Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

SIMULATION AND CONTROL OF REACTIVE SYSTEMS

Pawel Gburzynski

Department of Computing Science
University of Alberta
Edmonton, Alberta, CANADA T6G 2H1

Jacek Maitan

Tools For Sensor, Inc.
513 Marshall Avenue
Carmichael, CA 95608, U.S.A.

ABSTRACT

We introduce SIDE (the acronym stands for Sensors In a Distributed Environment)—a software package for developing control programs for reactive systems. One distinctive feature of SIDE is that it can be used as a simulator: some (or even all) components of the underlying physical network can be virtual. Notably, the control program itself need not be aware that some parts of its environment are not real. SIDE applications can be naturally distributed and interconnected via the Internet.

1 INTRODUCTION

These days people start doing remotely many things that traditionally have required their physical presence at the processing site, e.g., shopping, banking, conferencing, learning. There is no reason why remote supervision/coordination of manufacturing processes should be excluded from the list. To implement this idea, we have to change our attitudes toward the organization and interface of control networks. First, instead of being based on obscure and "internal" protocols incompatible with anything used outside, such networks should be naturally connectible to the Internet. Second, control programs driving these networks should be expressed in a friendly common language providing a unifying platform for interoperability, accessibility, and understanding. These postulates are now finding their way into industrial reactive networks (see IEEE P1451.1/D83, 1996).

The software package presented in this paper offers a number of tools aimed at fulfilling the postulates mentioned above. Interestingly, SIDE is a direct descendant of a network simulator (Dobosiewicz and Gburzynski 1993, Gburzynski 1996), to the point of retaining all the simulation features of its predecessor. Specifically, SIDE offers:

- A programming language for describing network configurations and specifying distributed programs organized into event-driven threads.
- A kernel for executing programs expressed in the language of SIDE.
- A Java interface (the DSD applet) that can be used to monitor the execution of a SIDE program from the Internet.
- A TCP/IP daemon interfacing physical networks of sensors and actuators to the Internet. This way such networks become visible to the SIDE kernel.

The SIDE kernel has two modes of operation. In the real mode, the events perceived and triggered by the threads (SIDE processes) occur in actual time (usually some of them are triggered by real sensors and some of them affect the behavior of real actuators). In the virtual mode (which is only possible if the entire environment of the control program is modeled), the time is virtual and the control program behaves as an event-driven, discrete-time simulator.

2 THE STRUCTURE OF SIDE

2.1 Control and Simulation

For the purpose of simulation, the source program in SIDE is logically divided into three parts. The *protocol part* represents the dynamics of the modeled system. The *network part* is a logical description of the hardware on which the protocol program is expected to run. Finally, the *traffic* part describes the load of the modeled system, i.e., the events arriving from outside and their timing.

The terms "protocol" and "traffic" reflect the fact that the primary application of SIDE's predecessor was simulating communication networks (e.g., see Bertan 1989, Dobosiewicz and Gburzynski 1992, Gburzynski 1996, Molle, 1994). However, it still makes sense to call a control program driving a network of sensors and actuators a *protocol*, because, owing to its reactive nature, such a program looks like a set of rules prescribing actions to be taken upon some specific events that may be coming from several different (and distant) sources. Similarly, it makes sense to talk about the input *traffic* in a (simulated) fragment of a reactive system, because such a system typically handles some objects arriving from outside, and it is natural to represent such objects as structured *packets*.

For the purpose of developing control programs in SIDE, we adopt a slightly more elaborate view of the source program (Figure 1). The protocol consists now of two parts: the control program proper and the simulator for the virtual components of the driven system. Similarly, the network part is split into the so-called *network map*, i.e., the mapping of logical sensors and actuators perceived by the control program onto their real (or simulated) counterparts, and the description of the modeled fragments of the underlying hardware, i.e., the hardware used by the simulator part of the protocol. The traffic specification only applies to the simulated part of the environment (real fragments handle real traffic that need not be specified).

