A Language for the Rapid Prototyping of Mobile Evolving Agents

Wolfgang Mueller, Arne Meyer, Henning Zabel
C-LAB, Fuerstenallee 11, 33102 Paderborn, Germany
wolfgang@org, henning@c-lab.de, arne@c-lab.de

Abstract

This article presents SAL, a general purpose scripting language for the rapid development of distributed software agents seamlessly embedded in a visual environment. Integrated facilities for dynamic visualization provide simple but powerful means for debugging and domain-oriented animation. SAL agents are arranged on a set of 2D worksheets which can be distributed over different machines. An agent’s program is defined by the means of a table specifying a set of state transition rules with a condition and a sequence of actions each. Beyond basic computation and communication, actions can dynamically modify the agent’s depiction, its program, and spawn arbitrary processes. A couple of examples demonstrate SAL’s applicability in various domains like electronic systems design and process management.

1 Introduction

With the wide availability of network-based solutions, development and implementation of distributed agent-based systems are of emerging interest. Despite the broad acceptance of this paradigm, dedicated development environments are less advanced. Far most of them only come with poor visual support and give only little support for rapid prototyping.

This article presents SAL (Synchronous Agent Specification and Implementation Language). SAL is a state-oriented parallel scripting language embedded in a visual environment with various built-in features for rapid prototyping and visualization of distributed systems. As a synchronous language SAL primarily addresses discrete electronic and reactive systems development.

SAL seamlessly combines a programming language with an animation description language. An SAL agent has a depiction and a behavior. Agents are placed on 2D worksheets. Multiple worksheets can be spawned as different processes on different machines. Immediately after its placement, the agent can communicate with neighbors through four predefined input and output ports. Advanced visualization can be easily assigned by the agent itself when displaying images and drawings on its rectangular surface. The behavior of an agent is specified by state-transition rules which are arranged as textual expressions and commands in a table. Each of these rules has a condition and a set of actions. Actions define local computations and communication by synchronously exchanging objects with other agents. Objects can be of basic types like long, double, string and of complex types like programs, rules, or processes. Through manipulation of complex objects, SAL provides advanced concepts for implementation of mobile and intelligent agents as well as for process management.

The remainder of this article is structured as follows. The next section discusses related works. Section 3 gives a detailed introduction to SAL. In Section 4, different examples demonstrate SAL’s applicability in various domains. After a short overview of the implementation, this article closes with a conclusion and an outlook.

2 Related Works

Multi purpose scripting languages like Perl, Tcl/Tk, and Python are attractive alternatives for quite a couple of applications. Though they typically provide efficient means for fast prototyping, their run-time behavior is mostly less satisfactory for the implementation of real-time properties which is mainly due to their more complex run-time type system. Most of popular textual scripting language are single threaded so that they are not easily applicable for the implementation of parallel programs.

Visual scripting languages are typically better suitable for parallel or pseudo-parallel systems implemen-

SAL is a multi-threaded, multi-process scripting language which come with a visual development environment. The general grid-oriented representation of SAL is inspired by Agentsheets [4]. In contrast to all published Agentsheets languages, input and output channels from non-neighbors can be directly addressed and communication can spread over different worksheets. An SAL agent can display images (including sprites) and perform general drawing operations for (colored) text, lines, circles etc.

SAL differs from Agentsheets with respect to rule representation, definition, and execution. SAL rules have the explicit notion of state and a textual syntax which is embedded in the spatial arrangement of a table. The textual syntax is derived from the synchronous language Esterel [9]. SAL supports operations on different sorts of complex objects including their programs. SAL’s synchronous execution semantics is inherited from a combination of SAM [10] and the widely applied synchronous systems description language Esterel [9].

3 SAL

SAL is a synchronous parallel, state-oriented multi-purpose scripting language with integrated animation facilities seamlessly embedded in a 2D visual development environment. SAL programs are partitioned into worksheets which can be spawned as different processes and windows on different machines.

A SAL program defines the interaction of parallel agents synchronously exchanging messages. Agents are instantiated when interactively placing them on the 2D grid of a worksheet. Immediately after its placement, an agent can communicate to its neighbors through the four predefined input and output ports.

