# Simulation and Control of Reactive Systems

Pawel Gburzynski* Jacek Maitan†

*Draft*

**Abstract**

We introduce SIDE[1]—a software package for developing control programs for reactive systems. The package offers an object-oriented programming language (based on C++) for describing configurations of reactive systems and specifying event-driven threads of their control programs. 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, which makes it possible to develop a control program together with the physical system to be controlled. Notably, the control program itself need not be aware that some parts of its environment are not real. This way, the same software can be seamlessly used in simulation, emulation, and real-time execution of the "real" system. SIDE applications can be naturally distributed and interconnected via the Internet. In particular, control programs in SIDE can be monitored and operated from remote locations via Java applets invoked from web pages.

## 1 Introduction

Traditionally, industrial process control networks, often implemented as lines connected directly to the backplane of the host, have been highly specialized, with their hardware and software (protocols) dedicated to their specific purpose. This approach was a consequence of the highly local character of these networks: there was no need for interoperability of different networks or for remote and friendly access to the processes controlled by them. The local nature of the control schemes also resulted in a master-slave system architecture, that did not scale up to match the growing size of the controlled system. With the advent

*Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, tel. (403) 492-2347, fax (403) 492-1071, url: http://www.cs.ualberta.ca/~pawel, email: pawel@cs.ualberta.ca.

†Tools for Sensors, 513 Marshall Avenue, Carmichael, CA 95608, tel. (916) 944-2714, email: jacek@earthling.net.

[1]SIDE stands for Sensors In a Distributed Environment.

of low-cost microprocessors and network componenents, an increasing number of real-time embedded systems used in automated manufacturing and industrial process control depend on the use of sensors and actuators interconnected by networks.

With the rapid growth of the Internet, people start to do remotely and cost-effectively many things that traditionally have required their physical presence at the processing site, e.g., shopping, banking transactions, 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" or proprietary 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 [5]. When they are fulfilled, we will see a drastic increase in applications of complex control networks. In particular, they will be used at home, not only as toys, but as sophisticated processing systems with considerable autonomy and intelligence.

The software package presented in this paper offers a number of tools aimed at fulfilling the postulates mentioned above. 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)[2] 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 in fact behaves as an event-driven, discrete-time simulator. The difference between the two modes is exclusively in the kernel; the control program itself is indifferent to the mode. Therefore, depending which way we look at it, SIDE can be viewed as a simulator or as an execution platform for control programs driving physical systems. The simulated components of the execution environment for a control program are most naturally programmed in SIDE.

[2]DSD stands for Dynamic Status Display.

# 2 The structure of SIDE

## 2.1 Control and simulation

The network-independent portion of SIDE consists of the compiler (`SMPP`) and the kernel library. As shown in Figure 1, a program in SIDE is first preprocessed by `SMPP` and turned into C++ code. This code is then compiled and linked with a library of modules, which produces a stand-alone executable program. If we are interested exclusively in simulation, the resultant program can be fed with some input data and run "as is" to produce performance results. SIDE is equipped with the standard features of simulators, as random number generators and tools for collecting performance data. It is also possible to monitor the execution of the simulation experiment on-line (possibly remotely) via DSD. Another tool that is useful at this stage is SERDEL—a program for automated distribution and supervision of multiple simulation experiments run on a local network of computers.

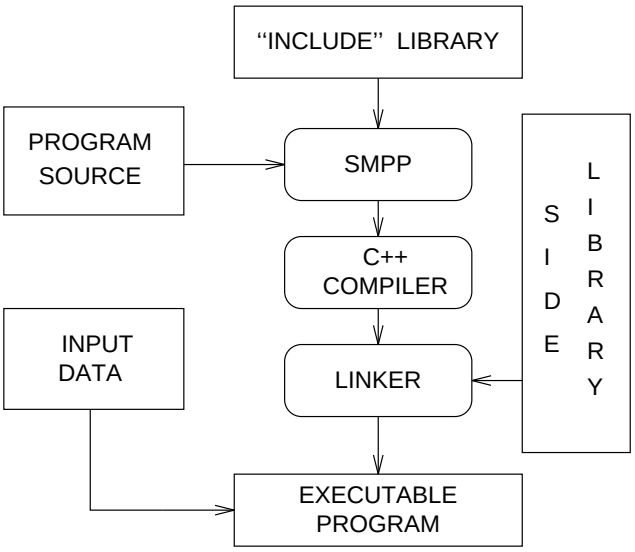


Figure 1: Program compilation in SIDE.