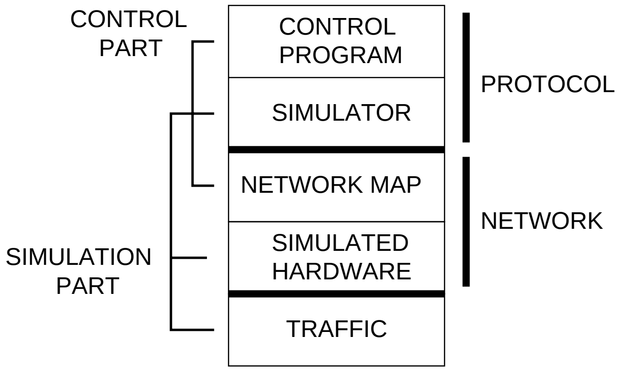

Figure 1: A Control System in SIDE.

With the above view, one can separate the software components that belong to the control system from those belonging to the simulator. Thus, "control program" + "network map" comprise the actual control system (this part represents the target of the development process, with the network map interpreted as a parameterization of the control program), while the remaining components will tend to vanish, until they ultimately disappear altogether in the complete version of the system.

2.2 Network Interface

A reactive system is defined as a collection of sensors and actuators. These two objects are very similar; in fact they are both described by the same data structure with the following layout:

```
mailbox Sensor {
  private:
    int Value;
    void mapNet ();
  public:
    NetAddress Reference;
    void setValue (int);
    int getValue ();
    void setup (NetAddress&);
};
mailbox Actuator : Sensor { };
```

The base type of `Sensor` and `Actuator` is `Mailbox`, which is one of the fundamental built-in data types in SIDE. The only relevant attribute of a `Sensor/Actuator` is its `Value`. For a sensor, the value represents the sensor's perception of its environment. The sensor mailbox triggers an event whenever its `Value` changes. For an actuator, the value describes the action to be performed by the actuator. By changing the `Value` attribute of the actuator we force it to carry out a specific physical operation.

The `setup` method plays the role of a *constructor*. Its argument specifies the sensor's coordinates in the controlled network. These coordinates may be interpreted as an actual network address (if the sensor/actuator has a physical counterpart), or they may be used to identify the object's model in a simulated fragment of the system. This mapping is carried out by method `mapNet` whose implementation belongs to the network map portion of the protocol program.

The actual mapping of a logical sensor/actuator into its real physical counterpart consists of two steps. The lower-level portion of this mapping is carried out by a daemon that interfaces a physical network of sensors/actuators to the Internet. The daemon acts as a server intercepting all status change events in the sensor network and transforming them into TCP/IP packets sent to the clients. Similarly, it receives status change requests from its Internet clients and transforms them into new values of actuators. The second level of mapping is performed by the network map portion of the protocol program in SIDE. This part is in fact optional, but highly recommended. Technically, its is possible to implement `setValue` and `getValue` in such a way that their functions correspond directly to daemon requests. It makes better sense, however, to keep these functions simple and generic, and impose one more level of software mapping. For example, the same logical sensor/actuator may be mapped differently in different versions of the control program (e.g., it may be simulated in some versions or mapped to a physical sensor/actuator in others). Another advantage of the extra level of mapping is the flexibility of being able to map one logical sensor/actuator into multiple physical sensors/actuators and vice versa. This way the same control program may be easily "recycled" in environments slightly different from the target one, which makes it easier to follow the *pattern approach* in its design (Gamma et al. 1994).

A control program in SIDE can be implemented as a single (multi-threaded) program, or as a set of embedded programs (Edwards et al. 1997, Paulin et al. 1997) run on independent (possibly diverse) machines connected via a network. These modules can communicate with operators (human supervisors) on other machines via standard Internet browsers capable of running Java applets.

3- THE PROGRAMMING LANGUAGE

3.1- Program Components

Typically, a program in SIDE consists of a number of source files. The basic unit of execution is called a *process* and it looks like a specification of a finite state machine. A process always runs in the context of some *station*, which conceptually represents a logical component of the controlled/modeled system. One process (`Kernel`) and one station (`SYSTEM`) are predefined and exist throughout the entire lifetime of the program. All other stations and processes must be declared and created by the program. One user process, called `Root`, is run by the kernel automatically; its role can be compared to the role of `main` in a C (or C++) program.