Agents have a depiction and a behavior. The default depiction is a colored rectangle. More advanced depictions can be easily assigned by the agent itself by explicitly displaying images and drawings on its surface. The state-oriented behavior of an agent is specified by state-transition rules which are specified as textual expressions and commands in a table. The spatial arrangement of the 2D-table provides an intuitive and yet concise separation of concerns.

SAL provides additional advanced concepts for implementation of mobile agents and inter-tool communication. Program code can be easily manipulated and exchanged between agents. Dedicated commands support easy management of external processes like spawning of programs and communication via standard input and output.

The remainder of this section starts with an introduction of agent communication w.r.t. their arrangement on the worksheet. Thereafter, we outline the programming of an individual agent, its depiction as well as the underlying model of computation.

3.1 Agent Communication

SAL agents are located on worksheets. Due to the individual application, the size of each worksheet can be arbitrarily scaled with respect to the number of agents. Agents are instantiated by placing them on the 2D-grid of a worksheet so that an agent can be identified by its x/y-position on the worksheet with (0,0) origin at the upper left.

Figure 1 gives an example with two agents in default depiction at (1,1) and (1,2) and a white output agent at (1,3) displaying an integer value. The three agents are located on a 7 x 4 worksheet. Empty grid cells are indicated by a "o" in the center of a cell.

![Figure 1: SAL Worksheet with three Agents](image)

In its 2D representation, an agent has four sides: north, east, south, west. Each side has a predefined
input and output port. When placing two agents on neighbor cells, an explicit communication channel is established between corresponding ports, e.g., east output to west input and east input to west output as given in Fig. 2. Agents communicate by receiving and emitting (i.e., sending) objects through these ports.

In an agent’s program, selfX and selfY are predefined variables which return the current x and y position of the agent. This allows the specification of relative communication and limited self reflexion. As an example, program[selfX+1,selfY] returns the program of the right neighbor. Correspondingly, wsWidth and wsHeight return the width and height of the current worksheet which may vary over run-time.

![Figure 2: Two Connected SAL Agents](image)

The previously introduced facilities gives the ability to arrive very fast at a first executable specification since, for instance, no agent identifier has to be specified, no ports have to be explicitly generated, and no communication channels have to be created by drawing lines between agents. However, in order not to limit agent communication to planar structures, SAL also provides secondary means to access any remote port via textual means. The direct remote reading access is specified within the rule of an agent by additionally defining the agent’s location, and the worksheet identifier. *west[2,1,"ws1"]*, for instance, returns the current value of the west output of the agent located at cell (2,1) on worksheet ws1. Write access to ports is specified by the emit command. emit south (7), for instance, sends the integer value 7 to its south output port. emit all (7) broadcasts the value to all four output ports. To limit non-determinism, remote write access to these ports is not permitted.

For remote write access each agent has an additional “input port” which is managed as an input queue. That queue can be accessed by any agent, e.g., input [2,1,"ws1"] specifies the input queue of the agent at (1,2). Objects are non-deterministically scheduled when sending them to input at the same point in time. The order they are scheduled is due to the underlying implementation and operating system. As an example, emit input[3,0] (p) schedules object p to the input queue of the agent located at (3,0) on the same worksheet.

The secondary communication structure through input ports has been introduced to avoid time-consuming re-arrangements of agents on the grid during first prototyping. Furthermore, it supports the monitoring of remote agents in test environments. The remote access of ports and programs is intentionally controlled from textual and and not from visual means since in most of our examples, agents are addressed by their relative position so that the communication structure mostly defines a dynamically evolving network which can hardly be captured by built-in visual means.

