VIRTUAL PROTOTYPING OF REACTIVE SYSTEMS IN SIDE

Pawel Gburzynski
Department of Comp. Science
University of Alberta
Edmonton, AB, CANADA T6G 2H1
pawel@cs.ualberta.ca

Jacek Maitan
Tools For Sensors, Inc.
3513 Marshall Avenue
Carmichael, CA 95608
jacek@concourse.net

Larry Hillyer
TechnoTrim
3749 Petersen Road
Stockton, CA 95215
Larry.D.Hillyer@jci.com

KEYWORDS

Reactive systems, control programs, specification, simulation, parallel design, rapid prototyping.

ABSTRACT

We introduce a software package, called SIDE,[1] for developing and executing control programs driving distributed 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 system can be virtual, which makes it possible to develop the control program together with the physical system to be controlled. 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.

Introduction

Many real-time systems used in automated manufacturing or industrial process control depend on the use of sensors and actuators interconnected by networks. Traditionally, such networks have been highly specialized, with their hardware and software 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. This old-fashioned attitude is changing today because of the ubiquitous Internet. More and more people get used to the idea of doing remotely many things that traditionally required their physical presence at the processing site.

[1] Sensors In a Distributed Environment.

Our package 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 a SIDE program from the Internet.
- A TCP/IP daemon interfacing physical networks of sensors and actuators to the Internet.

AN OVERVIEW OF SIDE

Program structure

A program in SIDE is first processed by the compiler, `SMPP`, and turned into C++ code (Figure 1). This code is then compiled and linked with a library of modules. 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 online (possibly remotely) via a special applet.

For the purpose of simulation, the source program in SIDE is logically divided into three parts:

- **System description** defining the configuration (hardware) of the modeled system.
- **Protocol program**, i.e., the executable specification of the system's behavior.

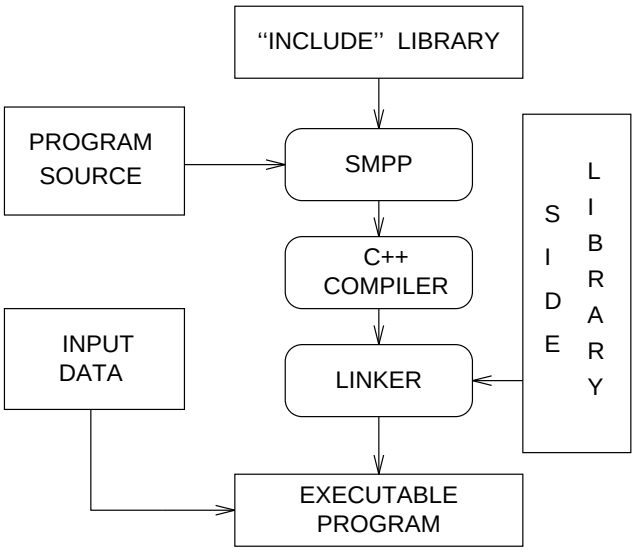

Figure 1: Program compilation in SIDE.

- **Traffic**, i.e., the specification of the offered load (events arriving from the 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., Bertan 1989; Gburzynski 1996; Molle 1994). For the purpose of developing control programs in SIDE, we change slightly the view of the source program. The *protocol program* consists now of two parts: the *control program* proper and the simulator for the virtual components of the driven system. The traffic specification only applies to the simulated part of the environment, which will tend to vanish in the complete version of the system.

