Graphical Description and Run-Time Environments for Real-Time Software

Sanjiv Taneja
Bruce W. Weide

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210

Abstract

The STILE system (STructure Interconnection Language and Environment) provides facilities for graphically describing, and executing programs consisting of, communicating concurrent activities such as those found in real-time control software. A prototype graphical editor has been developed that allows STILE programs to be created interactively. The editor also can be used to generate a description of the attributes of the components of the graph and its interconnection structure. Two run-time systems for executing identical STILE programs have been implemented and are also described — one for a multiprocessor, and one for a network. In each, processes are distributed over several processors.

1. Introduction

A long-standing problem in real-time systems is that of constructing control software. This software is inherently difficult because of the need for concurrency as well as the real-time constraints, and most existing programming languages and environments do not deal with these issues. The STILE project [Weide 83, Weide 84, Brown 84] involves our effort to take an entirely different approach to real-time software: graphical programming. We substitute drawing pictures of software systems for the traditional text-oriented languages in order to express concurrency.

This paper describes our first "proof of principle" that a well-defined meaning can be assigned to such "graphical programs", and that they can be "compiled" into executable code. Furthermore, it explains how the STILE virtual machine (model of computation) can be implemented equally well on a tightly-coupled multiprocessor and on a loosely-coupled network system, while hiding non-essential details of the target system from the real-time programmer.

The graphical editor runs on a Sun Workstation whose graphics capabilities include fast manipulation of image data, a 1152 x 900 bit mapped display that allows graphics and text to be mixed freely, and a mouse pointing device to select items from a mouse or point at text. Its software support for menus and selections made it attractive for our purpose.

The STILE run-time system has been implemented on a multiprocessor, consisting of Intel's 86/30 single board computers connected via a Multibus. This system has been used with robotics application software and with a variety of real-time applications simulated by varying the parameters of a synthetic workload generator. The STILE run-time system has also been implemented on a network of Sun Workstations connected via Ethernet. This was done more as an exercise to evaluate the ease or difficulty of implementing STILE on a general purpose operating system, and not as a potential testbed for real-time applications. Figure 1 illustrates the overall architecture of the prototype system described in this report.

The paper is organized as follows. In Section 2, we introduce the STILE "basic model" with the help of a sample robotics application program. Creating and compiling STILE graphs using the graphical editor are discussed in Section 3. Sections 4 and 5 describe the implementation of the STILE run-time system on a multiprocessor and in a network environment, respectively. Concluding remarks are given in Section 6.

2. The STILE Basic Model

The STILE "basic model" [Weide 83] is an abstract model of concurrency useful for describing parallel programs and for building domain-dependent abstract concurrency models. The main components of this model are:

- *Blocks* or *boxes*, which are agents of computations, or processes.

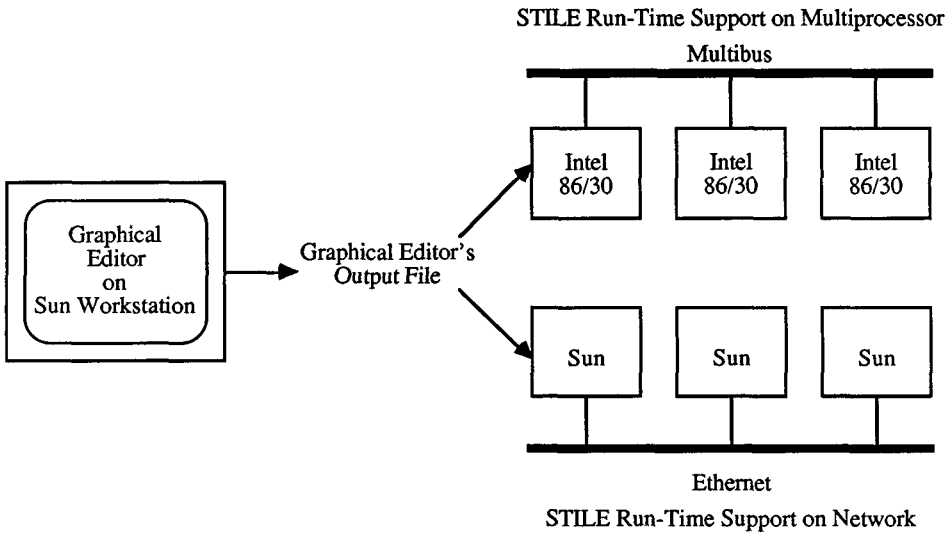

Figure 1 — Overall Architecture

- *Ports*, which are the external communication points of blocks.
- *Links*, which constitute the bindings between ports of various blocks.