For the purpose of simulation, the source program in SIDE is logically divided into three parts (see Figure 2). 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 the outside and their timing.

The terms "protocol" and "traffic" are usually applied to communication networks. 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 the outside, and it is natural to represent such objects as structured *packets*.

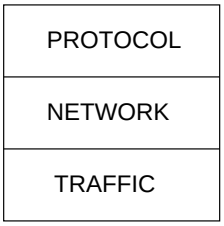


Figure 2: Components of a SIDE program.

For the purpose of developing control programs in SIDE, we adopt a slightly more elaborate view of the source program (see Figure 3). The protocol (i.e., the collection of processes) 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).

With the above view, one can separate the software components that belong to the control system from those belonging to the simulator. Thus, the "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

For the purpose of controlling by SIDE, a reactive system is defined as a collection of sensors and actuators (see Figure 4). These two objects are very similar; in fact they are both described by the same data structure with the following layout:

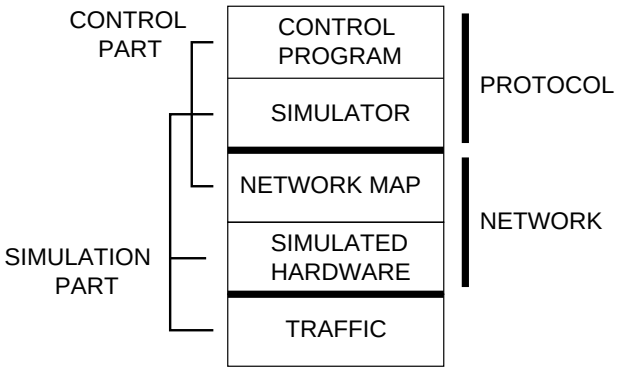


Figure 3: A control system in SIDE.

```
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

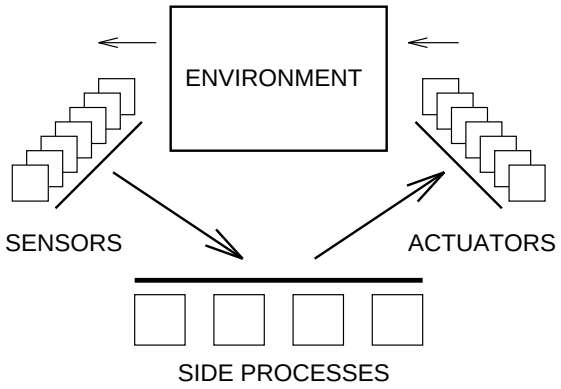


Figure 4: A reactive system controlled by SIDE.

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 [3, 8].

A control program in SIDE can be implemented as a single (mulithreaded) program, or as a set of embedded programs [2, 7] 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. One can enhance the reliability of a control program by providing several copies/versions of a single module. Those multiple copies can be programmed differently, e.g., by different people, and they can be run on different machines at different geographical locations, controlling the same physical system.

### 2.3 Observers

Besides conventional tools for program debugging, such as local assertions and tracing functions, SIDE offers means for verifying compound dynamic conditions that can be viewed as an alternative specification of the control program. These tools, the so-called observers, are thread-like objects that can be used for expressing global assertions involving the combined behavior of more than one regular thread. As regular assertions verify some Boolean properties of a program, observers verify state transitions in a distributed system and make sure that these transitions conform to a set of specifications. In this sense, observers are conformance testing tools [1, 4, 6].

## 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. Similar to `Kernel`, the `Root` process runs in the context of the `SYSTEM` station.

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)[3] 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. Although most of them are also created once at the beginning, it is not uncommon to create (and destroy) processes dynamically for various intermediate tasks.

All objects that exhibit dynamic behavior are dubbed *activity interpreters* (AI for short). 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.

[3] Links and ports are mostly used in simulators—to model the passage of packets through some "channels."

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). But stations do not exhibit any activities of their own; they do not trigger any events by themselves, and they are not activity interpreters.

Some object types used in SIDE programs are basic, in the sense that they are offered directly by the kernel. Some other types are described in SIDE and they represent a problem-oriented library useful in implementing control programs for networks of sensors and actuators. Thus, type `Mailbox` is basic but types `Sensor` and `Actuator` descend from `Mailbox` and are implemented in SIDE.

### 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' *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 process gets into this state immediately after it is created.

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:
      ...
   ...
   state sk:
      ...
};
```

Two standard pointers (whose declarations are implicit and invisible) are available to the code method. They are: `F` (of type `fptype`) pointing to the process' parent, and `S` (of type `stype`) pointing to the station owning the process. This way, besides having natural access to its private objects, a process can reference public attributes of its parent and its owner.