Besides processes and stations, SIDE offers a variety of built-in types, including tools for creating models of network channels (types `Link` and `Port`), traffic generators (`Traffic`, `Client`, `Message`, `Packet`), alarm clocks (`Timer`), and generic process synchronization tools (`Mailbox`). Objects of the last type can be bound to TCP/IP ports, providing a reactive interface to the Internet.

Stations (and also links and ports) represent the static components of the program, i.e., the logical view of the hardware on which the control program or simulator is run. These objects are typically created at the very beginning (by the `Root` process) and remain present throughout the lifetime of the program. Processes are more dynamic: it is not uncommon to create (and destroy) them dynamically for various intermediate tasks. Links and ports are mostly used in simulators—to model the passage of packets through some "channels."

All objects that exhibit dynamic behavior are dubbed *activity interpreters* (AI). Such objects are capable of generating events that can be awaited and perceived by processes. For example, whenever something is deposited in a mailbox, the mailbox triggers an event that a process interested in monitoring the mailbox contents can perceive and respond to. Similarly, an event is triggered by `Timer` when an alarm clock goes off. Processes are also capable of triggering some events; this possibility can be used as a direct means of inter-process communication (without the mediation of mailboxes). In contrast, stations do not exhibit any activities of their own; they do not trigger any events by themselves, and they are not activity interpreters.

3.2- Processes

A process consists of its private data area and a possibly shared code. Besides accessing its private data, a process can reference the attributes of the station owning the process, and some other variables constituting the so-called *process environment*. Processes can communicate in several ways, even if they do not belong to the same station.

A process type usually defines a number of attributes (they can be viewed as the local data area of the process), an optional setup method (a constructor), and the `perform` method specifying the process code. A process type declaration has the following syntax:

```
process ptype :␣ supptype (fptype, stype) {
   ... attributes and methods ...
   setup (...)  { ...  };
   states {s0, s1, ..., sk};
   perform { ...  };
};
```

where `ptype` is the name of the declared process type, `supptype` is a previously defined process type, `fptype` is the type of the process's *parent* process, and `stype` is the type of the station owning the process. As for other SIDE types, `supptype` can be omitted if the new process type is derived directly from the base type (`Process`). The two arguments in parentheses are also optional: they can be skipped if they are not useful to the process.

A process code method resembles the description of a finite state machine (FSM). The `states` declaration assigns symbolic names to the states of this machine. The first state on the list is the initial state.

The operation of a process consists in responding to events. The occurrence of an event awaited by a process wakes the process up and forces it into a specific state. Then the process (its code method) performs some operations and suspends itself. Among these operations are indications of future events that the process wants to perceive. A typical code method has the following structure:

```
perform {
   state s0:
      ...
   state s1:
      ...
};
```

Two built-in pointers are available to the code method: `F` (of type `fptype`) pointing to the process's parent, and `S` (of type `stype`) pointing to the station owning the process.

Processes in SIDE are executed as threads with very simple preemption rules. If we ignore process creation (when the created process is run for the first time), a process is always run in response to some event triggered by one of the activity interpreters. The first event starting a process is assumed to be triggered by the process itself; thus, in fact, there are no exceptions. One common element of the interface between an AI and a process is the AI's `wait` method callable as *ai*`->wait (`*ev, st, pr*`);`.

The first argument of `wait` identifies an event; its type and range are AI-specific. The second argument is a process state identifier: upon the nearest occurrence of the indicated event the process will be awakened in the specified state. Finally, the last (optional) argument gives the *priority* of the wait request. If the priority is absent, a default value (average priority) is assumed.

A process may issue a number of wait requests, possibly addressed to different AIs, and then it puts itself to sleep, either by exhausting the list of statements associated with its current state or by executing `sleep`. All the wait requests issued by a process at one state are combined into an alternative of waking conditions: as soon as any of these conditions is fulfilled, the process will be restarted in the state indicated by the second argument of the corresponding request. The priority argument indicates the precedence of events that occur simultaneously. This priority is interpreted globally, among all processes that perceive events at the current moment.

When a process is awakened, it always happens because of exactly one event. If the process has been waiting for other events, the pending wait requests are erased and forgotten. The process is awakened by the earliest of the awaited events. If several events are triggered at the same time, the event with the highest priority is selected. If several earliest events have the same priority, one of them is chosen at random. There is a way of eliminating this non-determinism (it exists because SIDE is also a simulation system) and assigning priorities to such events implied by the order of their perception by the SIDE kernel.