A STILE program (also termed STILE graph) is developed using these components, interconnecting ports of various blocks by using links.

To help understand the STILE basic model, we first describe the characteristics of these components in more detail.

2.1. Ports

Two different types of ports are defined: data and control. Ports are also distinguished as input and output ports. A data port is an object to which data can be written (*output data port*) or from which it can be read (*input data port*), and a control port is an object to which a boolean value can be written (*output control port*) or read (*input control port*). An input data port is like a hardware register in that its value is available for use as long as it has not been overwritten. An input control port is like a status register containing a boolean value indicating the status of the port. The status of a control port can either be *poked* (the status bit is set) or *clear* (the status bit is clear).

Two data ports are considered to be compatible only if their data types are the same. Any two control ports are considered compatible. A data port is, of course, not compatible with a control port.

The following operations are defined on ports:

- Write_Port (data, port): This operation writes the specified data to the specified output data port. This has the effect of setting the data value of each input data port to which the specified output data port has been bound with a link.
- Read_Port (data, port): This operation returns the current data value available in the specified input data port. Data is always present at the port; it is not destroyed by the Read_Port operation. If no new data has been written to any output port connected to the specified input port, the most recently written value is returned. Also, in case of multiple writes between two consecutive reads, the last value written is returned.
- Poke (port): This operation is defined on output control ports only and has the effect of setting the status to *poked* for the input control ports bound to the specified output port.
- Test_And_Clear (port): This operation checks the status of the specified input control port and clears the status if the port is found to be poked.

These operations are the user's interface to the STILE virtual machine for interprocess (inter-block) communication.

2.2. Links

Binding between an output port and a compatible input port can be established by connecting them by a *link*. This binding results in values written to an output port at run-time being transmitted to the input port it is connected to. More than one output port may be connected to the same input port, and

an output port may be connected to more than one input port.

2.3. Blocks

A block or box represents a sequential process that operates independently of other blocks in a STILE program. It contains a set of procedures written in a high level language such as C or Pascal, and may have zero or more ports of each type for communicating with other blocks. Within a block, each port has a local name. Sharing of data among the procedures in a single block is permitted. These procedures are written without regard to what the block interacts with, as all communication uses the port operations.

The STILE blocks described above are also termed *primitive* blocks, as opposed to *composite* blocks. The latter are constructed by combining primitive blocks. The ports of a composite block are a subset of the union of the ports of its constituents. Composite blocks help in hierarchical design, but are not provided by the prototype graphical editor described below.

Some mechanism is required for selecting a procedure for execution when the process represented by this block is to be run. The STILE basic model binds a procedure of the block to each input control port. A monitor performs the following job: it continually examines the status of the input control ports of the block. When it finds a port that has been poked, it clears the status of the port, executes (to completion) the procedure associated with it, and returns to examine the input control ports again. The order in which the ports should be examined is not specified. This leaves a choice of policies for scheduling activities within a block. The monitor has thus been dubbed a *micro-scheduler*, because of the analogy of the multiple streams of execution to light-weight processes (*micro-processes*) [Schwan 85] that have very little state information associated with them.

The code for the micro-scheduler must be included in the code for the block, and makes use of the Test_And_Clear operation. The procedures it calls use the Read_Port, Write_Port, and Poke operations.

