A Visual Framework for the Scripting of Parallel Agents

Wolfgang Mueller, Arne Meyer, Henning Zabel
C-LAB, Fuerstenallee 11, 33102 Paderborn, Germany

1 Introduction

Visual languages are well accepted and frequently applied in concurrent systems design. Whereas visual languages are mainly used for system specification, only little attention is paid for visual means w.r.t. integrated prototyping and dynamic visualization of parallel programs.

This article presents SAL (Synchronous Agent Language), a general-purpose grid-based 2D programming language for parallel system specification, implementation, and visualization. SAL agents have a depiction and a behavior. The behavior is defined by the means of state transition rules. The depiction is given by the drawing area on its surface. Actions in rules can execute drawing operations for dynamically changing the depiction during runtime. For a broader range of applications, SAL incorporates mobile code, code modification during runtime, and tool integration facilities.

SAL is strongly inspired by Agentsheets [3], SAM [1], and Esterel [2]. The synchronous execution model is derived from Esterel and the 3D programming language SAM whereas the visual/textual syntax is based on a combination of Agentsheets and Esterel. In contrast to Agentsheets, depictions can be arbitrary drawings modified during runtime; and rules are specified through text in tables rather than through the selection of icons. Moreover, SAL has an explicit notion of state and state transition where rules are explicitly partitioned into states.

The remainder of this article first sketches the SAL basic concepts and introduces its current implementation thereafter.

2 SAL

SAL is a synchronous parallel, state-oriented, multipurpose scripting language with integrated animation facilities seamlessly embedded in a 2D visual development environment.

Agents are instantiated when interactively placing them on a grid-oriented 2D worksheet so that an agent can be uniquely identified by its x/y-position on that sheet. Immediately after its placement, the agent can communicate with its direct neighbors through four predefined input and output channels: north, east, south, west. An agent can also directly address input and output to/from all other agents when explicitly specifying the destination.

Agents have a depiction and a behavior. The default depiction is a colored rectangle. Advanced dynamic depictions can be easily implemented by the agent itself which can explicitly display images and draw text, points, lines, circles, and boxes on its drawing area, i.e., surface. The state-oriented behavior of an agent is specified by a set of state-transition rules which are arranged in a table. Each rule follows the patterns:

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

where \( s \) defines the state in which the rule applies, \( c \) defines a conjunction of subexpressions over local variables and input channels, \( a_i \) define actions, and \( q \) gives the next state. A rule matches when the agent is in state \( s \) and \( c \) evaluates to true. Actions can manipulate or send (emit) objects to other agents. Objects can be basic like long, double, string and complex like programs, rules, and processes. Since actions can be executed on all sorts of objects, programs and their individual components like rules, conditions can be modified as well.

The execution of the program implements synchronous exchange of messages (i.e., objects) between agents. The SAL synchronous execution cycle has two phases: (i) agent execution and (ii) communication. In the first phase, all agents check the conditions of their rules. When matching a rule its actions are executed in sequential order. An action can execute a drawing operation, manipulate an object, or send an object to the input of another agent. When sending it, it is first copied into the output buffer. After all agents have completed the execution of their actions, in the second phase, all output objects are synchronously transferred to the inputs of the corresponding agents. Thereafter, rules are selected again etc.

We briefly outline SAL by the small example of a SR-flipflop composed of two Nor–Gates (see Fig.1). In this example, two Nor–Gates indicated by two different gif-images (upper and lower Nor–Gate) are embedded
in test environment with two input agent on the left and two output agents on the right. The behavior of the upper Nor-Gate is given by the following table.

<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>pictured &quot;uNor.gif&quot;</td>
<td>s1</td>
</tr>
<tr>
<td>s1</td>
<td>west nor south</td>
<td>emit all (1)</td>
<td>s1</td>
</tr>
<tr>
<td></td>
<td>not (west nor south)</td>
<td>emit all (0)</td>
<td>s1</td>
</tr>
</tbody>
</table>

The lower Nor-Gate is defined accordingly.

After initialization the above program displays an image ("uNor.gif") and changes to state s1. Thereafter, the lower Nor-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 upper output. Note here, that in this example such a multicast is possible since lower and left output are both ignored by others. Note further, that the synchronous semantics greatly helps to easily achieve the implementation of a stable flip flop.

3 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 given in Fig.2. Clients can run on different workstations or PCs. The communication and synchronization between worksheets are managed by the SAL server via TCP/IP.

![Figure 2: SAL System Architecture](image)

Figure 3 shows a screenshot of the visual SAL design environment with one input window for agent (0,1) on the left and a debugging window below. The debugger marks the currently executed rule in the upper rule editor. In the lower part, it gives a list of local object values. All values and rules can be interactively changed in that table, even when the interpreter is running. The execution speed of the interpreter can be controlled by a horizontal bar at the bottom of the worksheet.

The current implementation also supports automatic Java Code generation for individual agents. The generated code can be automatically compiled and replaces the agent’s behavior during run-time. Compiled code increases the execution by a factor of 10-20. Through this interface we can also easily link external Java code to visual agents so that even complex Java programs can be seamlessly integrated. For platform independent exchange, SAL agents are stored in XML-based file format.

![Figure 3: SAL Screenshot](image)

References