### 3.2 Agent Depiction

Each agent has a graphical representation on the worksheet. By default, its depiction is a uni-colored 50 x 50 pixel drawing area. When executing a rule, the agent can also execute drawing commands which can draw colored points, lines, text etc. and display gif and jpeg images on its drawing area. Table 1 gives an overview of all currently supported drawing commands and their parameters.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>x, y, color</td>
</tr>
<tr>
<td>point</td>
<td>x1, y1, x2, y2, color</td>
</tr>
<tr>
<td>line</td>
<td>x, y, width, height, color</td>
</tr>
<tr>
<td>box</td>
<td>x, y, width, height, color</td>
</tr>
<tr>
<td>circle</td>
<td>x, y, width, height, color</td>
</tr>
<tr>
<td>drawtext</td>
<td>x, y string, color</td>
</tr>
<tr>
<td>picture</td>
<td>x, y, filename</td>
</tr>
</tbody>
</table>

Table 1: Drawing Commands

These commands can be used to indicate different states, to display individual values for debugging or to perform arbitrary visualizations. Figure 3 gives a SAL snapshot of the computation and visualization of the Mandelbrot set. In the lower left of the worksheet, areas of 4 x 4 agents gives the currently computed values where the white parts and light grey area indicate not computed values and currently inactive cells. The large circles on top and on the left are agents which

---

1. In our implementation it depends on the individual implementation of the underlying Java Virtual Machine including the scheduling of the Operating System.
compute the individual values. The small dots are routers which forward computed values to the display area. Active routers are indicated by a line. In Figure 3, agent (1,2) is currently passing a value from right to left.

![Figure 3: Computing the Mandelbrot Set](image)

3.3 Agent Programming

SAL basically incorporates three different sorts of agents: (i) input/output agent, (ii) programmable agents, and (iii) external agents. The latter denote the tight integration of executables compiled from a high-level programming language like C or C++. Since this article mainly concerns the embedding of textual expressions in a visual environment the remainder of this article focuses on the first two sorts of agents.

3.3.1 Input/Output Agents

Input/output agents have predefined behavior for entering and displaying values in order to easily set up a first test environment. An input agent is given by a small keypad which is launched when double-clicking the corresponding agent on the grid. The entered value is sent to all selected output ports after pressing SET. The value remains at the output until a new value is set. SAL currently provides input agents for integer and string. Figure 4 shows an input agent for integer where the value 76 is set and the east output port is activated.

An output agent displays a value when receiving it at an input port. Output agents are polymorphic with respect to the received object type. Though the function of output agents can be easily specified by programmable agents our experience shows that they are most helpful for rapid developments.

3.3.2 Programmable Agents

A programmable agent is defined by a set of state transitions which come as a set of rules. When matching the rule the given sequence of actions is executed. Each rule follows the pattern:

\[
\text{IF } (s, c) \text{ THEN } \{a_1; \ldots; a_n\} \text{NEXT } q
\]

where \(s\) defines the state in which the rule applies, \(c\) stands for a conjunction of subexpressions over local variables and ports, \(a_i\) define actions, and \(q\) gives the next state. A rule matches when the agent is in state \(s\) and \(c\) evaluates to true.

Conditions can check object types and values. Actions modify these objects. Objects can be of type long, double, string, bit, boolean, process, and program. They are declared by their first usage on the left-hand-side of assignments. The object value is given as a string preceded by its type. "long:10", for instance, returns the integer value 10. Due to the context, it is often abbreviated as 10. Input port values are accessed through the ?-operator. The emit-command sends objects to ports. Similar to other scripting languages, most operators perform type conversions. So, for instance, the statement

\[
a := \text{"double:4.3" + "long:45"}
\]

creates an object \(a\) of type double with value 49.3. SAL supports mobile code and code manipulation.

\[
\text{prg} := \text{?program[x, y, worksheet]}
\]

for instance, assigns the program at (x,y) to object prg. The command

\[
\text{emit program [x, y, worksheet] (prg)}
\]
assigns the program given by object \texttt{prog} to the specified agent at (x,y). Additional commands allow the extraction and reassignment of single rules, conditions, actions, and states as well as their manipulation.

Objects of type process support the spawning and management of external programs, i.e., processes. For example, \texttt{cmd := "process: xterm"} creates the object \texttt{cmd} of type process. \texttt{start(cmd)} then spawns an xterm under X11. Additional functions give the status of the process and support reading/writing from/to standard input/output of the spawned process.