2.4. An Example Scenario

Here we illustrate the use of STILE in an admittedly rather trivial real-time control application. The application consists of controlling a robot arm with three orthognal axes, each of which is manipulated by a separate motor. Given the coordinates of the point where the arm is to be positioned, the controller should be capable of generating the appropriate inputs to the motors so that the arm is positioned at the desired location. The synchronization constraint is that the movements in the three axes must have been completed before the next desired position is input. Solutions to a similar problem in three concurrent language systems, namely Pascal-Plus, occam, and Edison have been used to evaluate the effectiveness of these

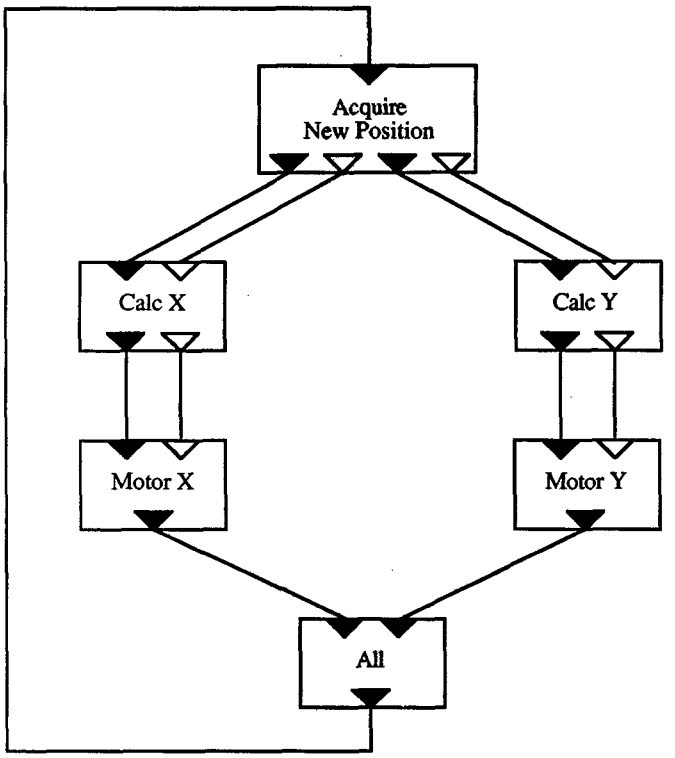

Figure 2 — A STILE Program for Controlling a Robot Arm

languages [Kerridge 84]. This is the reason we have chosen this example — not because it illustrates anything particularly unusual about STILE, but because it can be more easily contrasted to solutions in other languages. As one of the referees pointed out, STILE is not at its best when describing an essentially sequential program (such as this one), as the result is similar to a flow-chart; see [Brown 84] for other examples.

The corresponding STILE program is depicted in Figure 2, in which we have shown only two of the three dimensions of the problem for simplicity. The "Acquire New Position" box accepts input from the operator's console. The input is in the form of three integers corresponding to the x, y, and z coordinates of the desired position of the robot arm. It then writes these values on its output data ports. This results in the coordinates of the desired position becoming available to the motion calculation processes "CalcX", "CalcY", and "CalcZ" on their input data ports. In addition, it pokes its output control port, which has the effect of initiating the motion calculation processes. These processes now start executing in parallel to compute the number of steps and the direction in which each motor should be moved. On completing their computation, each of these processes writes the motion value on its output data port and pokes its output control port. This makes the motion

values available to the three motor driver processes, "MotorX", "MotorY", and "MotorZ" and activates their micro-schedulers. These processes generate the inputs to the motor for moving it a given number of steps and in a given direction. (Note that the physical motors themselves communicate with the motor control blocks, but this communication is done directly through external device hardware connections and not via STILE ports. It therefore does not appear in the STILE graph.)

After the arm movement for which the motor driver process was responsible has been completed, it pokes its control output port, thus activating the "All" box. This box detects the completion of *all* of a set of actions, which in this case is the movement of the three motors. When all its control inputs have been poked, it pokes its output control port, thus waking up the micro-scheduler of the "Acquire New Position" process, which can now accept a new set of input. Thus the decoupling of data and control in the STILE basic model has helped us to realize the synchronization constraints in a straightforward way. The graphical representation also has the advantage of clearly displaying the dependencies between processes cooperating to perform a common task, and the potential concurrency.

3. Writing STILE Programs

Developing STILE programs is a two step process:

- Writing code for the boxes in a high level sequential language.
- Specifying a graphical description and feeding the information about the interconnection structure of the STILE graph and about the attributes of its components to the STILE run-time system.