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.[4] 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 (which is not uncommon in SIDE), 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 (which is always deterministic).

[4] For uniformity, the first event starting a process is assumed to be triggered by the process itself; thus, in fact, there are no exceptions.

Once a process has been awakened by one of the awaited events, 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. Conceptually, they can be viewed as fast interrupt service routines in a real-time operating system. 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).

SIDE models time with practically arbitrary accuracy (at least in principle). 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. Therefore, with a negligible overhead, one can get down with the ITU to the Planck time, which (according to one of the best verified theories of physics) is the ultimate granularity of time from the viewpoint of controlling physical systems.

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. One special state that can be used in this context is `DEATH`. Thus,

with the following sequence:

```
pr = create MyChild;
pr->wait (DEATH, Done);
sleep;
```

the issuing process creates a child process and waits for its termination.

Another IPC tool is signal passing. Each process has a *signal repository* that can be used to receive signals (simple messages). There is a special class of *priority signals* that provide for a non-preemptible control transfer among processes.

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 completely 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 Case study: a conveyor system

For illustration, let us consider a system of conveyor belts built of the following types of units:

- segment, i.e., a piece of conveyor belt with one entry and one exit,
- merger, i.e., a unit with two entries and one exit,
- diverter, i.e., a unit with one entry and two exits.

In the subsequent sections, we will present some fragments of a SIDE program driving such a system.

### 4.1 Stations

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 (we will call them 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.[5]

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 actuator
              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)

[5]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.

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 &motor, double MotorInertia) {
  Exception = create Alert (getOName ());
  Motor     = create Actuator (motor);
  MD        = create MotorDriver (Motor, MotorInertia);
};

void Segment::setup (NetAddress &motor,
                     double MotorInertia,
                     NetAddress &entry,
                     double ESensorInertia,
                     NetAddress &exit,
                     double XSensorInertia,
                     double TransitTimeBound) {
  Unit::setup (motor, MotorInertia);
  BoxesInTransit = 0;
  In    = create Sensor   (entry);
  Out   = create Sensor   (exit);
  SDIn  = create SensorDriver (In, ESensorInertia);
  SDOut = create SensorDriver (Out, XSensorInertia);
  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 (i.e., it descends directly from the built-in type `Process`) and that processes of this type will run at stations belonging to type `Unit` (this includes subtypes of `Unit`, in particular `Segment`). 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, with the new value deposited in the process's *signal repository*. 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 LastOutTime, EndToEndTime;
  void setup (double);
  states {WaitSensor, Input, Output, ProcessOverride};
  perform;
};
```

This type descends from `Overrideable`, which is a process subclass (defined in SIDE) intended for processes whose actions can be overridden from the outside (e.g., by the operator). `Overrideable` offers some standard tools that can be used for this purpose. Processes of type `SegmentDriver` run at `Segment` stations.

The setup method of `SegmentDriver` (invoked when the process is created in the setup method of `Segment`) takes one `double` argument, which is the bound on the passage time through the segment. The process has four states and a number of attributes. As is typical for a SIDE process, most of these attributes are used to hold local copies of the relevant attributes of the station at which the process is run. We can see this from the following setup method of `SegmentDriver`:

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

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 IdleTime;
  state WaitSensor:
    onOverride (ProcessOverride);
    SDIn->wait (SIGNAL, Input);
    SDOut->wait (SIGNAL, Output);
    if (S->Motor->getValue () == ON) {
      if ((IdleTime = Time - LastOutTime) < EndToEndTime) {
        Timer->wait (EndToEndTime-IdleTime, WaitSensor);
      } else {
```

```
        MD->signal (OFF);
        Exception->notify ("Jam involving %1d boxes", S->BoxesInTransit);
      }
    }
  state Input:
    if (TheSignal == ON) {
      MD->signal (ON);
      S->BoxesInTransit ++;
    }
    proceed WaitSensor;
  state Output:
    LastOutTime = Time;
    if (TheSignal == OFF) {
      if (S->BoxesInTransit) {
        if (--(S->BoxesInTransit) == 0) MD->signal (OFF);
      } else
        Exception->notify ("Unexpected box");
    }
    proceed WaitSensor;
  state ProcessOverride:
    overrideAcknowledge ();
    switch (overrideAction ()) {
      case OVR_MOTOR_CNTRL:
        S->Motor->setValue (overrideValue ());
        onOverride (ProcessOverride);
        sleep;
      case OVR_SET_COUNT:
        S->BoxesInTransit = overrideValue ();
        LastOutTime = Time;
        onOverride (ProcessOverride);
        sleep;
      case OVR_RESUME:
      default:
        S->In->setValue (S->In->getValue ());
        S->Out->setValue (S->Out->getValue ());
        proceed WaitSensor;
    }
};
```