Rules are arranged as text in table cells. Figure 2 gives the example of an agent specification with two rules. Starting at state \texttt{s1} the program toggles between state \texttt{s1} and \texttt{s2} when receiving \texttt{0} and \texttt{1} at the agent’s west input. When matching, the west input is copied to its east output. In \texttt{s2}, the variable \texttt{cnt} additionally counts state transitions. The current value of the counter is sent to the south output. \texttt{cnt} is implicitly declared as (long) integer and initialized with \texttt{0}. Note here again that \texttt{0} is short for "long:0". The second rule additionally performs a type check before accessing the value of the west port. The ;-operator in the condition defines a logical AND with lazy evaluation from left to right.

<table>
<thead>
<tr>
<th>State</th>
<th>Cond</th>
<th>Actions</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{s1}</td>
<td>\texttt{?west \texttt{==} 0}</td>
<td>\texttt{emit \texttt{east} (?\texttt{west})}</td>
<td>\texttt{s2}</td>
</tr>
<tr>
<td>\texttt{s2}</td>
<td>\texttt{?west \texttt{instanceof} &quot;long&quot;;} \texttt{?west \texttt{==} &quot;long:1&quot;}</td>
<td>\texttt{\texttt{cnt}:=\texttt{cnt}+1;} \texttt{\texttt{emit \texttt{east} (?\texttt{west})};} \texttt{\texttt{emit \texttt{south} (\texttt{cnt})}}</td>
<td>\texttt{s1}</td>
</tr>
</tbody>
</table>

Table 2: SAL Agent Definition

The representation as a 2D table provides a couple of advantages in contrast to 2D graph-based representations like StateCharts. Firstly, in state diagrams, conditions and actions are usually given as annotations to links or boxes. The available space usually does not allow the specification of long expressions and actions which are typically given in complementary, completely textual definitions. That mostly means, that the user has to combine two means to one program which gives real challenges in the treatment of larger programs. Secondly, the meaning of pure diagram-oriented means can hardly be captured when they range over several pages or when they exceed the size of a few windows. Thirdly, static analysis of a state-oriented program, for instance, by a code review, often means to analyze the relationship between the condition and the next state. A clear separation within the code greatly supports the maintenance of the code. To our experience, 2D tables are ahead of diagrammatic means since they contain all information which is necessary to understand the complete program. They can be easily printed and managed even when they range over several pages.

### 3.3.3 Computation Model

The execution of a SAL program implements synchronous communication. Synchronous communication means that all agents exchange their messages at same points in time. The SAL interpretation is a finite number of execution cycles. During one execution cycle, the SAL computation goes through two mutually exclusive phases:

1. agent execution and
2. communication.

In the first phase, all agents check the conditions of their rules of the current state where the first state in the table is assumed to be the initial one. The rules which match are selected in a list. All rules after the first one which changes the current state are removed in that list. Thereafter, all commands in the remaining rules are executed in sequential order as they are given in the table from top to bottom. When assigning a value to an output port the value remains in the output buffer until it is overwritten by a new value. After having executed the last command within the list, a possible state transition is executed and the agent suspends.

In the communication phase, when all agents are suspended all outputs of all agents are assigned to the inputs of the connected agent. Input values remain until they are overwritten by a new value. After the communication phase the interpreter executes the agents again etc.

### 4 Application Examples

SAL has been developed for the modeling of discrete electronic, reactive, and intelligent systems. Additional features support inter-tool communication or more generally-process or workflow management. The following examples sketch some SAL implementation features in these domains.
4.1 Digital Counter

The first example gives a complete program of a simple application from electronic design. Figure 5 shows four agents implementing a digital counter displaying their current value each.

![Figure 5: Digital Counter](image)

Table 3 gives the program of one generic agent which can be connected to a counter of arbitrary length. The behavior of one agent is defined by two states displaying values as text (1 and 0). When receiving a carry bit from the right (east) neighbor the program emits a carry bit to the left when in state one and changes to zero. Otherwise, no carry bit is emitted. After changing the state the agent first draws the new value.