In this section we discuss how code is written for STILE boxes, and how STILE programs can be created and compiled using the graphical editor and then executed in a multiprocessor or a distributed system.

3.1. Writing Code for a STILE Box

Each box must contain a call to an initialization procedure, *stile_init*, as the first line of its code. The purpose of the initialization procedure is to set up the data structures used by the STILE run-time system (see Sections 4-5). The code inside a box contains a micro-scheduler and a set of procedures written in a high level language such as Pascal or C.

The template for writing the code for a STILE box in C looks as follows:

```
main()
{
  stile_init (boxname);
  <Code for micro-scheduler>;
  <Code for procedures of the box>;
}
```

Referring to the example scenario presented earlier, and using a round-robin micro-scheduling scheme, the code for the box named "CalcX" in Figure 2 looks as follows:

```
/* file containing code for run-time
   procedures */
#include "stile.h"

/* Port Definitions */
#define NEWPOS 1    /* Input data
                       port */
#define MOTION 2    /* Output data
                       port */
#define CALC   1    /* Input control
                       port */
#define DONE   2    /* Output control
                       port */

main()
{
  stile_init("CalcX");
    /* Code for the microscheduler */
  for (;;)
  {
    if test_and_clear (CALC)
      compute_motion();
    sleep_until_poked;
      /* the sleep call above is only
         needed in the multiprocessor
         version; see Section 4 */
  }
}

/* Code for procedures of the box */
compute_motion()
{
  static int oldpos_x;
  int newpos_x, motion_x;
  read_port(NEWPOS,&newpos_x);
    /* compute motion from the new and
       the old position; details of
       this are not presented here */
  write_port(MOTION,motion_x);
  poke(DONE);
    /* update old position */
  oldpos_x = newpos_x;
}
```

As the above example shows, writing code for a STILE box is no harder than writing a high level language program. It is even possible to generate the code for the micro-scheduler automatically, since the underlying model for each box is the same, but the present system does not do this.

3.2. Creating STILE Graphs

Once the sequential code for the boxes has been written, the graphical editor can be used to draw the components of a STILE graph (namely, boxes, ports,

and links) and to interconnect the boxes together. The display screen is divided into three windows: a graphics window where a STILE graph can be drawn, a prompts and command arguments window, and a window for displaying error messages. The user enters commands by selecting items from a pop-up menu. Command arguments are entered by pointing on the screen and also using keyboard input.

A box is represented as a rectangle and a port as a triangle drawn on an edge of a box. For an input port the triangle is drawn with its head pointing into the box, and pointing out in case of an output port. A control port is distinguished from a data port by being filled in solid black. Output ports can be connected to input ports by one or more straight line segments. Selective erasure of boxes, ports and links is supported. Boxes and ports can also be moved around on the screen.

The graphical objects on the screen are not considered simply bitmaps. Rather they have semantics associated with them. This is reflected in the constraints imposed on graphical construction. For instance, an attempt to connect ports of incompatible types is flagged as an error, and the pop-up menu displayed depends on the region of the screen and the component in which the mouse was positioned at the time the menu was requested.

Once a graph is drawn, it can be saved in a file. The complementary operation of drawing a graph by reading its description from a file can also be carried out. Finally the graph can be compiled to generate a file containing the information needed by the run-time system. This includes box and port attributes, which are filled in by the user with a menu in the prompts window.

3.3. Compiling a STILE Graph

A STILE graph can be compiled to generate the information about the attributes of its components and its interconnection structure. The information generated is divided into five parts:

- Attributes of Boxes: The attributes of a box consist of an integer used solely as a box identifier (generated automatically), its name, the processor on which it is to be run, the timeslice interval for which gets the processor, its scheduling priority, and the stack size it requires.
- Input Data Port Attributes: An input data port is characterised by the following attributes: the number of the box to which the port belongs, the port identifier (an integer), the port data type, and its initial value.
- Input Control Port Attributes: An input control port has the following attributes: the number of the box to which the port belongs, its identifier (an integer), and an initial value.
- Data Port Interconnections: The data port connections are specified as a pair of <box number, port number> pair.
- Control Port Interconnections: These are specified in a manner similar to the data port interconnections.

