Specifying control programs for reactive systems

(extended abstract)

Pawel Gburzynski
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
url: http://www.cs.ualberta.ca/~pawel

Jacek Maitan
Tools For Sensors, Inc.
3513 Marshall Avenue
Carmichael, CA 95608
USA
e-mail: jacek@concourse.net

April 23, 1998

1 Introduction

By a *reactive system*, we understand a dynamic physical object whose logical behavior can be described as a sequence of discrete chronological changes of some finite set of parameters. Examples of such systems include industrial manufacturing and control processes, robots, and immobots [4].

The software package presented in this paper, SIDE,[1] provides a platform for specifying and executing distributed control programs driving reactive systems. Interestingly, the package is a direct descendant of a network simulator [1, 2], to the point of retaining all the simulation features of its predecessor. Specifically, SIDE offers:

- a programming language for describing configurations of distributed reactive systems and specifying distributed programs organized into fine-grained event-driven threads
- tools providing a reactive interface between a SIDE program and the outside world
- a kernel for executing programs expressed in the language of SIDE
- a Java interface for monitoring the execution of SIDE programs from the Internet
- *daemons* interfacing commercial networks of sensors and actuators to the SIDE kernel

The SIDE kernel has two modes of operation. In the real mode, the events perceived and triggered by threads (SIDE processes) occur in actual time. Usually, some of them are triggered by real events, and some of them affect the behavior of some physical objects. In

[1] SIDE stands for "Sensors In a Distributed Environment." The package is distributed by ToolsForSensors, Inc.

the virtual mode (which is only possible if the entire environment of the control program is simulated), the time is virtual and the control program behaves as an event-driven, discrete-time simulator. In the real mode, simulated components of the controlled system can coexist with is real parts. This may be useful for fast prototyping, parallel engineering, or fault tolerance. For example, a failed real sensor can be replaced by its virtual counterpart driven from a model without modifying the control program [5].

2 Case study: water tanks

For illustration, let us consider a system consisting of an interconnected row of tanks shown in Figure 1.

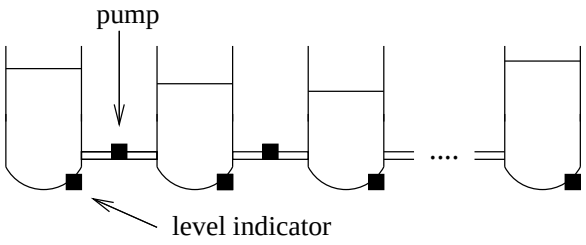

Figure 1: A row of tanks.

Each tank is equipped with a sensor indicating the water level. Two neighboring tanks are connected with a pipe and a bidirectional pump. We assume that the tanks are filled (and possibly emptied) from outside. The role of our control program is to maintain the same level of water in all tanks.

2.1 Stations

The primary action agents in our simple setup are pumps; such agents are most naturally represented in SIDE as *stations*. A pump station can be declared as follows:

```
station Pump {
  Sensor *LeftLevelIndicator, *RightLevelIndicator;
  Actuator *Motor;
  void setup (NetAddress&, NetAddress&, NetAddress&);
};
```

When a `Pump` is created, its setup method (the constructor) receives three network addresses: two of them describe two sensors (indicating the water level in the tanks connected by the pump), and one identifies the pump's motor actuator. The setup method is responsible for creating those objects, in the following way:

```
void Pump::setup (NetAddress &ls, NetAddress &rs, NetAddress &mt) {
  LeftLevelIndicator = create Sensor (ls);
  RightLevelIndicator = create Sensor (rs);
  Motor = create Actuator (mt);
};
```

The arguments of the `create` operation are passed to the setup method of the created object. The (virtual) sensors and the actuator will be mapped into their physical counterparts by the *network map* layer of the program

2.2 Sensors and actuators

Sensors and actuators are implemented as *mailboxes* which are basic reactive IPC tools in SIDE. A mailbox can be optionally bound to a network port or a device.

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

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 setting the `Value` attribute of the actuator we force it to carry out a specific physical operation.

The setup argument specifies the object's coordinates in the controlled system. 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 control program.

2.3 Processes

Each pump station is controlled by a single process whose responsibility is to monitor the water level in the neighboring tanks and start the pump whenever there is a difference. This process is declared as follows:

```
process PumpDriver (Pump) {
  Sensor *LLI, *RLI;
  Actuator *M;
  void setup () {
```

```
    LLI = S->LeftLevelIndicator;
    RLI = S->RightLevelIndicator;
    M = S->Motor;
  };
  states {WaitStatusChange, StatusChange};
  perform;
};
```

The first line of the above declaration indicates that `PumpDriver` will run at a station belonging to type `Pump`. The setup method creates local copies of the pointers to the attributes of the owning station—for convenient access from the process. The process has two states and it runs the following code method:

```
PumpDriver::perform {
  state WaitStatusChange:
    LLI->wait (NEWITEM, StatusChange);
    RLI->wait (NEWITEM, StatusChange);
  state StatusChange:
    if (LLI->getValue () < RLI->getValue ())
      M->setValue (PUMP_LEFT);
    else if (LLI->getValue () > RLI->getValue ())
      M->setValue (PUMP_RIGHT);
    else
      M->setValue (OFF);
    proceed WaitStatusChange;
};
```

Having started in state `WaitStatusChange`, `PumpDriver` issues two wait requests, one to each of the two sensor mailboxes. The process wants to get to its second state, `StatusChange`, as soon as the value of any of the two sensors changes. This corresponds to a change in the water level in any of the two tanks connected by the pipe (and pump) managed by the process. When a change is detected, `PumpDriver` compares the indications of the two sensors. If there is less water in the left tank, the pump motor is started in the left direction. Otherwise, if the right tank contains less water, the pump is started in the opposite direction. Finally, if the two levels are the same, the pump is switched off. At the end of this action, the process unconditionally transits to its initial state.

