to do in finalizer
```

In the initializer shown above, a function `print_numbers` is defined to print out numbers from 0 to 9. It is a Python function that can be invoked in the output of the transition. The designer of a model may also use the initializer to define more functions, or to import Python modules to be used throughout the model description, or to initializer the hardware devices that the model controls.

The finalizer may be used to release allocated system resources. In this case, it simply prints out a message.

As is introduced for the first time, the transition in this example is a `self-loop`, which has the same definition of source state and new state. Note that in some cases, self-loops change the current state of the model, because (if the source/new state has substates) they causes the model to go to the default substate(s) of that state. A self-loop to the deep history of the new state never changes the model state.

2.5.7 An example of timed transition

Timed transitions are automatically triggered after a certain state is entered. The time between the execution of the enter actions of that state and the triggering of the transition is the *interval* of a timed transition. In a textual description, a timed transition is a transition with a `T` property but no `E` property as in normal transitions. (It is illegal to specify both `T` and `E` for the same transition.)

There are two kinds of timed transitions:

1. *Repeated timed transitions* are repeatedly scheduled if the source state `S` is equal to the new state `N`;
2. *Once timed transitions* are scheduled only once if `S` is equal to `N`.

Repeated timed transitions are the default type of timed transitions. This semantics conforms to David Harel's original statecharts. Once timed transitions must be explicitly specified by the designer by giving the `[OTT]` property to their intervals. (The `[RTT]` property for repeated timed transitions is negligible.)

The following is an example (`timed1.des`) of the two kinds of timed transitions:

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

```
TRANSITION:
    S: A
    N: B
    T: 1 [OTT]
    O: [DUMP("start printing")]

TRANSITION:
    S: B
    N: B
    T: 1  # by default, it is [RTT]
    O: [DUMP("tick")]
```

The simulation result is below:

```
['A'] > start printing
tick
tick
tick
tick
tick
tick
tick
tick
tick
exit
```

When started, this model is in state `A`, and the first timed transition (with `[OTT]` property) is scheduled after 1 second. When the transition is triggered, it turns the model to state `B`, and another timed transition is scheduler every 1 second to print out a "tick" message. Note that the current state (`['B']`) of the model is not printed for timed transitions, which are a kind of internal transitions. As a result, the user can only see the message keep appearing in the console, until the `exit` command is entered, when the simulation stops.

A timed transition may have a guard that specifies the condition that must be `true` to trigger the transition. The guard is evaluated after the scheduled time instead of at the time when the transition is scheduled. For example, if a transition is scheduled 1 second later, its guard is evaluated 1 second later. If the guard is `false`, the transition is not triggered. If the transition is a repeated timed transition, it is re-scheduled and its guard will be evaluated 1 second later again.

2.5.8 An example of importation

Importation is specified in model descriptions in two steps:

1. The designer declares all the components to be imported under the `IMPORTATION` descriptor. There may be multiple `IMPORTATION` descriptors in a single model description. Each component is located by its file name (usually ending with `.des`). A GUID (Globally Unique ID) is assigned to each component, which is used to referred to the component in other parts of the model description.

2. In the specification of the state hierarchy, the GUIDs of those imported components can be used as properties for states to make them importation states. The states with such properties must be leaf states. For example, if property `[b]` is given to state `A` and `b` is the GUID of a component, `A` becomes an importation state, where the component will be dynamically loaded in it during a simulation.

The following is a model (import1_a.des) that imports a component into its two leaf states (so that the two states have exactly the same internal structure):

```
IMPORTATION:
    b = import1_b.des

STATECHART:
    A [DS] [b]
    B [b]

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

The following is the component (import1_b.des) to be imported by the above model:

```
STATECHART:
    COMP_A [DS]
    COMP_B [FS]

TRANSITION:
    S: COMP_A
    N: COMP_B
    E: f
```

The submodel, import1_b.des is a model in its own right. It is possible to simulate it in SVM with command:

```
svm -t import1_b.des
```

When it is imported into import1_a.des, it is regarded as a component. All its states and transitions will be copied to the importation states at run-time. To simulate import1_a.des, run the following command instead:

```
svm -t import1_a.des
```

The simulation result is as following:

```
WARNING: Imported model has final states. They are automatically converted to ordinary states.
['A.COMP_A'] > f
['A.COMP_B'] > e
WARNING: Imported model has final states. They are automatically converted to ordinary states.
['B.COMP_A'] > f
['B.COMP_B'] > exit
```

Two warning messages are produced by SVM in this simulation, but the user can safely ignore them. When the simulation is started, the model is in its default state A. Because A has a property [b] and b is the GUID of component import1_b.des, SVM looks for import1_.des and import it into A to find the concrete current state of the model. When SVM detects a final state is defined in the imported component, it produces a warning telling the user that the final state is converted into an ordinary state. This is necessary because a component should not have the ability to terminate the simulation (for modularity).

When the user sends the f event, which is handled by the component, the current state of the model changes from A.COMP_A to A.COMP_B. After that, event e causes the model to leave state A and enter state B. (The component imported in state A is not deleted and can be reused later.) Because B is also an importation state, SVM dynamically loads the required component (import1_b,des) in it, and the second warning is produced. Event f received after this changes the model from state B.COMP_A to B.COMP_B.

2.5.9 An example of recursive importation

Since there is no essential difference between a model and a component, a model may import itself as a component directly or indirectly. This creates an infinite state hierarchy that cannot be flattened into a state hierarchy without importation. The following is an example of direct recursive importation (import2.des):

```
IMPORTATION:
    self = import2.des

STATECHART:
    A [DS]
    B [self]

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

The simulation result is as following:

```
['A'] > e
['B.A'] > e
['B.B.A'] > e
['B.B.B.A'] > e
['B.B.B.B.A'] > e
['B.B.B.B.B.A'] > e
['B.B.B.B.B.B.A'] > exit
```

In this example, the model imports itself into its importation state B by giving self (could be any other identifier) as the ID of its textual description and using this ID as a property of state B. As a result, when event e is received, import2.des itself is imported into state B and the current state becomes B.B.A. At this time, event e is again acceptable and is handled by the component that is just imported. This importation continues infinitely.

As an importance notice, it is the designer's responsibility to ensure there is no dead-lock in the model. If we slightly modify the above example, dead-lock is created: give the [self] property to state A or remove the [DS] of state A but add it to state B. In both cases, the default state of the model imports itself, and the default state of the just imported component again requires an importation. This operation never stops and the simulation hangs until all the memory is exhausted. SVM would never be able to provide the current state string to the user, because its length grows to infinity.

If model A imports model B and model B imports model A, an indirect recursive importation is created, which is not demonstrated here.

2.5.10 An example of macro redefinition

Macro redefinition for components in a model is accomplished simply by specifying the macros (both left-hand side and right-hand side) behind the name of the importation state in the state hierarchy. The names

of the macros should be the same as those used in the component. The following is an example of macro redefinition (macro2_a.des), which imports a component and redefines the macros in it:

```
MACRO:
    MYEVENT = e

IMPORTATION:
    b = macro2_b.des

STATECHART:
    A [DS] [b] [MYEVENT=[MYEVENT]]
```

The component (macro2_b.des) to be imported into the above model is defined below:

```
MACRO:
    MYEVENT =

STATECHART:
    COMP_A [DS]
    COMP_B

TRANSITION:
    S: COMP_A
    N: COMP_B
    E: [MYEVENT]
```

Simulation result:

```
['A.COMP_A'] > e
['A.COMP_B'] > exit
```

In this example, MYEVENT is a macro defined both in macro2_a.des and macro2_b.des. When macro2_b.des is imported, macro2_a.des redefines its MYEVENT macro to be the value of its own MYEVENT macro. (Note that in the property string [MYEVENT=[MYEVENT]], [MYEVENT] is replaced with e at the time when macro2_a.des is loaded, so it becomes [MYEVENT=e]. As a result, the MYEVENT macro in macro2_b.des is redefined to e.)

Another type of macro redefinition is done by the user on the command-line to invoke SVM. For example, the user may execute the following command to override the MYEVENT macro in macro2_a.des:

```
svm -t macro2_a.des "MYEVENT=f"
```

This command redefines the MYEVENT in macro2_a.des to be f. Since macro2_a.des redefines the MYEVENT macro in macro2_b.des to be its own MYEVENT macro, the accepted event in macro2_b.des now becomes f.

Macro redefinition is the only means for a model to modify the behavior of its components.

2.5.11 An example of ITF and OTF transition priorities

The following is a model (prio1_b.des) that tests the ITF (Inner Transition First) and OTF (Outer Transition First) transition priorities:

```
STATECHART:
```

```
    A [DS] [ITF]
      A1 [DS] [OTF]
        A11 [DS] [RTO]

TRANSITION:
    S: A
    N: A
    E: e
    O: [DUMP("transition from A")]

TRANSITION:
    S: A.A1
    N: A.A1
    E: e
    O: [DUMP("transition from A.A1")]

TRANSITION:
    S: A.A1.A11
    N: A.A1.A11
    E: e
    O: [DUMP("transition from A.A1.A11")]
```

The following is the simulation result of `prio1_b.des`:

```
['A.A1.A11'] > e
transition from A.A1
['A.A1.A11'] > exit
```

In this model, a conflict on event `e` is deliberately created. All the three transitions are able to handle event `e` when it is received: one transition from `A`, one from `A.A1` and the other from `A.A1.A11`. State `A` is set to be ITF, so the transition from `A.A1` and the transition from `A.A1.A11` have higher total priority than the transition from `A`. State `A.A1` is set to be OTF, so the transition from `A.A1` has higher total priority than the transition from `A.A1.A11`. (The OTF property of `A.A1` does not affect state `A`.) At last, state `A.A1.A11` is set to be RTO. This means transitions within its scope has a reverse ordering than its parent state, so the transition ordering of this state is equivalent to ITF. However, there is only one transition within its scope, so this property does not affect the result.

As a result, the second transition has the highest total priority. It is triggered when event `e` is received. Its output prints out the state where it is from: `A.A1`.

If a top-level state (like `A` in the above example) has not explicitly specified its transition priority, it is set to `OTF` by default. This behavior can be changed by reverting the global flag `InnerTransitionFirst` under the `OPTIONS` descriptor. More about global options are discussed later.

Because of this global option always presents, the behavior of a component is protected whether it is imported into an importation state with ITF property or an importation state with OTF property. The following model (`prio1_a.des`) imports `prio1_b.des`:

```
IMPORTATION:
    b = prio1_b.des

STATECHART:
    A [DS] [OTF] [b]
```

```
    B [ITF]
      B1 [RTO] [DS]
        B11 [DS] [b]

TRANSITION:
    S: A
    N: B
    E: e
    O: [DUMP("A-B")]

TRANSITION:
    S: B
    N: A
    E: e
    O: [DUMP("B-A")]
```

The simulation result is as following:

```
['A.A.A1.A11'] > e
A-B
['B.B1.B2.A.A1.A11'] > e
transition from A.A1
['B.B1.B2.A.A1.A11'] > exit
```

In this example, two more transitions participate in the conflict of event `e`. The `A` state of `prio1_a.des` is set to be OTF, so the first transition in the main model has higher priority than the `b` component in its state `A`. This results in the first transition (`A-B`) in the main model being triggered when event `e` is received for the first time. In state `B`, the transition has lower total priority than the transitions in the component imported into `B.B1.B11`. As a result, when event `e` is received for the second time, the transition in the component is triggered (a self-loop on state `B.B1.B2.A.A1.A11`).

2.5.12 An example of priority numbers

Priority numbers of transitions are arbitrary integer numbers used to solve conflicts that cannot be solved with the ITF and OTF mechanism. An example (`prio2.des`) is shown below:

```
STATECHART:
    A [DS]
      A1 [DS] [CS]
      A2 [DS] [CS]

TRANSITION:
    S: A.A1
    N: A.A1
    E: e
    O: [DUMP("transition from A.A1")]

TRANSITION:
    S: A.A2
    N: A.A2
    E: e
    O: [DUMP("transition from A.A2")]
```

```
TRANSITION: [0]
    S: A
    N: A
    E: f
    O: [DUMP("transition 1 from A")]

TRANSITION: [-1]
    S: A
    N: A
    E: f
    O: [DUMP("transition 2 from A")]
```

The simulation result is as following:

```
['A.A1', 'A.A2'] > e
transition from A.A1
transition from A.A2
['A.A1', 'A.A2'] > f
transition 2 from A
['A.A1', 'A.A2'] > exit
```

When event `e` is received in this model, two transitions are able to handle it. However, there is no conflict, because the two transitions are from two different orthogonal components `A.A1` and `A.A2`. Because of this, both of the transitions are triggered, and two messages are printed to the console. (The sequence of the triggering of those transitions is implementation-dependent.) When event `f` is received, two transitions are able to handle it, but they are from the same state. A conflict occurs at this time, and only one of those transitions is triggered. The ITF and OTF scheme can not be applied to this situation, because there is no superstate-substate relation between their source state. In this case, the transition with a smaller priority number has higher total priority. As a result, the last transition is triggered.

By default, each transition has a priority number of 0. If the conflict still cannot be solved with priority numbers, the final decision is implementation dependent.

2.5.13 An example of event parameters

The `EVENT` macro is predefined and is used to output an event in a model. This macro accepts one to three parameters. The first parameter is a string that specifies the event name. The second parameter is usually a Python list containing arbitrary parameters that are sent with the event. (By default, it is an empty list.) The transition that handles the event can thus retrieve the parameter list with macro `PARAMS`. Individual parameter is accessible with `[PARAMS][i]`, where `i` is the index of the parameter in the list. The third parameter of the `EVENT` macro may be a lock to be released when the event has been handled (discussed later).

The following example (`param1.des`) demonstrates the use of the two macros:

```
STATECHART:
    A [DS]
    B

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

```
    O: [EVENT("f", ["message", "Hello World"])]

TRANSITION:
    S: A
    N: B
    E: f
    O: [DUMP([PARAMS][0]+": "+[PARAMS][1])]
```

The simulation result is as following:

```
['A'] > e
message: Hello World
['B'] > exit
```

In this example, the user sends event `e` to the model. The event triggers a self-loop on `A`. In the output of the self-loop, event `f` is output with two parameters: Python strings `"message"` and `"Hello World"`. This event is then handled by the other transition defined in the model. In the output of that transition, `[PARAMS][0]` is equal to `"message"`, and `[PARAMS][1]` is equal to `"Hello World"`.

2.5.14 An example of model-specific interface

`INTERACTOR` is a special descriptor, which starts a piece of Python code to be executed in a thread separated from the thread of the SVM simulator. (If there are multiple `INTERACTOR` descriptors, the code under them is literally concatenated.

The code under the `INTERACTOR` descriptor is started after the initializer of the model is executed, but before the simulation of the model actually starts. This code should first initialize the user interface and then call the `start` method of the `eventhandler` object. This starts the simulation, and the model begins to accept events and produce outputs. The interactor does not exit after this, but it usually enters a loop. The code in this loop repeatedly accepts events from the user via the interface, and feeds those events to the model by calling the `EVENT` predefined macro or the `event` method of the `eventhandler` object.

The following example (`inter1.des`) demonstrates a textual model-specific interface:

```
STATECHART:
    A [DS]

INTERACTOR:
    import thread
    import string

    lock=thread.allocate_lock()

    # make sure the model is started before the events are sent to it
    lock.acquire()
    eventhandler.start(lock)
    lock.acquire()
    lock.release()

    # repeatedly handle events
    cmd=""
    while cmd!="quit":
      sys.__stdout__.write("CMD >  ")
```

```
      cmd=string.strip(sys.__stdin__.readline())
      if cmd!="quit":

        # split the cmd into [event, param] tuple
        [event, param]=string.split(cmd, ",")

        lock.acquire()
        [EVENT(string.strip(event), [eval(string.strip(param))], lock)]
        lock.acquire()
        lock.release()

    # shutdown the simulator
    eventhandler.shutdown()

TRANSITION:
    S: A
    N: A
    E: eval
    O: [DUMP("result: "+str([PARAMS][0]))]
```

The simulation result is as following:

```
CMD >  eval, 1+1
result: 2
CMD >  eval, "message"+": %s"%"Hello World"
result: message: Hello World
CMD >  eval, 2*3.1415927*0.5
result: 3.1415927
CMD >  quit
```

Here are the two points to be emphasized:

- A lock is used to synchronize the interactor thread with the simulator thread. When the interactor calls the `start` method of the `eventhandler` (an internal object that represents the simulator with the model loaded), the start event (a special event) is put on the top of the event queue (a global queue to record all the events that have not been handled yet). The method returns immediately without waiting for the actual starting of the simulation.