<table>
<thead>
<tr>
<th>State</th>
<th>Cond.</th>
<th>Actions</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>true</td>
<td>drawtext(25,25,'0')</td>
<td>zero</td>
</tr>
<tr>
<td></td>
<td>(east == 'bit1')</td>
<td>emit west ('bit0')</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>(east == 'bit0')</td>
<td>emit west ('bit1')</td>
<td>zero</td>
</tr>
<tr>
<td>one</td>
<td>true</td>
<td>drawtext(25,25,'1')</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>(east == 'bit1')</td>
<td>emit west ('bit0')</td>
<td>zero</td>
</tr>
<tr>
<td></td>
<td>(east == 'bit0')</td>
<td>emit west ('bit1')</td>
<td>one</td>
</tr>
</tbody>
</table>

Table 3: Program of a Counter Agent

4.2 Flip-Flop

The next example gives the definition of an SR-flipflop embedded in a test environment with two input agent on the left and two output agents on the right (see Figure 6). The representation of the two Nor-Gates are given by two different gif-images (upper and lower Nor-Gate).

The behavior of the upper Nor-Gate is given in Table 4. At initialization, the program displays an image ("nNor.gif") and changes to state s1. Thereafter, the agent checks the values received from the left (west) and lower (south) input and sends (emits) a value to all outputs – in particular to the right and lower output. Note here, that in this example a broadcast is valid since no agent is connected to the upper port and the left output is ignored by connected input agent. The program of the lower agent is basically a copy of the upper one only exchanging the id of the gif-file and replacing south by north. The synchronous semantics of SAL helps in this example to achieve the implementation of a stable flipflop.

![Figure 6: SR-Flipflop](image)

<table>
<thead>
<tr>
<th>State</th>
<th>Condition</th>
<th>Actions</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>true</td>
<td>picture(&quot;nNor.gif&quot;)</td>
<td>s1</td>
</tr>
<tr>
<td>s1</td>
<td>&quot;west nor south&quot;</td>
<td>emit all (1)</td>
<td>s1</td>
</tr>
<tr>
<td></td>
<td>&quot;west nor south&quot;</td>
<td>emit all (0)</td>
<td>s1</td>
</tr>
</tbody>
</table>

Table 4: Behavior of a Nor-Gate

4.3 Automatic Guided Vehicles

The next example gives an application from manufacturing automation [11]. Automatic guided vehicles (AGVs) move parts between manufacturing stations for drilling, washing etc. Figure 7 gives two AGVs with the representation of a forklifter, four manufacturing stations, and one primary input and output station.

The behavior of an AGV is implemented as mobile code exchanging the program with the program of its neighbor agent. The image of the forklifter indicates the position of the AGV’s program on the grid. Cells with white dots are empty, inactive agents. The line from the lower AGV gives the direction into which the vehicle is moving. Table 5 sketches the part of the program which performs the exchange of program code. The first rule assigns the current program of the agent to object p and sets the current state of p to exchanged. When the AGV has decided to move to the west it exchanges its program with the program of its west neighbor. In a first step, it reads the program of that neighbor at location [selfX-1, selfY] and overwrites its own program by emit program(...).
Table 5: Behavior of an AGV

<table>
<thead>
<tr>
<th>State</th>
<th>Cond</th>
<th>Actions</th>
<th>Nat</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveto</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>p := setState(['program', 'exchanged'])</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td>west</td>
<td>emit program [/program [edX-1, edY]];</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>emit program [edX-1, edY] [p]</td>
<td>moving</td>
</tr>
</tbody>
</table>

The next statement assigns its program p to the specified neighbor. Due to the synchronous execution cycle both assignments are executed at the same time in the same communication phase of the execution cycle.

4.4 Inter-Tool Communication

A final example gives a program for tool management in a document workflow. Consider the workflow for compiling this HICCS paper with the tex document processing system as given in Figure 8. It starts with specifying a file name entered through the input agent at the upper left. Thereafter, an ASCII editor is launched. After quitting the editor the tex file is compiled by tex. xdvi displays the generated dvi output file. After exiting, the editor is invoked again. Concurrently, a PostScript file and a pdf file is generated. Figure 8 gives a screenshot when invoking tex which opens a separate xterm window displaying errors. When getting active each agent becomes red so that the actual workflow status can be easily monitored.