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 `WaitSensor` 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 `LastOutTime`. 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 `LastOutTime`, 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.[6] If the motor has been continuously on for `EndToEndTime` units, and `LastOutTime` 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 `ProcessOverride` 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.[7] Then 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 (`WaitSensor`), `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.

### 4.3 Mapping virtual sensors/actuators to their physical counterparts

The `SegmentDriver` process (as well as the processes driving other components of our conveyor system) assume some simple and uniform functionality of a sensor/actuator. These virtual objects are mapped to their physical (or simulated) counterparts in a way

[6] This is a natural example of a situation in which a control program may be interested in reading the value of an actuator, as opposed to modifying it.

[7] Otherwise the request would remain pending, and it would keep triggerring more override events until acknowledged.

that is transparent to the control program. For illustration, we will explain here how this mapping is set up for a simple SDS sensor.

A real network of sensors/actuators is made visible to the SIDE kernel through a daemon that translates between the network's internal protocol and TCP/IP. On the kernel's side, the TCP/IP interface is visible as a mailbox bound to a TCP/IP port. The following process carries out the mapping of a binary SDS sensor:

```
process SDSToVirtual {
  Sensor *Sn;
  Mailbox *Sm;
  void setup (Sensor *s, Mailbox *m) {
    Sn = s;
    Sm = m;
    Sm->connect (INTERNET+CLIENT, SERVNAME, PORT, SBUFSIZE);
    requestChangeReports (Sm, Sn);
  };
  states {WaitSDSInit, WaitSDSStatus};
  perform {
    char msg [STATMESSIZE];
    state WaitSDSInit:
      if (Sm->read (msg, CONFMESSIZE) != OK) {
        Sm->wait (CONFMESSIZE, WaitSDSInit);
        sleep;
      }
    transient WaitSDSStatus:
      if (Sm->read (msg, STATMESSIZE) == OK) {
        Sn->put (msg [STATMESSTAT] == SDS_ON ? ON : OFF);
        proceed WaitSDSStatus;
      } else
        Sm->wait (STATMESSIZE, WaitSDSStatus);
  };
};
```

The setup method of the process binds the generic mailbox pointed to by its second argument to a TCP/IP port on the server running the SDS interface daemon.[8] Then it calls `requestChangeReports`, which is in fact a macro, to send to the daemon a message requesting status updates of the indicated physical sensor (the one into which the virtual sensor `Sn` has been formally mapped). This is how the macro is defined:

```
#define requestChangeReports(m,s)\
  do { (m)->put (CRT_AIS);\
       (m)->put (0);\
       (m)->put (2);\
```

[8]For secure communication, it is possible to use SSL (Secure Socket Layer) with RSA encryption for this connection.

```
(m)->put ((s)->Reference.Net);\
(m)->put ((s)->Reference.NetId); } while (0)
```

It simply sends the prescribed sequence of bytes (the command byte, followed by the length of the operands, followed in turn by the operands) to the daemon using the standard `put` operation of the mailbox. The operands specify the physical address of the real sensor in the SDS network. Having received this request, the daemon will send back a short confirmation message, and then it will start communicating all changes in the sensor value to the mailbox.

The first state of the above process is solely used to absorb (and for simplicity ignore) the confirmation message. The semantics of `read` issued on a mailbox connected to the network is to receive into the buffer specified as the first argument the number of bytes indicated by the second. If that many bytes are not immediately available, the operation returns `ERROR` (which is different from `OK`). Then the process may issue a wait request to the mailbox, with the first argument specifying the requested number of bytes. As soon as that many bytes become available for an immediate read, the awaited event will be triggered.

Having received the confirmation, the process transits to its second state where it reads subsequent update messages arriving from the daemon. One byte of such a message (at least for a simple binary sensor) contains the on/off value, which is converted and passed to the virtual sensor.