  To avoid the error caused by sending events to the model before it is fully initialized and the simulation is started correctly, the interactor must wait until the start event is handled. To ensure this, it calls the `start` method of an already acquired lock. The simulator releases the lock when it is started, but before that, the next acquisition of the same lock is blocked. This blocking causes the interactor thread to wait until the simulator is started.

- The `EVENT` macro is used in the interactor to send events to the model. The names of those events are the first part of the commands entered by the user (separated from the second part with a comma). There is exactly one parameter for each event (the second part in the command). Python function `eval` is used in the interactor to constructor the parameter object from the string entered by the user. The lock is also used to guarantee that the last event is handled before the user enters the next command.

3

CODE SYNTHESIS WITH SCC

SCC is a code synthesis tool distributed in the same package as SVM. It reuses the parser of SVM to interpret DCharts model descriptions, and then generate code from the internal data structures in different target languages. Since version 0.3, SCC provides exactly the same support for DCharts as SVM does.

SCC need not be installed separately. The `scc` (or `scc.bat` for Windows) script invokes it on the command-line. If the Windows binary package is used, the user should look for `scc.exe` in the SVM directory.

3.1 SCC Command-line Parameters

When SCC is invoked without any parameter, it prints out the following message about the usage:

```
================================================
|  Python Implementation of Statechart Compiler  |
|                   Version 0.3                  |
|       Presented by Thomas Feng, Nov. 2003      |
================================================

Usage: scc [options...] <.des file> [parameters...]
       options:
           -l <lang>: generate code in language <lang>
                      supported languages: java (default), cpp, csharp, python
           -i <file>: include a file (to the head)
           -I <file>: include a file (to the tail)
           -q:        quiet
       language-specific options:
         cpp:
           --head:    generate head file
           --ext:     include extensions for actions (require Python dynamic library)
         python:
           --ext:     include extensions for actions
       parameters:
           "name=value"
```

The following command is the simplest use, which compiles `model.des` into Java source `model.java`:

```
scc model.des
```

The -i and -I parameters are similar to their counterparts on the SVM command-line. -i is used to insert a text file to the head of the model description, while -I is used to append a text file to the end of it.

It is also possible to redefine the macros in the model description, as can be done in SVM. The redefined macros are statically compiled into the generated code. (Note that it is impossible to dynamically redefine macros in the generated code. If a model redefines macros in its components, those redefinitions are ignored by SCC.) The model redefinitions are written after the name of the model description. (This is similar to SVM.)

The -l parameter is followed by the name of the target language. Currently Java, C++, C# and Python are supported as target languages. For example, to compile model.des into model.cpp, execute the following command:

```
scc -lcpp model.des
```

or

```
scc -l cpp model.des
```

By default, SCC does not generate any action code or guard in the model. They are simply ignored. This is because, enabling the action code and guards for target languages other than Python requires extra Python library. To minimize system requirement and still guarantee unique target-language-independent behavior of the generated code, SCC ignores all the action code and guards even if Python is the target language. However, the user may specify the --ext command-line parameter to enable the action code and guards for *some* target languages. Uniqueness of behavior between different target languages is not guaranteed in this case. (See the examples discussed later.)

After code is generated, SCC prints out information about the time spent in the code synthesis, the command to compile the source code in the target language (if the target language is not interpreted), and the command the execute the model after the generated code is compiled. The user may disable this information with the -q parameter.

3.2 HelloWorld Example

The same HelloWorld example as discussed in the SVM chapter is used here. It is compiled into source code in different target languages. The readers may compare the result of their executions with the simulation result shown previously.

```
DESCRIPTION: # description of the model
    Hello World Example

STATECHART:  # definition of the state hierarchy
    A [DS]
    B [FS]

TRANSITION:  # one single transition
    S: A
    N: B
    E: e
    O: [DUMP("Hello World!")]
```

3.2.1 Python code synthesis

To synthesize Python code from the model description, the user may execute the following command:

```
scc -lpython HelloWorld.des
```

`HelloWorld.py` will be generated. The following information is printed on the console:

```
-----
Time spent on compilation:         0.010 (sec)
File(s) generated:                 HelloWorld.py
-----
Command to compile the source:     (N/A)
Command to run the compiled code:  python HelloWorld.py
-----
```

This output tells the user that:

1. SCC spent 0.010 second in generating the target code;
2. the only file produced is `HelloWorld.py`;
3. because target language Python is an interpreted language, it is not necessary to compile `HelloWorld.py` before execution;
4. to execute the generated code, execute command:

   ```
   python HelloWorld.py
   ```

Now execute `HelloWorld.py` and enter some events in the console:

```
    Hello World Example
['A'] > e
['B'] > exit
```

When the execution is started, the model description is printed, and a prompt for input appears. The current state of the model is also displayed in the format of Python list. If event `e` is entered, the model correctly changes to state `B`. Command `exit` ends the execution and stops the program.

One may notice that the “Hello World!” message was not printed as it was printed in the simulation in SVM. This is because the output of the transition is ignored by SCC, which defaults not to generate any action code for the target code. To synthesize code with support for action code, the user should execute the following command instead:

```
scc -lpython --ext HelloWorld.des
```

Again, similar information about the code synthesis is printed on the console:

```
-----
Time spent on compilation:         0.015 (sec)
File(s) generated:                 HelloWorld.py
-----
Command to compile the source:     (N/A)
Command to run the compiled code:  python HelloWorld.py
-----
```

Now execute the target code again. The result should look like:

```
    Hello World Example
['A'] > e
Hello World!
['B'] > exit
```

This execution result is exactly the same as simulating the model in SVM:

```
svm -t HelloWorld.des
```

3.2.2 C++ code synthesis

The same model can also be compiled to equivalent C++ code with the following command:

```
scc -lcpp HelloWorld.des
```

The following information is displayed in the console:

```
-----
Time spent on compilation:         0.045 (sec)
File(s) generated:                 HelloWorld.cpp
-----
Command to compile the source:     g++ -o HelloWorld HelloWorld.cpp
Command to run the compiled code:  ./HelloWorld
-----
```

File `HelloWorld.cpp` is generated, which can be compiled with G++ (the GNU C++ compiler) with the following command:

```
g++ -o HelloWorld HelloWorld.cpp
```

In Windows, compiling C++ source code with G++ requires Cygwin.

The `HelloWorld` executable can be directly executed on the command-line. The result is as following:

```
    Hello World Example
['A'] > e
['B'] > exit
```

The action code is ignored in this executable.

If the user wants to get a C++ head file, which can be included in other C++ sources, he/she may use the `--head` command-line parameter. In that case, two files are generated: `HelloWorld.h` and `HelloWorld.cpp`. The G++ command to compile this source is not changed.

The `--ext` parameter can be used to instruct SCC to include support for the action code and guards. However, the Python library is required to compile the generated C++ code, and if the model loads user-defined Python libraries with `import` statements, those libraries must be in the `PYTHONPATH` environment variable. For example, to synthesize code for the `HelloWorld` example with action language support, execute the following command:

```
scc -lcpp --ext HelloWorld.des
```

Again, `HelloWorld.cpp` is generated. The command to compile this C++ source is different and much more complex:

```
g++ -I/usr/include/python2.2 -L/usr/lib/python2.2/config -Xlinker -export-dynamic \
HelloWorld.cpp -lpython2.2 -lm -ldl -lpthread -lutil -o HelloWorld
```

(The user may concatenate the two lines with the ending "\" mark on the first line removed.)