Table 6 gives the program of the tex agent. The file name is passed as a string object from the input agent. If the received object is of type string, the initial command line is assigned to p. Thereafter, setCommand appends the filename to the initial command line. The next step executes the command by starting the process in the condition. When detecting the termination of the command (not running) the file name is passed to the south agent. The south agent represented by a black vertical line with white background forwards the file name to the xdvi tool launcher etc.

Figure 7: Automatic Guided Vehicles in Manufacturing

Figure 8: Workflow for Compiling a Tex Document
5 Implementation

We have implemented a SAL prototype in Java 2 with Swing–GUI. The implementation is based on a client-server architecture with one client for each SAL worksheet. As shown in Fig.9, clients can be arbitrarily distributed over different workstations or PCs. The communication and synchronization between worksheets are managed by the SAL server via TCP/IP.

<table>
<thead>
<tr>
<th>State</th>
<th>Cond</th>
<th>Actions</th>
<th>Nat</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>west instanceed &quot;string&quot;;</td>
<td>cmd := &quot;process; res 4 latex&quot;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p := setCommand(cmd, ?west)</td>
<td>launch</td>
<td></td>
</tr>
<tr>
<td>launch</td>
<td>start(p)</td>
<td>top</td>
<td>active</td>
</tr>
<tr>
<td>active</td>
<td>not running(p)</td>
<td>emit south (?west)</td>
<td>idle</td>
</tr>
</tbody>
</table>

Table 6: Behavior of Tool Launching Agent

5 Implementation

We have implemented a SAL prototype in Java 2 with Swing–GUI. The implementation is based on a client-server architecture with one client for each SAL worksheet. As shown in Fig.9, clients can be arbitrarily distributed over different workstations or PCs. The communication and synchronization between worksheets are managed by the SAL server via TCP/IP.

<table>
<thead>
<tr>
<th>Workstation</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAL Client</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>SAL Client</td>
<td></td>
</tr>
<tr>
<td>SAL Server</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: SAL System Architecture

Figure 10 shows a screenshot of the visual SAL design environment with one input window for agent (0,1) on the left and a debugging window for the program of agent (1,1) on the right. The debugging window shows the rules on the top and the current values and types of variables and breakpoints in the lower part. Since SAL is interpreter-based, changes in the code have immediate effect on the running program. For debugging purpose each rule can be interactively enabled and disabled. The execution speed of the interpreter can be controlled by a horizontal slider at the bottom of each worksheet always indicating the required and the actual speed for the current execution cycle.

The current implementation also supports multi-threaded execution on multi-processor machines and automatic Java code generation for individual agents. The generated code is automatically compiled and can replace the agent’s behavior during run-time. Compiled code typically increases the execution time by a factor of 10-20. Through this software interface we can also easily link external Java code so that even complex Java programs can be integrated on source code level. SAL agents are stored in XML-based file format for platform independent exchange.

6 Conclusion and Outlook

We have introduced SAL, a synchronous scripting language for the rapid development of mobile agent in distributed environments with program code modification facilities. Our current examples have shown that SAL is well applicable for high-level modeling of discrete electronic systems as well as for reactive systems since these domains mostly deal with state-oriented models. Our experiments have also shown that our state-transition tables provide a much better overview in contrast to annotated graphs as soon as the behavior exceeds a certain level of complexity. So far, we have not performed complex experiments with code self modification. First results show that operators probably have to be improved to facilitate the code management.

One of the main drawbacks of SAL is its limitation to the synchronous computation model. For this we already have extended the current implementation towards an optional asynchronous execution. In another project, we have (re)used parts of the language and the interpreter for the implementation of a Petri-Net-oriented graphical capture and a combined synchronous/asynchronous simulator. Additionally, we are working on an ASM (Abstract State Machines\(^2\)) [12] editor with simulator for executable formal specifications which is derived from the same implementation. Based on these experience we are currently considering the implementation of a highly customizable simulation kernel integrating various models of

---

\(^2\)Formally known as Evolving Algebras
computation such as it is provided by the Ptolemy II framework [13].

Acknowledgements
The work described herein is partly funded by the EC through the IST project #1999-11746 E-COLLEG and by the DFG through project #1064. “Integration of software specification techniques for engineering applications”.

References