Note that storing the same value in the motor actuator several times is harmless, although redundant. The code method of `PumpDriver` could be easily "optimized" by remembering the previous status of the actuator and setting it to the new value only if that value has in fact changed. The gain would be negligible, however. Besides, the persistent behavior of the process may be advantageous from the viewpoint of reliability.

Another problem with the simple code method presented above is the lack of any tolerance for the water level. If the sensor is very accurate, the code method may behave in a "jumpy" fashion, obsessively equalizing the levels that for all practical purposes appear equal. It would not be a big problem to add a tolerance to the two comparisons at state `StatusChange`; however, a better place to dampen the indications of the level sensors may be in the network map layer—as shown below.

2.4 Mapping virtual sensors/actuators to their physical counterparts

The virtual sensors and actuators are mapped to their physical (or simulated) counterparts in a way that is transparent to the control program. We will sketch here how this mapping is set up for a simple SDS sensor.[2]

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 level sensor:

```
process SDSToVirtual {
  Sensor *Sn;
  Mailbox *Sm;
  int Granularity, NewStat;
  void setup (Sensor *s, Mailbox *m, int g) {
    Sn = s;
    Sm = m;
    Granularity = g;
    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) {
        NewStat = (msg [STATMESSTAT] << 8) | msg [STATMESSTAT+1];
        Newstat = (Granularity * Newstat) / MAXSTAT;
        if (Sn->getValue () != NewStat) Sn->setValue (NewStat);
        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.[3] Then it calls `requestChangeReports` 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). The function simply sends a prescribed sequence of bytes to the daemon. Having

[2]Information on SDS sensing equipment manufactured by Honeywell can be found at `http://www.honeywell.sensing.com/sds/sdspec.html`.

[3]It is possible to use SSL (Secure Socket Layer) for this connection.

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 setup method also sets the "granularity" parameter specifying the number of discrete levels of the virtual sensor. The indications of the physical sensor will be transformed into this many levels—to provide a tolerance margin and avoid a jumpy behavior of the pump.

The first state of the above process is solely used to absorb (and for simplicity ignore) the confirmation message sent by the daemon in response to the update request. 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 process may issue a wait request to the mailbox, with the first argument specifying the requested number of bytes. As soon as the 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. Two consecutive bytes of such a message starting at position `STATMESSTAT` contain the value of the sensor. This value is transformed into one of `Granularity` levels and inserted into the virtual counterpart of the 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. Also, the motor actuator responds to three discrete values that need not be dampened.

2.5 Other features

The sensor/actuator library of SIDE offers a collection of tools for interfacing control processes with the operator. These tools come as types `Alert` and `Override`.

An `Alert` is a mailbox that can be used for depositing messages addressed to the operator. Such a message implicitly identifies the object responsible for triggering it. An `Override` is another mailbox whose purpose is to manually override the action of a SIDE process. A process declared as "overrideable" is automatically equipped with an `Override` mailbox. A message received on such mailbox is interpreted as an override request that unconditionally forces the process into a specific state.

The contents of mailboxes interfacing the SIDE program to the outside world can be "journaled," i.e., mirrored to files. It is possible to rerun the program binding selected mailboxes to journal files. This way, it is possible to re-execute fragments of the control program, repeating a sequence of operations from a past epoch.

3 Summary

We have presented SIDE—a a programming environment for developing reactive distributed programs implemented as collections of threads responding to events.

The interface of a SIDE program with the controlled environment is contained in a single type (mailbox) that can be optionally associated with a networking port or a

device. 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 [3]) 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 `PumpDriver` process presented in Section 2.3 will operate correctly if another `PumpDriver` is controlling the same pipe at the same time. This property is natural for many reactive systems and it can be explored in a properly designed control program.

At `http://sheerness.cs.ualberta.ca/~pawel/SIDE/product.html`, the reader will find a set of pages about SIDE with pointers to three on-line experiments including the (simulated) water tanks discussed in this paper. One of those experiments involves a simple (but real) SDS network controlled by a SIDE program.

References

[1] W. Dobosiewicz 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.

[2] P. Gburzyński. *Protocol design for local and metropolitan area networks.* Prentice-Hall, 1996.

[3] 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.

[4] B. Williams and P. Nayak. Immobile robots AI in the New Millennium. *AI Magazine*, 17(3):16–35, 1996.

[5] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, Portland, Oregon, Aug. 1996.