Interface to the world

A reactive system is defined as a collection of sensors and actuators. Both types of objects are described by the following generic data structure:

```
mailbox Sensor {
  private:
    int Value;
  public:
    void setValue (int);
    int getValue ();
};
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 setting the `Value` attribute of the actuator we force it to carry out a specific physical operation.

The mapping of a sensor/actuator to its physical counterpart is done in a way that doesn't change the above general view. In particular, 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 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 "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; Pree 1995).

Other features

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 (Ayache et al. 1979; Groz 1986; Molva et al. 1986), are thread-like objects that can be used for expressing global assertions involving the combined behavior of more than one regular thread.

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

Program components

The basic unit of execution is called a *process* and it looks like a specification of a finite state machine (FSM). A process always runs in the context of some *station*, representing a logical component of the controlled/modeled system.

Stations give the logical view of the hardware on which the control program or simulator is run. All objects that exhibit dynamic behavior are dubbed *activity interpreters* (AI). Such ob-

jects 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.

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.

Processes

A process consists of a 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*.

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 `states` declaration assigns symbolic names to the states of the FSM represented by the process. 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 performs some operations and suspends itself. Among these operations are indications of future events that the process wants to perceive. A typical code method (declared with `perform`) has the following structure:

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

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: the process will be awakened by the earliest of the awaited events in the state indicated by the second argument of the corresponding wait request. The priority argument indicates the precedence of events that occur simultaneously. 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 forcing such events to be processed in the order of their perception by the SIDE kernel.

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. 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.

Time in SIDE

The SIDE kernel has two modes of operation. In the real mode, the events perceived and triggered by processes occur in actual time. Usually, some of them are triggered by real *events* (e.g., perceived by sensors), and some of them affect the behavior of some physical objects (e.g., via actuators). In the virtual mode (which is only possible—although not required—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. The difference between the two modes is exclusively in the kernel, not in the control program. 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.

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 process (or even itself) to get into a specific state (or become terminated).

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 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 an `UPDATE` event. This event will be perceived by the process (or processes) monitoring the changes of the sensor value.

EXAMPLE

Below we list the implementation of a programmable logic controller (PLC) as a SIDE actuator. This implementation is generic and assumes that every actual PLC type will provide a method (declared as virtual in the following `PLC` class) to turn the actuator value into a message to be sent to the PLC device.

A logical PLC is described by two objects: a SIDE `Actuator` and a process passing the changes in the actuator's value to the physical PLC. The actuator part is implemented as follows:

```
mailbox PLC : Actuator {
  char Msg [MSGLEN];  int MsgLen;
  void setup (const char *dev) {
    connect (DEVICE+WRITE+RAW, dev, MSGLEN);
    create PLCDriver (this);
  };
  int update () { return write (Msg, MsgLen); };
  virtual mkMessage (int) { };
};
```

The setup method of the above class is called automatically when a `PLC` object is created. It connects the actuator's mailbox to the device specified as its argument. Note that the program must be able to write to the device. The `RAW` flag indicates that the system should not try to interpret the written data (e.g., as text lines) before submitting them to the physical PLC. The method also creates the following process:

```
process PLCDriver {
  int New, Old;  PLC *P;
  void setup (PLC *p) { P = p; Old = NONE; };
```

```
  states {WaitCommand, SendMessage};
  perform {
    state WaitCommand:
      if ((New = P->getValue ()) != Old) {
        P->mkMessage (Old = New);
        proceed SendMessage;
      }
      P->wait (UPDATE, WaitCommand);
    state SendMessage:
      if (P->send () == OK) proceed WaitCommand;
      P->wait (OUTPUT, SendMessage);
  };
};
```

which perceives the changes in the actuator's `value` and passes them to the PLC.

It is assumed that `mkMessage` generates a sequence of bytes to be written to the controller's device. This sequence is stored in the `Msg` array, and `MsgLen` is set to its actual length. Every specific PLC type, which can be declared as a descendant of type `PLC`, must specify an actual implementation of `mkMessage`.

The `PLCDriver` process starts in its first state (`WaitCommand`) and it gets there whenever it suspects that the value of the logical actuator has changed. Such a change triggers an `UPDATE` event on the actuator's mailbox.

The `send` method of the controller issues a write operation which, for a bound mailbox, is transformed into a non-blocking write to the underlying device. The written bytes are inserted into a buffer which will be emptied asynchronously with other activities of the program. It is possible (although not very likely) that the operation will fail, which means that the previous contents of the buffer haven't been emptied. In such a case, the process will wait for the `UPDATE` event triggered when some buffer space becomes available, and then it will retry the operation.

SUMMARY

SIDE is 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 TCP/IP 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 virtual model. This way, SIDE is 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 (Schmidt and Stephenson 1995) 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.

With the right design of the control program, the reliability of the system controlled by SIDE can be enhanced by providing multiple versions of the same program controlling the same equipment from different sites.

REFERENCES

Ayache, J.M. and 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.

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

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

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

Groz, R. "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.

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

Molva, R.; 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.

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

Schmidt, D. 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.