4. Run-Time Support for STILE on a Multiprocessor

The multiprocessor available to us for this research consists of sixteen tightly coupled Intel 86/30 single-board computers, each containing an Intel 8086 microprocessor, an 8087 floating point co-processor, and 128KB of memory. Access to global memory is done via a Multibus. Not all memory is shared; each processor has 96KB of local memory and 32KB of global shared memory.

Operating system support was provided by a special purpose operating system, Generalized Executive for real-time Multiprocessor applications (GEM) developed by our research group in The Ohio State University's Computer and Information Science Department, to address the requirements of real-time software [Schwan 85]. The design of GEM was influenced by the requirements of the STILE run-time system, but it was not tailored to STILE.

For process scheduling, GEM provides various operations by which processes can control themselves or be controlled by system or application dependent *scheduler processes.* A process can put itself to sleep for a specific time or until the occurrence of a particular event, or both, and it can be awakened by other processes. These scheduling facilities had a direct bearing on the implementation of the micro-scheduler for a STILE box. In particular, the micro-scheduler did not have to perform a busy wait when none of the input control ports of the box were in a poked state.

For process communication, GEM supports three models of communication, namely, asynchronous with data loss, synchronous without data loss, and synchronous or asynchronous with possible loss of aged data. To realize these models, it uses *mailboxes*, which have global identifiers and a pool of free buffers to which data can be written or from which it can be read.

The STILE run-time system establishes the initial set-up of the processes, and implements the inter-process communication primitives of STILE. The interface to the run-time support is in the form of the following operations:

- read_port (port, data)
- write_port (port, data)
- poke (port)
- test_and_clear (port)

The semantics of these operations has already been explained in an earlier section.

The input to the run-time system is the output of the graphical editor: a description of the attributes of the components, and the interconnection structure of the STILE graph. The initial set-up performed by the *stile_init* procedure consists of the following:

- It creates a process for each box in the STILE graph. Process creation is parametrized with respect to the processor name on which the process is to be run, the time-slice interval for which it runs before being descheduled, the stack size it requires, and its scheduling priority. All these parameters are attributes of a box and hence available from the input to the system.
- It creates globally accessible tables for storing information about input data ports, input control ports, data port connections, and control port connections.
- For each input data port, it creates a mailbox and stores the mailbox identifier in the data port table.
- For each input control port, its initial value is entered in the control port table.
- The data and control ports connection information is translated to mailbox identifiers and stored in the data port connection table and control port connection table, respectively.

The information thus stored is used by the run-time system in executing the read_port, write_port, poke and test_and_clear operations. In one of the models of communication supported by GEM, the read operation on a mailbox returns the most recent data written to it, even if it has been read before. This made the implementation of an input data port and its associated read_port operation straightforward. A single *sticky* mailbox was associated with an input data port. The local name of the input data port was mapped to the global identifier for a mailbox.

5. Run-Time Support for STILE in a Distributed System

The STILE distributed run-time system runs on Sun Workstations connected via a 10 Mbit/sec Ethernet. The multiple windowing environment supported by these workstations is very well suited for monitoring STILE programs. New shells are easily created, each allocated to one process. This makes the interaction among running processes visible to the user.

The Sun Workstations run the 4.2 BSD UNIX[1] operating system. The features of this UNIX especially relevant for our purposes are its interprocess communication facilities: sockets as a building block for communication. As described later, the run-time system uses sockets to implement some of its operations.

As in the case of the multiprocessor, the STILE run-time system in a distributed environment establishes the initial set-up of the processes, represented as boxes in a STILE graph, and implements the inter-process communication primitives of STILE. The interface to the run-time support is in the form of the following operations:

- read_port (port, data)
- write_port (port, data)
- poke (port)
- test_and_clear (port)

As can be seen the interface presented to the sequential code for a box is uniform in the two run-time environments.