Once a process has been awakened, it will not be preempted until it decides to put itself to sleep. It is assumed that processes are strongly I/O bound (using the operating systems terminology), and the non-preemptive, declared-priority scheduling policy used in SIDE is appropriate for their pattern of activity. By enforcing the FSM structure of the process code method, SIDE forces its threads to be organized as fast-responding interrupt processors. If there is a computationally intensive task to be performed in a SIDE process, it is natural to split such a task into a chain of interrupts communicating via the IPC mechanisms offered by the SIDE kernel. Each of those interrupts is non-preemptible, but their sequence is subject to priority scheduling that accounts for the importance of other tasks. One should notice here that computationally intensive tasks are not typical in SIDE. If there is a true demand for number crunching in a SIDE system, the best way to include this capability is to set up a CPU server running the CPU-bound tasks and communicating the results to the SIDE kernel via a networked mailbox.

3.3 Time in SIDE

A SIDE program uses its internal notion of time. This internal time can be mapped to real time (which must be done if there is at least one real piece of equipment controlled by the program), or not (in which case the program behaves as an event-driven, discrete-time simulator).

Time intervals are expressed in the so-called ITUs (*indivisible time units*) and represented as objects of type `TIME`. The precision/range of `TIME` is selected by the programmer; there is no explicit limit on this precision. By default, when SIDE is set up to work in real time, the ITU is mapped to one microsecond. If required, type `TIME` is implemented using multiple-precision integer arithmetic.

Besides the ITU, SIDE defines another unit of time, the so-called ETU, which stands for the *experimenter time unit.* The reason for this duality is that the ITU (which determines the internal granularity of time) may not be convenient for the the human operator. By default, in the real-time setting of SIDE, the ETU is mapped to one second.

3.4 Mailboxes and Other IPC Tools

Processes in SIDE can communicate in three different ways. First, they can take advantage of the fact that they are themselves activity interpreters capable of triggering events. Thus, it is legal for a process to issue a wait request for another (or even the same) process to get into a specific state. Another IPC tool is signal passing. Each process has a *signal repository* that can be used to receive signals (simple messages).

The third and most flexible IPC mechanism is communication via mailboxes. A generic mailbox is a repository for possibly structured messages whose arrival may trigger various events. Below we list the implementations of the two public methods of `Sensor` and `Actuator`.

```
void Sensor::setValue (int v) {
  Value = v; put ();
};
int Sensor::getValue () { return Value; };
```

The second method is trivial, but the first one, having modified the `Value`, executes `put`, which is a standard mailbox operation used to deposit an object in the mailbox. In our case, the object is dummy: `put` has no argument and its only action is to trigger a `NEWITEM` event. This event will be perceived by the process (or processes) monitoring the changes of the sensor value.

4 EXAMPLES

4.1 Stations

For illustration, let us consider a system of conveyor belts. Each unit of our conveyor system is equipped with a motor (a switch actuator) driving the unit and a number of sensors detecting the presence of objects (boxes) passing through the unit. In agreement with the object-oriented paradigm of SIDE, all these objects can be represented as stations descending from a single station type capturing the common structure of all units. This common station type can be declared as follows:

```
station Unit {
  Actuator *Motor; MotorDriver *MD;
  Alert *Exception;
  void setup (NetAddress&, double);
};
```

Type `Actuator` has been presented already. `Alert` is another descendant of `Mailbox`, whose role is to pass alerts (messages about the abnormal behavior of the unit) to the operator. `MotorDriver` is the type of the process that will be responsible for the operation of the unit's motor.

When a `Unit` is created, its setup method (the constructor) receives the network address of the motor actuator and a value representing the inertia of the motor. This value will be used by the motor driver process to make sure that the motor is not switched on and off too fast. When we talked to people using conveyor systems driven by commercial software, they complained about the jerky behavior of the motors triggered by intermittent spurious sensor signals. Consequently, we have decided to mediate all references to sensors and actuators through simple processes whose sole purpose is to dampen the rate of status changes.