A similar process can be easily set up to carry out the mapping in the opposite direction, i.e., for an actuator. In fact, it is even simpler because there is no need to send the initial request message in that case.

### 4.4 Observers

An observer looks like a special process that, instead of the events triggered by activity interpreters, responds to state transitions in regular processes. This way, it can easily verify compound dynamic conditions expressed as patterns of state transitions in the control program. This capability is achieved with remarkably simple tools: there are just two operations specific to observers: `inspect` declaring an expected state transition, and `timeout` used to detect timeouts (i.e., the lack of an expected transition within a given time interval).

For example, consider the following observer that asserts the correctness of the `DiverterDriver` process driving a diverter unit in our conveyor system:

```
observer DiverterVerifier {
  Alert *Exception;
  states {WaitBox, NewBox, Error};
  void setup () {
    Exception = create Alert ("Diverter timeout alert");
  };
```

```
  perform {
    state WaitBox:
      inspect (ANY, DiverterDriver, DivertDecision, NewBox);
    state NewBox:
      inspect (TheStation, DiverterDriver, EndDivert, WaitBox);
      timeout (DivertTimeout, Error);
    state Error:
      Exception->notify ("Diverter timeout");
      proceed WaitBox;
  };
};
```

Similar to a regular process, the above observer has its list of states and a code method (`perform`) looking like a description of a finite state machine. The inspect request issued in the first state indicates that the observer wants to be awakened (in state `NewBox`) as soon as the `DiverterDriver` process at `ANY` station (unit) is restarted in state `DivertDecision`. We can guess that this is the state in which the process receives a box, and it has to decide whether the box should be diverted or not. Then, in state `NewBox`, the observer expects the same `DiverterDriver` process (`TheStation` points to the station owning the process that caused the observer's transition to `NewBox`) to transit to state `EndDivert` (apparently this is the state where the diverter is done with the box). However, if the transition doesn't occur within `DivertTimeout` time units, the observer will transit to state `Error`, where it will notify the operator about the problem.

Observers are powerful enough to provide a mechanism for building alternative specifications of the intended behavior of the control program. They can be programmed independently (by different people) and executed together with the imperative part of the program, to make sure that the program conforms to its specification. Normally, observers are passive (i.e., they don't interfere with the operation of the program), but if needed, they can perform operations affecting the controlled system.

# 5 Summary

We have presented SIDE—a a programming environment for developing reactive distributed programs implemented as collections of threads responding to events. The semantics of the concurrency in SIDE is very simple: threads are not preemptible and they resemble coroutines with implicit control transfer. This approach simplifies synchronization (all synchronization problems occur at event boundaries) and, with the right organization of the threads, does not impair the real-time performance of the control 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.

The networking interface of SIDE avoids the problem of blocking I/O (that may cause problems in distributed reactive systems [9]) by implementing it through mailboxes that trigger events when the I/O becomes possible. A process willing to read something from a mailbox with no data pending has three options: to suspend itself awaiting data arrival (the conventional approach), to ignore the operation and poll the mailbox later (non-blocking I/O), or to spawn a separate process to read the data when it becomes available and notify its parent about that fact via a signal.

The reliability of a system controlled by SIDE can be enhanced by providing multiple versions of the same program controlling the same equipment from different sites. Note for example that the `SegmentDriver` process presented in Section 4.2 will operate correctly if another `SegmentDriver` is controlling the same segment at the same time. This property is natural for many reactive systems and it can be explored in a properly designed control program.

## References


[1] J. M. Ayache, P. Azéma, and M. Diaz. Observer: a concept for on-line detection of control errors in concurrent systems. In *Proceedings of the 9th Symposium on Fault-Tolerant Computing*, pages 1–8, Madison, WI, June 1979.

[2] S. Edwards, 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.

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

[4] R. Groz. Unrestricted verification of protocol properties in a simulation using an observer approach. In *Proceedings of the IFIP WG 6.1 6th Workshop on Protocol Specification, Testing, and Verification*, pages 255–266. North-Holland, June 1986.

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

[6] R. Molva, M. Diaz, and J. Ayache. Observer: a run-time checking tool for local area networks. In *Proceedings of the IFIP WG 6.1 6th Workshop on Protocol Specification, Testing, and Verification*, pages 495–506. North-Holland, 1987.

[7] P. Paulin, 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.

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

[9] D. Schmidt and P. Stephenson. Experiences using design patterns to evolve systems software across diverse OS platforms. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, Aug. 1995.