SCC automatically detects the libraries and head files of the Python version that is being used. In the above example, the Python head files are located at `/usr/include/python2.2` and the Python libraries are at `/usr/lib/python2.2/config`. Other command-line parameters for G++ depend on the parameters with which Python itself was compiled. This command-line differs in different systems. If the user does not want to compile the source with the Python that is being used (the Python in which SCC is executed), he/she must modify the paths to the head files and libraries according to another version of Python.

When compiled, `HelloWorld` is the executable, which yields the following result:

```
    Hello World Example
['A'] > e
Hello World!
['B'] > exit
```

This execution result is exactly the same as the execution result of `HelloWorld.py` and the simulation result of `HelloWorld.des` in SVM.

3.2.3 Java code synthesis

SCC is also able to synthesize Java code from DCharts models. The Java code requires JDK (Java Development Kit) to compile to Java byte-code. The most recent version of JDK is downloadable from:

`http://java.sun.com/`

(JDK 1.4 or above is recommended.)

The following command synthesizes Java code for `HelloWorld.des`:

```
scc -ljava HelloWorld.des
```

The following message is printed out to the console:

```
-----
Time spent on compilation:         0.007 (sec)
File(s) generated:                 HelloWorld.java
-----
Command to compile the source:     javac HelloWorld.java
Command to run the compiled code:  java HelloWorld
-----
```

`HelloWorld.java` is the only Java file for the model. It contains the definition of several classes. When it is compiled with `javac`, the following class files are produced in the current directory:

```
EventList.class
HelloWorld.class
Hierarchy.class
History.class
State.class
StateMachine.class
StringList.class
```

Among those class files, HelloWorld.class provides the entrance of the program. To execute it, use the following command:

```
java HelloWorld
```

(Note: the postfix of the main class file is removed.)
The result of the execution is as following:

```
    Hello World Example
['A'] > e
['B'] > exit
```

The --ext parameter is currently not supported for target code in the Java language. Later versions of SCC will add support for action code and guards. However, to provide this support, an interface between the Java code and the C library provided by Python is required, which makes the target code platform-dependent.

3.2.4 C# code synthesis

To synthesize C# code for the HelloWorld example, execute the following command:

```
scc -lcsharp HelloWorld.des
```

HelloWorld.cs is generated and the following message is printed out to the console:

```
-----
Time spent on compilation:         0.007 (sec)
File(s) generated:                 HelloWorld.cs
-----
Command to compile the source:     mcs -main:HelloWorld HelloWorld.cs (with Mono)
                                   csc /main:HelloWorld HelloWorld.cs (with .Net Framework SDK)
Command to run the compiled code:  mono HelloWorld.exe (with Mono)
                                   HelloWorld.exe (with .Net Framework)
-----
```

This tells the user how to compile the source and execute the program in Linux and Windows.

- Linux.

  Mono (a C# compiler and run-time environment) must be installed. It is downloadable from:

  http://www.go-mono.com/

  To compile the C# source generated by SCC, execute the following command:

  ```
  mcs -main:HelloWorld HelloWorld.cs
  ```

  Here, the -main parameter is specified on the command-line, which tells Mono the entrance of the program is in class HelloWorld. (The entrance is the Main method of the class.) This is necessary because, if the model imports other components, C# classes are also generated for those components by SCC in the same source file. In that case, there will be multiple Main methods, and the user must explicitly specify only one class on the Mono command-line with the -main parameter.

  After compilation, HelloWorld.exe is produced. It is a valid Windows executable, but in Linux, it must be executed with the support of Mono:

```
mono HelloWorld.exe
```

(Note: this executable may be executed in Windows with Microsoft .Net Framework installed.)

- Windows.

  Microsoft .Net SDK (Software Development Kit) is required to compile `HelloWorld.cs` in Windows. The SDK is downloadable from:

  `http://download.microsoft.com/`

  To compile the C# source, execute the following command:

  ```
  csc /main:HelloWorld HelloWorld.cs
  ```

  Similar to Mono, .Net SDK generates `HelloWorld.exe`. The executable can be directly executed in Windows, but it requires .Net Framework, which is included in the SDK. The end users who only want to run the executable model may download .Net Framework only from the download website of Microsoft.

The execution result of the C# program is as following:

```
    Hello World Example
['A'] > e
['B'] > exit
```

Currently the `--ext` parameter is not supported for the C# target language. All the action code and guards in the model are ignored.

3.3 An example of model-specific interface

Interactor, like other parts of a model that contain arbitrary Python code (e.g., initializer and finalizer), is supported by SCC. The interactor of a model is executed in a separate thread, and the exiting of that thread denotes the end of the execution. If Python is the target language, the thread is created with the methods provided by the `thread` native Python module. (This requires the Python environment to be compiled with thread support.) If C++ is the target language, the thread is created with the functions in the pthread library. This library is available in both Linux and Windows.

Here is the model (`inter1.des`) copied from section 2.5.14:

```
STATECHART:
    A [DS]

INTERACTOR:
    import thread
    import string

    lock=thread.allocate_lock()

    # make sure the model is started before the events are sent to it
    lock.acquire()
    eventhandler.start(lock)
    lock.acquire()
```

```
    lock.release()

    # repeatedly handle events
    cmd=""
    while cmd!="quit":
      sys.__stdout__.write("CMD >  ")
      cmd=string.strip(sys.__stdin__.readline())
      if cmd!="quit":

        # split the cmd into [event, param] tuple
        [event, param]=string.split(cmd, ",")

        lock.acquire()
        [EVENT(string.strip(event), [eval(string.strip(param))], lock)]
        lock.acquire()
        lock.release()

    # shutdown the simulator
    eventhandler.shutdown()

TRANSITION:
    S: A
    N: A
    E: eval
    O: [DUMP("result: "+str([PARAMS][0]))]
```

To compile this model into Python code, execute the following command:

```
scc -lpython --ext inter1.des
```

The generated Python source (`inter1.py`) can be executed with command:

```
python inter1.py
```

The execution result is as following, which is the same as the result shown in section 2.5.14:

```
CMD >  eval, 1+1
result: 2
CMD >  eval, "message"+": %s"%"Hello World"
result: message: Hello World
CMD >  eval, 2*3.1415927*0.5
result: 3.1415927
CMD >  quit
```

Similarly, the following is the command to synthesize C++ code from the same model:

```
scc -lcpp --ext inter1.des
```

And the following is the command to compile the generated C++ source code (`inter1.cpp`):

```
g++ -I/usr/include/python2.2 -L/usr/lib/python2.2/config -Xlinker -export-dynamic \
inter1.cpp -lpython2.2 -lm -ldl -lpthread -lutil -o inter1
```

The execution result of the binary is exactly the same as above.

3.4 Reusing the Synthesized Code

The code generated by SCC from a model description can be reused by other applications. In that case, the code becomes a part of the application, and the end user may not know that this part is generated from a DCharts model. An example of use is that designer models the control logic of a system in DCharts. The model can be validated with tools. The designer then synthesizes target code from the model. This code is thus incorporated into the whole system. It provides robust support for the control logic.

The following is a very simple model (`abc.des`) that specifies (sort of) control logic:

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

INITIALIZER:
    Accept = 0

FINALIZER:
    Accept = 1

TRANSITION:
    S: A
    N: B
    E: a

TRANSITION:
    S: B
    N: C
    E: b

TRANSITION:
    S: C
    N: D
    E: c
```

This model checks whether events `a`, `b` and `c` appear in the same order in all the events that it receives. If the check is successful, the flag `Accepted` is set to 1; otherwise, it is set to 0. The following event lists are checked successfully (each character is an event in the string):

```
abc
aebdac
eghabbabcyucvb
```

The following are invalid event lists that cause failure:

```
a
abbdadayule
cababauolp
```

This model can be executed in SVM:

```
svm -t abc.des
```

However, though the `Accept` flag is set in the model, it is not displayed to the user (or model tester).

Before being used in an application (written in native Python), the model must first be compiled into Python source (`abc.py`) with the following command:

```
scc -lpython --ext abc.des
```

This code is used in an application (`abc_file.py`) that tests a text file according to the same logic. The name of the text file is specified on the command-line:

```
import code
import sys
import thread

# Import the abc class from the abc model (abc.py)
from abc import abc

# Get the file name from the command-line
fname=sys.argv[1]

# Read in the while file
infile=open(fname, "r")
content=infile.read()
infile.close()

# Create an interpreter with the local dictionary
# (so that the Accept flag can be access in the current scope)
interpreter=code.InteractiveInterpreter(locals())

# Create a lock to synchronize the events
lock=thread.allocate_lock()

# Initialize a model with the interpreter
abc_model=abc(interpreter)
abc_model.initModel()

for c in content:
  # Treat every character as an event and handle it
  lock.acquire()
  abc_model.event(c, [], lock)
  lock.acquire()
  lock.release()

# Print out the result
if Accept:
  print "success"
else:
  print "failure"
```

This Python source is the application that imports the code generated by SCC, and uses the methods in it to check a text file. Here are several important notes:

- An interpreter is created and passed to the constructor of the `abc` class (synthesized from the `abc` model). All the action code and guards are executed or tested with this interpreter. If this parameter of the constructor is omitted, the model automatically creates an interpreter that has a scope different from those of all the existing Python interpreters.

  In this example, the interpreter is created with the local dictionary (returned by the Python function `locals`). As a result, the action code is executed as if it is directly written in the current context. The `Accept` flag can thus be accessed. (However, this may cause the name conflict between the variables in the model and the variables in the current scope of the native program.)

- The model must be initialized before it is able to handle events. This is done by calling its `initModel` method. (This method is different from the `start` method that is called from the interactor of the model.)

- When initialized, the application sends events to the model by invoking its `event` method. As discussed before, the first parameter is a string that specifies the event name, the second parameter is an arbitrary Python variable, which is usually a list of parameters for the event, and the third parameter is a lock that is released when the model finishes handling the event. The last 2 parameters can be omitted.

Now, edit a text file and type in some characters. Save it as `abc_test.txt`. For example:

```
dsa dewrevbd
dtgbtyrtvc
```

To check this file, execute the following command (make sure that `abc.py` is in the same directory):

```
python abc_file.py abc_test.txt
```

As expected, the result is “success”.

It is similar to reuse the code synthesized in other target languages. This is not discussed here.

4

SVM INTEGRATED WITH ATOM^3

SVM can be used as a plugin for AToM^3 (A Tool for Multi-formalism Meta-modeling, developed at MSDL, McGill University by Prof. Hans Vangheluwe). It simulates the DCharts models designed in AToM^3 on-the-fly, and generates textual model descriptions from the graphical models. It is also able to generate target code directly from the graphical models.

The functionality of the SVM simulator and code synthesizer can be added to an existing AToM^3 environment. If the user does not have AToM^3 installed yet, he/she must download it from the AToM^3 homepage

`http://atom3.cs.mcgill.ca/`

and install it manually (simply by extracting the package).

After AToM^3 is installed, the user should take the following steps to install the SVM plugin:

1. Download and install a recent version of SVM. Though the CVS version is not stable, it is recommended for AToM^3 users, because it has increased compatibility with AToM^3 and an improved code generator.
2. Add the SVM directory to the `PATH` environment variable and `PYTHONPATH` environment. This is to allow the user and the AToM^3 environment to invoke SVM directly from any other directory.
3. Copy the `plugins/AToM3/DCharts/` directory (and all the files in it) to the AToM^3 directory.
4. Start AToM^3 and select the “Options” item in the “File” menu. Modify the settings in the “Options” dialog according to Figure 4.1. (Set the “Dir. for Code Generation” to “DCharts”; Set the “Initial Meta-Model” is to “DCharts”; and click on the “new” button to insert “DCharts” to the “Path Directories”.)
5. Restart AToM^3 and the DCharts meta-model will be loaded automatically. It is now possible to design DCharts models in the AToM^3 environment.

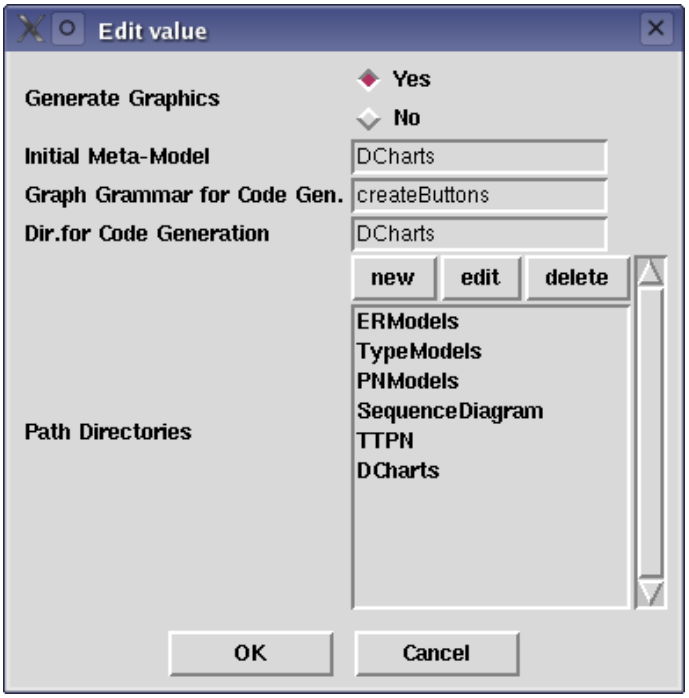

Figure 4.1: Settings in the AToM^3 “Options” dialog to enable the DCharts meta-model

5 DISTRIBUTED SIMULATION WITH SVM

SVM supports distributed simulation with PVM (Parallel Virtual Machine). 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 can run on the same machine. Multiple machines may be involved in the distributed simulation enabled by PVM, after they are added to the PVM daemon.

5.1 PVM Requirement

PVM must be installed for this distributed simulation. The source code of PVM is available from its homepage:

`http://www.csm.ornl.gov/pvm/pvm_home.html`

For Windows users, a binary installer is provided. For other operating systems, the user must download the source code and manually compile it according to the readme file in the package. Fortunately, PVM binary is included in most Linux distributions, such as RedHat Linux 9.0, SuSE Linux 9.0 and Mandrake Linux 9.2.

Some environment variables must be set before using PVM in SVM. `PVM_ROOT` points to the path where PVM is installed (for RedHat Linux 9.0, it is `/usr/share/pvm3`). `PVM_ARCH` should be set to the operating system and architecture of the computer. Its value should be the same as the name of the only subdirectory in `$PVM_ROOT/lib` (the `lib` subdirectory in the path where PVM is installed). Use the following command to set the `PVM_ARCH` variable in Linux systems (assuming the shell is bash):

```
export PVM_ARCH=`ls -F $PVM_ROOT/lib | gawk 'sub(/\//, "");'`
```

`PVM_RSH` points to the remote shell to be used by PVM. PVM executes this shell program when it attempts to connect to another machine. If `ssh` is in the `PATH` environment variable, set `PVM_RSH` to `ssh`; otherwise, set it to the full path of the `ssh` program, for example, `/usr/bin/ssh`. If the user wants to run the PVM console from any directory, the path `$PVM_ROOT/lib/$PVM_ARCH` should be appended to the `PATH` environment variable.

To start the PVM daemon, run the PVM console with the `pvm` command, and enter `quit` in the PVM prompt. The PVM console is closed but the PVM daemon is running in the background. Alternately, use the following Linux command:

```
echo quit | pvm
```

To stop the PVM daemon, run the PVM console and enter `halt`, or execute:

```
echo halt | pvm
```

To add another machine to the current PVM daemon, start the PVM daemon on the local machine. Make sure the following conditions are met:

- PVM is correctly installed on the other machine.
- The required environment variables are correctly set on the other machine in its startup script for the login user.
- PVM daemon has *not* been started on the other machine.
- The current user is able to ssh to the other machine without extra command-line parameters. If another user name is used on the other machine, specify it in file `/.ssh/config`. For example, the user wishes to connect to machine `pvm2.private.net` via ssh with user name `tfeng`, put the following lines in `/.ssh/config`:

  ```
  Host pvm2.private.net
       User tfeng
  ```

  Consult the ssh manual for more information about `/.ssh/config` and ssh protocol 1 and 2, or type the following command in Linux to get help:

  ```
  man ssh_config -S 5
  ```

The `add` command in the PVM console adds a machine to the current PVM daemon. It has a single parameter which is the host name of the machine to be added. For example, the following commands start the PVM daemon and add machine `pvm2.private.net` to it:

```
echo quit | pvm
echo add pvm2.private.net | pvm
```

The following command displays the current configuration of the PVM daemon:

```
echo conf | pvm
```

Sometimes when PVM applications crash, the PVM daemon cannot be stopped or restarted because of the trash left in the temporary directory. This can be fixed by removing all the PVM temporary files from the temporary directory. For Linux, the following command is useful:

```
rm -rf /tmp/pvm*
```

5.2 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 5.1 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 correspond 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 an SVM process:

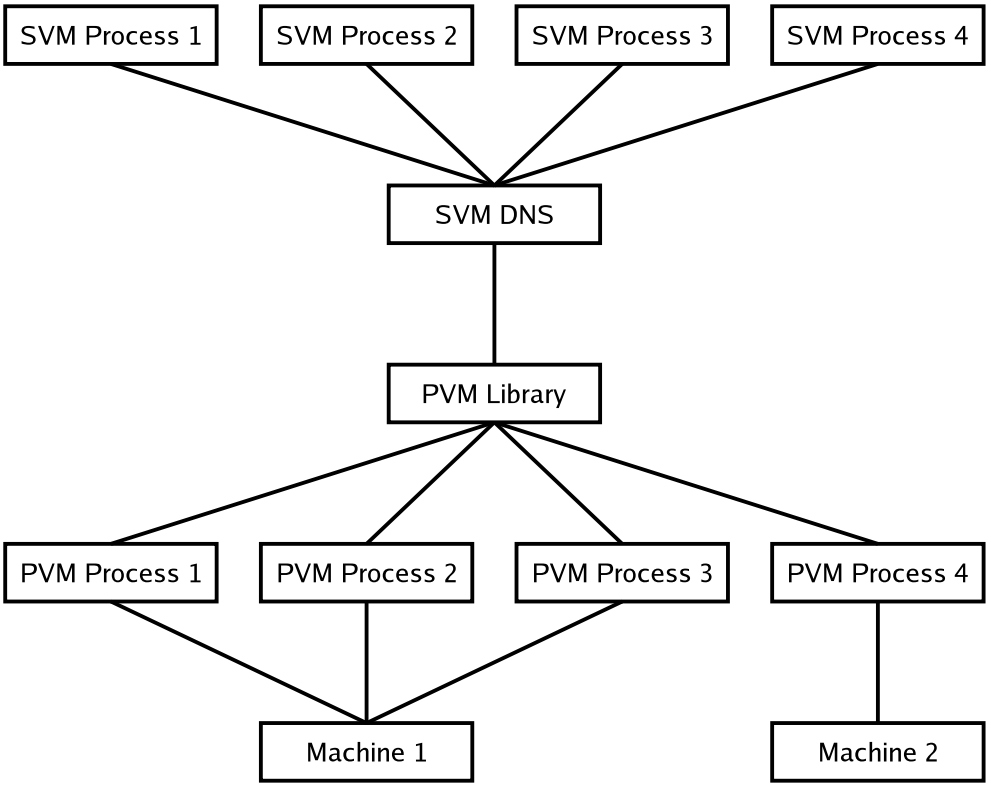

Figure 5.1: Multiple layers for distributed simulation in SVM

- Registration. Each SVM process that interacts with remote components must register itself to the 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.
- 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 is able to locate the registered components with those name patterns and types. It establishes connections between components.
- SVMDNS also maintains the connections between different components. The 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 inter-component communication.

To use SVMDNS, make sure the PVM requirement is satisfied, and compile the PYPVM (`http://pypvm.sourceforge.net/`) support with command "`make pypvm`" in the SVM directory. PYPVM enables access to the native PVM library from Python code.

The script `svmdns` in the SVM directory starts the SVMDNS console. The SVMDNS command prompt appears, and commands can be entered. The `help` command displays all the available commands and their concise descriptions (Table 5.1). Initially the SVMDNS daemon is not started. To start it, execute the "`start`" command. To close the SVMDNS console, execute "`exit`". This command does not stop the SVMDNS daemon, which keeps running in the background. Other commands are explained below:

```
SVM DNS User Interface
 |  SVM DNS is not running.
>>> help
 |  Statechart Virtual Machine Dynamic Naming System
 |  USAGE:
 |    svmdns <param>
 |  PARAMETERS:
 |    help          print this help message
 |    check         check if DNS has been started
 |    exec "cmd"    execute a python command on the DNS (quotes are removed)
 |    send "msg"    sends msg to the DNS (quotes are removed)
 |    start         starts the DNS
 |    stop          stops the DNS
>>>
```

Table 5.1: SVMDNS console

- The "`check`" command checks the current status of the daemon (whether it is started).
- The "`exec "cmd"`" command executes an arbitrary Python statement on the daemon. It can only be executed after the daemon is started with the "`start`" command. The "`exec`" command is useful for debugging. For example, "`exec "check_master_started()"`" checks the status of the daemon, and command "`exec "print SVMs"`" displays all the components registered in the daemon.
- The "`send "msg"`" command sends a message to the daemon. It can only be executed after the daemon is started with the "`start`" command. Messages received by the daemon, whether they are sent from SVM simulators or from the SVMDNS console, are displayed on the console where SVMDNS is started.
- The "`stop`" command stops the running daemon. It has no effect if the daemon is not running on the local machine.

The user may directly execute a command from the shell by running `svmdns` with the command as a command-line parameter. For example, to check the status, run "`svmdns check`" in the command-line; to execute an arbitrary Python statement, run "`svmdns exec "cmd"`".

File `DNS` in the SVM directory records the PVM ID of the SVMDNS daemon. Once the SVMDNS daemon is started, it modifies this file with its PVM ID (the return value of C function `pvm_mytid()`). SVM simulators use this file to locate the SVMDNS daemon. When the daemon is on another machine, the user must ensure that the content of the `DNS` file in its SVM directory is the same as the one on the machine where the SVM daemon is running. For a distributed simulation, there should be exactly one SVMDNS daemon running in the background. Each SVM simulator locates the same daemon with the `DNS` file on their own machines.

5.3 Distributed Simulation on Top of SVMDNS and PVM

SVM supports distributed simulation on top of SVMDNS and PVM. The user should start PVM first and then SVMDNS. When the two daemons are running in the background, run SVM with a model that contains at least one port. If the simulator successfully registers itself to the SVMDNS daemon, the message "`init dns success (PID=xxxxxx)`" is printed to the console, where "`xxxxxx`" is a 6-digit integer PVM ID of the simulator. After the simulator is initialized, the DCharts component running on it may communicate with other components via the ports defined in it.

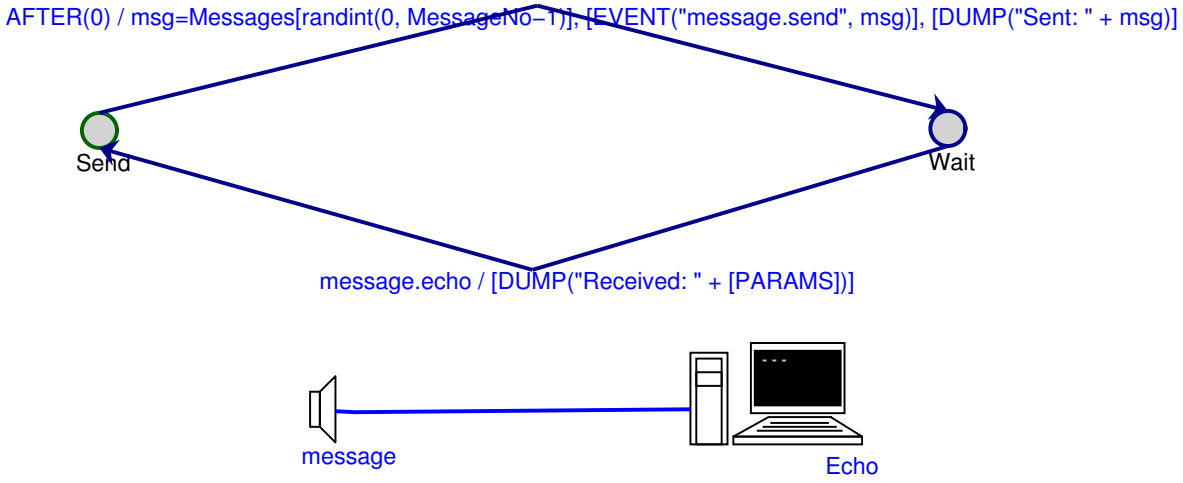

Figure 5.2: `Sender` of the `Echo` example

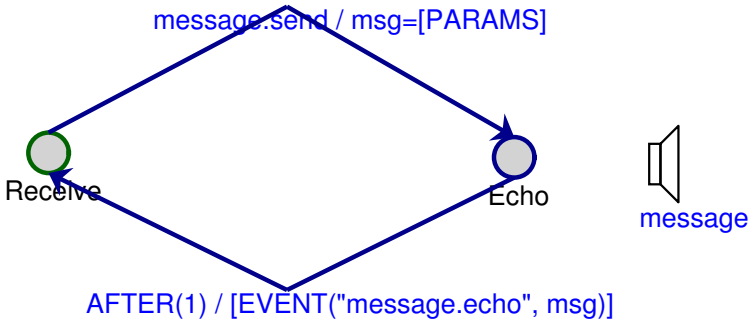

Figure 5.3: `Echo` of the `Echo` example

5.4 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.

The design of the components in AToM^3 is shown in Figure 5.2 and Figure 5.3. In Figure 5.3, an input/output port named `message` is defined. The `Sender` component in Figure 5.2 also defines a port called `message`. The port of the `Sender` is connected to the port of the `Echo`. The name pattern of the server with ID `Echo` (a name chosen by the designer) is `Echo` (Figure 5.4). This matches the `Echo` component only. The link between the `Sender` port and the server has a property that specify the server port `message` (Figure 5.5). 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. The user is prompted for a file name. By default, it is the same name as the AToM^3 component with postfix changed to `.des`. 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
#   Time:   21:29:44
```