Below we list the station type representing a segment with one entry and one exit.

```
station Segment : Unit {
  Sensor *In, *Out; SensorDriver *SDIn, *SDOut;
  int BoxesInTransit;
  void setup (NetAddress&, double,  // Motor
    NetAddress&, double,  // Entry sensor
    NetAddress&, double,  // Exit sensor
    double);  // Upper bound on passage time
};
```

The setup method of `Segment` accepts seven arguments describing the parameters of one actuator (the motor switch) and two sensors (one sensing boxes entering the belt, the other monitoring the output end of the segment). The `double` argument associated with a sensor/actuator specifies its inertia, i.e., the amount of time for which a new condition (value) must persist to be considered valid. The last argument is a bound (in seconds) on the amount of time needed by a box to travel through the segment. It will be used to diagnose jams.

This is the actual code of the setup methods announced in the two station types:

```
void Unit::setup (NetAddress &mr, double inr) {
  Exception = create Alert (getOName ());
  Motor␣    = create Actuator (mr);
  MD␣       = create MotorDriver (mr, inr);
};
void Segment::setup (NetAddress &mr, double mi,
                     NetAddress &en, double ei,
                     NetAddress &ex, double xi,
                     double TransitTimeBound) {
  Unit::setup (mr, mi);
  BoxesInTransit␣ = 0;
  In␣   = create Sensor␣  (en);
  Out␣  = create Sensor␣  (ex);
  SDIn␣ = create SensorDriver (In, ei);
  SDOut = create SensorDriver (Out, xi);
  create SegmentDriver (TransitTimeBound);
};
```

The setup methods create the needed components of the station, i.e., the mailboxes and processes. This is accomplished by operation `create` whose arguments are passed to the setup method of the created object.

Method `getOName`, invoked to produce the argument of `Alert`'s setup method, returns a character string representing the name of the current object. This way, the alert will be tagged with the name of the segment, and the operator will be able to tell the source of the messages appearing on the screen. Objects in SIDE have several kinds of naming attributes that can be used to identify them for the purpose of displaying their status by DSD.

4.2- Processes

Let us start from the sensor driver process, which dampens the rate of changes in the sensor value, so that it is kept below the specified inertia. This process is declared as follows:

```
   process SensorDriver (Unit) {
     Sensor *TheSensor; int LastValue;
     TIME Inertia, Resume;
     void setup (Sensor *s, double inertia) {
       LastValue = (TheSensor = s)->getValue;
       Inertia = (TIME) (Second * inertia);
     };
     states {StatusChange};
     perform;
   };
```

The first line of the above declaration indicates that `SensorDriver` is a basic process type and that processes of this type will run at stations belonging to type `Unit` or its subtypes. The setup method sets `TheSensor` to point to the sensor driven by the process, converts the specified inertia to internal time units (ITUs) and initializes `LastValue` to the current (initial) value of the sensor. The process has only one state; its simple code method is listed below.

```
SensorDriver::perform {
  int NewValue;
  state StatusChange:
    if ((NewValue = TheSensor->getValue ())
                                  == LastValue) {
      TheSensor->wait (NEWITEM, StatusChange);
      sleep;
    }
    signal (LastValue = NewValue);
    Timer->wait (Inertia, StatusChange);
};
```

When the process wakes up (in its only state), it checks whether the current value of the sensor is the same as the previous value. If this happens to be the case, the process issues a wait request to the sensor (to perceive the change in its value) and puts itself to sleep. Otherwise it executes its own `signal` method, passing it the new value as the argument, and sleeps for `Inertia` time units before transiting back to `StatusChange`. This way, all changes in the sensor value will be ignored for `Inertia` ITUs after the last change was reported.

`SensorDriver` reports changes of the sensor value by sending a signal to itself. The signal repository of `SensorDriver` can be consulted by any process that wants to perceive the dampened status of the sensor. The same idea (but acting in the opposite direction) is used in `MotorDriver`.

Now we may have a look at the process actually driving the conveyor segment. Its type is declared as follows:

```
 process SegmentDriver : Overrideable (Segment) {
   MotorDriver *MD; SensorDriver *SDIn, *SDOut;
   Alert *Exception; TIME OutTime, EETime;
   void setup (double);
   states {WtSensor, Input, Output, PcOverride};
   perform;
 };
```

This type descends from `Overrideable`, which is a process subclass intended for processes whose actions can be overridden from outside (e.g., by the operator). `Overrideable` offers some standard tools that can be used for this purpose.