The graphical editor's output consisting of a description of the attributes of the components, and the interconnection structure of the STILE graph, is fed to the run-time system. The initial set up phase is performed by the *stile_init* procedure, which does the following:

- It creates a UNIX process for each box on the processor whose name is included as an attribute of the box.
- For each input port (of both types, data and control), a UNIX socket is created. The socket is bound to a name using a system-wide name space. This allows data to be sent to this socket by referring to it by its name.
- A socket is also created for each output data and control port. However, in this case, the binding of sockets to a name is not done, because only the process that created the output socket needs to access it, and the local identifier obtained as a result of creating this socket is sufficient for this purpose.
- The interconnection information is translated in terms of socket names and distributed in tables local to each process.

This information, along with the underlying communication facilities offered by the operating system, is used to execute the read_port, write_port, poke, and test_and_clear operations.

The read_port and test_and_clear implementations deserve special mention because sockets involve queueing. The communication protocol of the STILE basic model specifies that a read operation on a data port should return the most recent data if there is no new data present. So if there have been multiple writes be-

[1]UNIX is a trademark of AT&T Bell Laboratories.

tween two consecutive reads, all the data should be "pumped" out of the socket and the last value returned. However, a receive operation on a socket destroys the data present in the socket and blocks if there is no data available. To implement the read_port operation, the socket representing the input data port was made non-blocking and the following code used to make the data "stick" to the port. The same method is used for test_and_clear.

```
{ . . .
  static char *buf;
  . . .
    /* remove  all data from socket
       "s" before returning */
  while (recvfrom(s,data,buflen,0,
      &from,&fromlen) > 0) ;
    /* "stick" the data to the port */
  strcpy(buf, data);
  . . .
  }
```

The static data at *buf ends up holding this port's data value. The parameters of the "recvfrom" system call used above are the socket descriptor from which the data is to be read, a pointer to the buffer in which the data is to be stored, the length of the buffer, a flag to peek at incoming data, a buffer in which the sender's address is to be filled, and its length.

6. Summary

We have described the implementation of a prototype graphical editor for interactively developing STILE programs, and two run-time systems for executing them on a distributed system and a multiprocessor. The graphical editor has been designed with an eye toward extensibility. Being a prototype implementation, it does not support advanced features such as hierarchy of components, and automated routing while connecting components. However, it does provide minimal features to serve as a useful tool for developing STILE programs. Other members of the STILE research group, Gregor Taulbee and Michael Stovsky, are addressing these issues in a more advanced version of the graphical editor that is based on a different user interface and incorporates other new features.

The provision of the graphical editor and the run-time support makes it easier to construct real-time software systems from sequential programs, and facilitates linking, loading and execution on a multiprocessor and in a distributed environment. Our goal was to demonstrate the feasibility of graphical programming and automatic compilation for execution on parallel hardware. This having been accomplished, we look forward to more extensive testing on real examples, and to more advanced graphical editing features.

Acknowledgements

We would like to express our thanks to Professor Karsten Schwan, Tom Bihari, Gregor Taulbee, Michael Stovsky and Clayton Elwell of the Department of Computer and Information Science at The Ohio State University for their help and encouragement.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and monitored by the Army Tank Automotive Command under contract number DAAE07-84-K-R001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either express or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Mr. Taneja is currently with AT&T Bell Laboratories, Murray Hill, NJ. None of the material in this paper is associated with his work at AT&T Bell Laboratories.

References

[Brown 84] Brown, M.E., and Weide, B.W. Automating process-to-processor mapping under real-time constraints. In *Proc. 1984 Real Time Systems Symposium.* IEEE, Dec., 1984.

[Kerridge 84] Kerridge, J.M., and Simpson, D. Three solutions for a robot arm controller using Pascal-Plus, occam and Edison. *Software Practice and Experience* 14(1):3-15, Jan., 1984.

[Schwan 85] Schwan, K., Bihari, T., Weide, B.W., and Taulbee, G. GEM: Operating system primitives for robotics and real-time control. In *Proc. IEEE Intl. Conf. for Robots and Real-Time Control.* IEEE, Mar., 1985.

[Weide 83] Weide, B.W. *Graphical Programming of Process Control Software Using STILE: A Progress Report and Future Directions.* Technical Report, Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, OH, Sep., 1983.

[Weide 84] Weide, B.W., Brown, M.E., Ramanathan, J., and Schwan, K. Process control: integration and design methodology support. *Computer* 17(2):27-32, Feb., 1984.