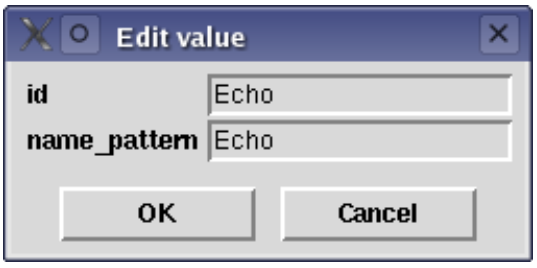

Figure 5.4: Name pattern of the `Echo` server

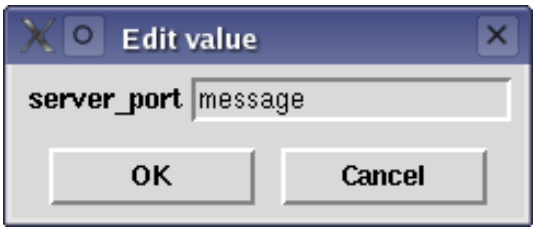

Figure 5.5: Port name of the `Echo` server

```
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 the `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)]
```

To simulate this example in AToM^3, take the following steps:

1. Install and start PVM (section 5.1).
2. Compile PYPVM and start SVMDNS (section 5.2).
3. Start an instance of AToM^3 and load `Echo.py`. Click on “simulate in SVM”.
4. Start an instance of AToM^3 and load `Sender.py`. Click on “simulate in SVM”.
5. The output box of the `Sender` component should display such messages as “sent: xxxxxx” and “receive: xxxxxx”, where “xxxxxx” is a message. The received message should be the same as the last message being sent.
6. To stop the simulation and close the SVM simulators, click on the “Exit” button.

To simulate this example in SVM, take the following steps:

1. Start an instance of AToM^3 and load `Echo.py`. Click on “to SVM Des.” to generate `Echo.des` and save it in a user-specified directory. Close AToM^3.

2. Start an instance of AToM^3 and load `Sender.py`. Click on “to SVM Des.” to generate `Sender.des`. Close AToM^3.
3. Install and start PVM (section 5.1).
4. Compile PYPVM and start SVMDNS (section 5.2).
5. Change to the directory where `Echo.des` is in and execute “`svm Echo.des`”.
6. Change to the directory where `Sender.des` is in and execute “`svm Echo.des`”.

6

USING SVM AND SCC IN CYGWIN

SVM (with SCC) can be easily installed in a Linux system (simply by unpacking the package and setting a few environment variables). However, installing and using it in Windows requires much more effort.

Cygwin (`http://www.cygwin.com/`) is a Unix-like environment for Windows. It eases the problems of running SVM in Windows and compiling the code synthesized by SCC. Though it is possible to run SVM and SCC without Cygwin, the support for such a native Windows environment is limited. The users are strongly recommended to use Cygwin.

6.1 Installing Cygwin

Cygwin is free and can be downloaded from its website. The setup program allows the users to choose from the available packages. SVM and SCC users must make sure the following packages and the packages that they depend on are chosen before performing the installation (however, it is possible to add more packages after the installation is finished):

1. Python. The Python interpreter should be chosen by default.
2. G++. It is the GNU C++ compiler. In order to compile the C++ source generated by SCC, the users should have this package installed.
3. GCC. It is the GNU C compiler. It is used to compile the PYPVM module to be used by SVM in distributed simulations. (However, the users need not compile PYPVM if they are not intended to run distributed simulations.)
4. Make. The GNU Make utility to automate the building of applications.
5. Bash. It is the shell to be used by the Make files.
6. SunRPC. It is the library required by PVM. Users who want to enable distributed simulations must install this package.
7. Patch. It is a small utility to apply patches to the PVM source (version 3.4.3) before it can be smoothly compiled and used by SVM in distributed simulations.
8. UPX. It is a tool to compress executable programs.

9. If the users want to keep their SVM and SCC source up-to-date with the CVS, they are highly recommended to install CVS and OpenSSH.
10. XFree86 and XTerm. If the users want to interact with the SVM graphical interface, they must install these packages.

When Cygwin is installed, it is usually in the `Cygwin` directory of the `C:` drive. The users can thus start Cygwin by double clicking on its icon on the desktop (if they choose to create an icon on the desktop at the end of the installation).

The users may run the setup program again to add more packages to the existing Cygwin installation.

6.2 Installing PVM3

This step is optional and necessary only for those users who are interested in SVM distributed simulations.

PVM (Parallel Virtual Machine) is the underlying library that supports communication among multiple processes and multiple machines. Its source can be downloaded from its website:

`http://www.csm.ornl.gov/pvm/pvm_home.html`

PVM is not originally designed for Cygwin, and its version 3.4.3 is used here. The users are highly recommended to download the source from the following location:

`http://scicomp.ewha.ac.kr/netlib/pvm3/pvm3.4.3.tgz`

Note that newer versions of PVM may be found at:

`http://scicomp.ewha.ac.kr/netlib/pvm3/`

However, the only version that has been tested and successfully compiled is 3.4.3, and the following patch is applicable only for that version:

`http://msdl.cs.mcgill.ca/people/tfeng/svmsccdoc/pvm3.4.3-Cygwin.patch`

The content of the patch is:

```
Only in pvm3_patched/console: CYGWIN
Only in pvm3_patched/examples: CYGWIN
Only in pvm3_patched/gexamples: CYGWIN
Only in pvm3_patched/lib: CYGWIN
Only in pvm3_patched/libfpvm: CYGWIN
Only in pvm3_patched/pvmgs: CYGWIN
Only in pvm3_patched/rm: CYGWIN
Only in pvm3_patched/src: CYGWIN
diff -uar pvm3/src/lpvm.c pvm3_patched/src/lpvm.c
--- pvm3/src/lpvm.c 2000-02-17 18:12:10.000000000 -0500
+++ pvm3_patched/src/lpvm.c 2004-03-21 18:26:33.069064000 -0500
@@ -619,7 +619,7 @@

 #ifndef HASERRORVARS
 extern int errno; /* from libc */
-extern char *sys_errlist[];
```

```
+//extern char *sys_errlist[];
 extern int sys_nerr;
 #endif

diff -uar pvm3/src/lpvmgen.c pvm3_patched/src/lpvmgen.c
--- pvm3/src/lpvmgen.c 2000-02-17 18:12:12.000000000 -0500
+++ pvm3_patched/src/lpvmgen.c 2004-03-21 18:26:20.340761600 -0500
@@ -578,7 +578,7 @@

 #ifndef HASERRORVARS
 extern int errno; /* from libc */
-extern char *sys_errlist[];
+//extern char *sys_errlist[];
 extern int sys_nerr;
 #endif

diff -uar pvm3/src/pvmlog.c pvm3_patched/src/pvmlog.c
--- pvm3/src/pvmlog.c 2000-02-10 15:46:43.000000000 -0500
+++ pvm3_patched/src/pvmlog.c 2004-03-21 18:25:59.741140800 -0500
@@ -192,7 +192,7 @@
 extern int errno; /* from libc */
 #ifndef WIN32
 extern int sys_nerr;
-extern char *sys_errlist[];
+//extern char *sys_errlist[];
 #endif
 #endif

Only in pvm3_patched/tracer: CYGWIN
Only in pvm3_patched/xep: CYGWIN
```

After obtaining the source (`pvm3.4.3.tgz`) and the above patch (`pvm3.4.3-Cygwin.patch`), the users may use the following commands to extract the package and apply the patch (assuming that both files are in the current directory):

```
$ tar zxvf pvm3.4.3.tgz
$ patch -p0 < pvm3.4.3-Cygwin.patch
```

A directory named `pvm3` will be created, which contains the patched source. To build PVM, the users must first set the `PVM_ROOT` environment variable according to the PVM path. For example, the following command may be used (suppose the current shell is `bash`):

```
export PVM_ROOT=`pwd`/pvm3
```

Then the source can be made with the following command:

```
make -C pvm3
```

If all goes well, PVM will be built in the `pvm3` directory, and the executables can be found in `pvm3/lib/CYGWIN`. The users should set the following environment variables in their startup scripts (`/.bashrc` for the `bash` shell):

1. PVM_ROOT. As mentioned above, set it to the PVM path.
2. PVM_ARCH. For Cygwin, this variable should always be set to CYGWIN.
3. PATH. The users may add the path of the PVM executables to the PATH environment variable so that they can be accessed from any directory.

6.3 Installing Java SDK

It is very easy to install Java for use in Cygwin. The users simply download a recent version of J2SDK (the lowest acceptable version is 1.4), install it, and add the path of the Java tools to the PATH environment variable.

The Windows drives are mapped in the /cygdrive directory. The names of the sub-directories correspond to the drive letters. For example, drive c in Windows is mapped to /cygdrive/c. In this way, the users can find the path (in Cygwin) to access their J2SDK installed in the Windows environment. Placing the bin subdirectory of the J2SDK path in the PATH environment allows them to run the Java tools from any directory.

6.4 Installing PyGame

PyGame (http://www.pygame.org/), a gaming library for Python, is required by some example models of SVM, such as the CD Player and the MP3 Player. Hence, the users are highly recommended to install PyGame, though it is not necessary for SVM or SCC.

First, the users must download the PyGame source. It is available at:

http://www.pygame.org/download.shtml

The version that is used here is 1.6.

After download the source package (pygame-1.6.tar.gz), the users should extract it, and a directory named pygame-1.6 is created.

Before PyGame can be compiled, the users must manually install the SDL library for Cygwin. To do this, the following steps should be taken:

1. Install Numerical Python. The source package (Numeric-23.1.tar.gz) can be downloaded from:

   http://sourceforge.net/projects/numpy/

   After downloading the package, extract it and a directory named Numeric-23.1 is created. Go into the directory and execute the following command:

   ```
   python setup.py install build
   ```

2. Download the SDL binary for Win32 from the following location:

   http://www.pygame.org/ftp/win32-dependencies.zip

   Extract the package, and a directory named prebuilt is created.

3. Get pexports-0.43.zip from any of the following locations:

```
        http://www.is.lg.ua/ftp/gnuwin32/altbinutils/pexports-0.43.zip
    http://www.emmestech.com/software/cygwin/pexports-0.43/pexports-0.43.zip
```

Extract the package and copy `pexports.exe` to `/usr/local/bin`.

4. Create the following script (`dll2a.sh`) in `/usr/local/bin`:

```
#!/bin/sh

mkdir -p /usr/local/lib
for fn in "$@"
do
  lib=${fn%%.dll};
  dest=/usr/local/lib/lib$lib.a
  if test -f $dest
  then
    echo $dest already exists. Skipping ...;
  else
    echo "$fn ===> /usr/local/lib/lib$lib.a";
    pexports.exe $lib.dll > $lib.def && dlltool --def $lib.def --dllname $lib.dll --output
    mv lib$lib.a /usr/local/lib;
  fi;
done
```

5. Go to the `prebuilt` directory that was just generated, cd to its `lib` sub-directory. Execute the following command (assume `/usr/local/bin` is in the path):

```
dll2a.sh *.dll
```

The script generates several SDL libraries in `/usr/local/lib`.

6. Copy all the head files (`*.h`) in `prebuilt/include` into `/usr/local/include/SDL`. (Create it if the directory does not exist yet.)

7. Execute the following commands in `prebuilt/lib` to allow the execution of the `.dll` files, and copy them to `/usr/local/bin`:

```
chmod +x *.dll
cp *.dll /usr/local/bin
```

8. Go to the `pygame-1.6` directory and manually create the following `Setup` script (it is supposed to be automatically generated by `config.py`, however, `config.py` does not work in Cygwin):

```
#This Setup file is used by the setup.py script to configure the
#python extensions. You will likely use the "config.py" which will
#build a correct Setup file for you based on your system settings.
#If not, the format is simple enough to edit by hand. First change
#the needed command-line flags for each dependency, then comment out
#any unavailable optional modules in the first optional section.
```

```
SDL = -I/NEED_INC_PATH_FIX -L/NEED_LIB_PATH_FIX -lSDL
FONT = -lSDL_ttf
IMAGE = -lSDL_image
MIXER = -lSDL_mixer
SMPEG = -lsmpeg
NUMERIC = -I.



#the following modules are optional. you will want to compile
#everything you can, but you can ignore ones you don't have
#dependencies for, just comment them out

imageext src/imageext.c $(SDL) $(IMAGE)
font src/font.c $(SDL) $(FONT)
mixer src/mixer.c $(SDL) $(MIXER)
mixer_music src/music.c $(SDL) $(MIXER)
movie src/movie.c $(SDL) $(SMPEG)
surfarray src/surfarray.c $(SDL) $(NUMERIC)
sndarray src/sndarray.c $(SDL) $(NUMERIC) $(MIXER)



#these modules are required for pygame to run. they only require
#SDL as a dependency. these should not be altered

base src/base.c $(SDL)
cdrom src/cdrom.c $(SDL)
constants src/constants.c $(SDL)
display src/display.c $(SDL)
event src/event.c $(SDL)
key src/key.c $(SDL)
mouse src/mouse.c $(SDL)
rect src/rect.c $(SDL)
rwobject src/rwobject.c $(SDL)
surface src/surface.c src/alphablit.c $(SDL)
surflock src/surflock.c $(SDL)
time src/time.c $(SDL)
joystick src/joystick.c $(SDL)
draw src/draw.c $(SDL)
image src/image.c $(SDL)
transform src/transform.c src/rotozoom.c src/scale2x.c $(SDL)


#the following are placeholders. setup.py can use them to help
#auto-copy needed DLLs into the pygame installation folder.
#you can simply ignore these lines under non-windows, no need to
#comment out.
COPYLIB_smpeg $(SDL) $(SMPEG)
```

Execute the following commands to build and install PyGame (assuming that the current shell is

bash):

```
export LIBRARY_PATH=/usr/local/lib:$LIBRARY_PATH
export CPATH=/usr/local/include/SDL:$CPATH
python setup.py install build
```

After all these steps, PyGame should have been built for the Python that was used in the build. Try the following command to see if everything goes right:

```
python -c "import pygame; print pygame.__version__"
```