The setup method of `SegmentDriver` takes one `double` argument, which is the bound on the passage time through the segment. It is typical for a SIDE process to store in its attributes local copies of the relevant attributes of the station at which the process is run. We can see this in the following setup method of `SegmentDriver`:

```
 void SegmentDriver::setup (double ttime) {
   Overrideable::setup (S->getOName ());
   MD␣    = S->MD; SDIn␣  = S->SDIn;
   SDOut  = S->SDOut; Exception = S->Exception;
   EETime = (TIME) (Second * ttime);
 };
```

Each "overrideable" process is linked to an override object that can be referenced by the operator to request an override action for the process. Overrides are not basic objects in SIDE, but they are implemented via mailboxes. The setup method of the `Overrideable` portion of `SegmentDriver` is invoked to tag the process's override object with an identifier (the name of the station at which the process is running), so that it can be easily located by the operator.

Now we are ready to look at the code method of `SegmentDriver`.

```
SegmentDriver::perform {
  TIME TIdle;
  state WtSensor:
    onOverride (PcOverride);
    SDIn->wait (SIGNAL, Input);
    SDOut->wait (SIGNAL, Output);
    if (S->Motor->getValue () == ON) {
      if ((TIdle = Time - OutTime) < EETime) {
        Timer->wait (EETime-TIdle, WtSensor);
      } else {
        MD->signal (OFF);
        Exception->notify ("Jam");
      }
    }
  state Input:
    if (TheSignal == ON) {
      MD->signal (ON); S->BoxesInTransit ++;
    }
    proceed WtSensor;
  state Output:
    OutTime = Time;
    if (TheSignal == OFF) {
      if (S->BoxesInTransit) {
        if (--(S->BoxesInTransit) == 0)
                                MD->signal (OFF);
      } else
        Exception->notify ("Unexpected box");
    }
    proceed WtSensor;
  state PcOverride:
    overrideAcknowledge ();
    switch (overrideAction ()) {
      case OVR_MOTOR_CNTRL:
        S->Motor->setValue (overrideValue ());
        onOverride (PcOverride);
        sleep;
      case OVR_SET_COUNT:
        S->BoxesInTransit = overrideValue ();
        OutTime = Time;
        onOverride (PcOverride);
        sleep;
      case OVR_RESUME:
      default:
        S->In->setValue (S->In->getValue ());
        S->Out->setValue (S->Out->getValue ());
        proceed WtSensor;
    }
};
```

The process starts in its first state; this is also the main state where the process awaits the sensor events. The first operation in state `WtSensor` is `onOverride` (defined in the `Overrideable` class), which declares the state to be assumed when an override action is forced by the operator. Then the process issues two wait requests addressed to the signal repositories of the two processes driving the entry and exit sensors. Whenever there is a change in the value of the entry sensor (dampened by the driver process), `SegmentDriver` will transit to state `Input`. Similarly, a change in the value of the exit sensor will force the process to state `Output`.

In state `Input`, the process checks whether the new value of the entry sensor is `ON`, which indicates the presence of a new box. If this is not the case, the signal is ignored and the process returns immediately to its initial state. Otherwise, the motor is switched on (this operation has no effect if the motor is already running), and the number of boxes perceived by the process to be in transit is incremented by one.

In state `Output`, a transition of the exit sensor from `ON` to `OFF` is interpreted as a departure of one box from the segment. The time of this event (the global variable `Time` tells the current time in ITUs) is recorded in `OutTime`. Then the number of boxes in transit is decremented by one, but not below zero. If this number is zero already, the event is inconsistent with the process's perception (there are no boxes in transit, so no boxes should be departing from the segment) and the case is reported to the operator. If the updated number of boxes in transit turns out to be zero, `SegmentDriver` stops the motor. It will be switched back on as soon as a box is perceived by the entry sensor.

The value of `OutTime`, i.e., the time when the last box departed from the segment, is used for jam detection. Each time `SegmentDriver` gets to its initial state, it checks the status of the motor, i.e., the value of the `Motor` actuator. If the motor has been continuously on for `EETime` units, and `OutTime` hasn't changed in the meantime, the process concludes that the last box got stuck somewhere on the belt. In such a case, the motor is stopped and the operator is notified about the problem.

State `PcOverride` is assumed when an explicit override action is requested by the operator. The standard protocol of responding to such an event requires the process to acknowledge the reception of the override request. Otherwise the request would remain pending, and it would keep triggering more override events until acknowledged. The process can learn about the specific action requested by the operator by calling two methods defined in `Overrideable`. Intentionally, `overrideAction` tells the type of operation to be performed (e.g., motor control, resume normal operation) and `overrideValue` specifies an optional parameter of the operation. We can see that the process remains in the overridden state until its normal operation is resumed by an explicit request of the operator. Note that before transiting to its initial state (`WtSensor`), `SegmentDriver` sets the sensors to their current values. This operation leaves the sensor value intact, but it forces a sensor event. This way the values of sensors will be immediately re-examined in the normal mode of operation. Note that these values may have changed while the process was overridden.

5 SUMMARY

We have presented SIDE—a a programming environment for developing distributed reactive programs. The semantics of concurrency in SIDE is simple: non-preemptible threads act like coroutines with implicit control transfer. This approach simplifies synchronization (all problems occur at event boundaries) and, with the right organization of the threads, does not impair the real-time performance of the program.

The interface of a SIDE program with the controlled environment is contained in a single type (mailbox) that can be optionally associated with a TCP/IP port. As the control program only sees virtual sensors and actuators separated from their physical counterparts by a translation layer implemented in SIDE, there is no principal difference between a real system and its simulated artificial model. This way, SIDE is also a rapid prototyping tool. A control program in SIDE can be built together with the development of its underlying physical system.

At `http://sheerness.cs.ualberta.ca/~pawel/SIDE/`, the reader will find a set of pages about SIDE with pointers to three on-line experiments, including two simulated networks of conveyor belts.

REFERENCES

Bertan, B. R. Simulation of MAC layer queuing and priority strategies of CEBus. *IEEE Transactions on Consumer Electronics*, 35:557–563, Aug. 1989.

Dobosiewicz, W., and P. Gburzyński. SMURPH: An object oriented simulator for communication networks and protocols. In *Proceedings of MASCOTS'93, Tools Fair Presentation*, pages 351–352, Jan. 1993.

Dobosiewicz W., and P. Gburzyński. An alternative to FDDI: DPMA and the Pretzel Ring. *IEEE Transactions on Communications*, 42:1076–1083, 1994.

Edwards, S., L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of IEEE*, 85(3):366–390, 1997.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

Gbrzynski, P. *Protocol design for local and metropolitan area networks.* Prentice-Hall, 1996.

IEEE P1451.1/D83 Draft standard for a smart transducer interface for sensors and actuators—Network Capable Application Processor (NCAP) information model, Dec. 1996.

Molle, M. A new binary logarithmic arbitration method for Ethernet. CSRI-298, Computer Systems Research Institute, Toronto, Ontario, Canada, 1994.

Paulin, P., C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded software in real-time signal processing systems: Application and architecture trends. *Proceedings of IEEE*, 85(3):419–435, 1997.

Pree, W. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

AUTHOR BIOGRAPHIES

PAWEL GBURZYNSKI received his MSc and PhD in Computer Science from the University of Warsaw, Poland in 1976 and 1982, respectively. Before coming to Canada in 1984, he had been a research associate, systems programmer, and consultant in the Department of Mathematics, Informatics and Mechanics at the University of Warsaw. Since 1985 he has been with the Department of Computing Science, University of Alberta where he is a Professor. His research interests are in network protocols, operating systems, simulation, and performance evaluation.

JACEK MAITAN received his MSc in Automation from the Silesian Technical University in 1976, PhD in Electrical Engineering from the University of Arizona in 1984, and MBA from Queens College in 1996. He held engineering and management positions at MCC, RCA, CompuServe, and Lockheed, providing technical leadership for a number of contracts with government agencies, including NASA, ARPA, and ONR. He has extensively consulted for the industry in the areas of communication and distributed systems. Jacek Maitan is President of Tools For Sensors, Inc.