# System-Level Modeling and Evaluation of Network Protocols

by

**Shang-Pin Chang**
Master of Engineering in Electrical Engineering and Computer Sciences
University of California at Berkeley
Professor David G. Messerschmitt

## Abstract

Since the first large-scale computer network was built in the early 1960s, the protocol design problem has become a more important issue to efficiently coordinate distributed system nodes. Recently, in response to the fast growing demand for connecting various devices with current network infrastructures, many intricate protocols have been designed to support communications across such heterogeneity. However, today very few tools that we can identify allow such a system-level simulation, including both protocols and models of system entities. Since simulation is the major stage in the development cycle of a complex hardware and software distributed system, a tool facilitates modeling and simulating protocols in a system context is substantially valuable.

In this report, we propose a hybrid model of computation including CSP, FSM, and DE for specifying protocols as well as to enable mixing them with other subsystem models. Based on this proposal, a software tool, SiP (SPIN in Ptolemy), has been implemented by integrating a protocol simulation tool, SPIN, into a system-level design environment, Ptolemy. We demonstrate the expressive power of SiP by using it to specify several fundamental elements of network protocols ranging from the data link layer to the session layer in the OSI Reference Model. We also leverage the reusability feature of SiP to construct a model of a complete network system using those elements. From both the experience of protocol specification and the result of system-level simulation, SiP is proved to remarkably facilitate the design and performance evaluation of network protocols.

# 1

# Introduction

Modern communication systems are more powerful and complex than their older counterparts. They are at the same time more compact and cheaper due to the improvement of hardware technology. This is achieved by integrating many subsystems into a tiny module, e.g. a single-chip processor [1], and fabricating them together.

The distributed and heterogeneous nature of subsystems enter the picture when this approach is adopted. Even inside such a compact system, protocol elements are necessary to guide data interchange and handle interfaces. For example, a general-purpose micro-controller usually contains control, signal processing and communication elements. Those subsystems could have very different reaction speeds and I/O rates in face of a request to interchange data [25]. Therefore, various protocols are often embedded into that system to implement reliable interaction over unreliable channels, synchronization across distributed elements and security in transactions among system nodes.

To verify the functionality and evaluate the performance of such a system is a difficult task. First, A framework to model and simulate heterogeneous systems is desirable. It should be able to model each subsystem in a natural and efficient manner and have an interface mechanism to integrate them into a whole. Ptolemy, developed at UC/Berkeley, is a system-level design framework that allows mixing of multiple models of computation called domains [4]. Using Ptolemy, users can freely choose the most suitable domain to describes each subsystem and perform system-level simulation. Therefore, Ptolemy is a good candidate as a modeling framework to meet our need.

Another consideration in choosing a simulation tool is the expressiveness to represent a protocol compactly and intuitively. Possible choices are synchronous language [11], process network (PN) [21], finite state machine (FSM) [40] and communicating sequential processes (CSP) [18]. Ptolemy itself provides a preliminary hierarchical FSM domain [3] which allows nested embedded domains in an FSM and any built-in concurrency model. However, some implementation issues of complex guard/action transition and repeated triggering in the current FSM domain of Ptolemy are still envolving. We thus chose a more sophisticated CSP-like description language, PROMELA, for protocol specification. PROMELA (PROcess MEta LAnguage), developed by Lucent Technologies - Bell Labs, is widely adopted in academe for protocol modeling and validation. Associated with the language is an interpreter, called SPIN, to simulate and verify the protocol specification. In this report, we will propose a methodology to integrate SPIN into Ptolemy for simulating protocols in a system context.

The integration will utilize Ptolemy's ability to support heterogeneity. Fortunately, Ptolemy is designed with an object-oriented paradigm and supports heterogeneity using the principle of polymorphism. Its kernel defines basic classes and generic functions. The application-dependent objects are derived from these classes and overridden with specific functions. Also, data abstraction and encapsulation make the maintenance easier. The ultimate goal is to retain a compact and generalized kernel which is extensible. As a result, any object derived from a domain-specific class would be regarded as an specialized object in that domain, but is still reachable from the corresponding basic class. This implies that if the behavior of the derived object follows the loosely predefined requirements, it works well with Ptolemy kernel. An intuitive idea is to encapsulate a desire operation as a regular computation unit in Ptolemy. However, two problems arise by doing this: Does the semantics of synchrony of host domain match the parasitic modules? Is the concurrency model still applicable to them? [13]

Many researchers have done several similar embedding or combination. The Argos language combines FSMs with a synchronous/reactive (SR) concurrency model. SDL embeds FSMs in process networks. Codesign FSM (CFSM) [6] embeds FSMs in DE. Simulink, form MathWorks, Inc., mixes continuous-time concurrency model with FSMs. The main consideration of such a coupling is just the questions we posed. This is because computation cannot be scheduled across two domains without given careful definition of their synchrony and concurrency [10][17].

To integrate SPIN into Ptolemy, we intend to model and simulate protocols with other heterogeneous systems. Therefore, we should select an appropriate domain in Ptolemy as the host platform for SPIN. Leveraging on Ptolemy's ability to support heterogeneous design, SPIN imitates a regular Ptolemy computation unit to interact with units in other domains. In this report, we will show that it is appropriate to embed protocol modules in a discrete-event (DE) concurrency model with careful definition of its semantics.

The rest of this report is organied as follows. In Chapter 2, we propose a hybrid architecture of the domains to model protocols. Based on that proposal, we have developed a software tool by integrating SPIN into Ptolemy. Chapter 3 gives a detailed roadmap of the implementation. In Chapter 4, we specify several fundamental buidling blocks of network protocols using our tool to demonstrate its expressive power. By reusing these blocks, in Chapter 5 we construct an application example involving all protocols we discussed in Chapter 4. Finally, in Chapter 6, we summarize our ideas and draw conclusions.

# 2

# Computational Model of Protocols

In [3], B. Lee et al. characterize a concurrent system as "modules consisting of relatively autonomous agents that interact through messaging of some sort", and gives the definition of its computation model as "the rules of interaction of the agents and the semantics of the composition". This description is general enough to include most popular models in the literature such as a process network, discrete event, synchronous reactive, multi-thread, dataflow, and $\pi$-calculus.

Among these models of computation, the discrete-event (DE) model is especially useful and commonly adopted in modeling distributed or parallel entities together with their communication infrastructure. Its system states evolve at the granularity of the time spans of consecutive events and is assumed static between them. In addition, the transition of states is regarded as instantaneous. It hence well coincides with our perception of protocols, which usually neglect the details of message propagation and respond to occurring events with a negligible latency as compared with the duration between events..

However, an appropriate model to govern the concurrency and synchrony of distributed modules of a protocol is not necessarily a good model to specify the modules themselves. In fact, the behavior of a protocol module is best characterized by a set of control sequences and I/O actions [8] rather than a series of predefined discrete events. Therefore, a control-dominated computation model with the expressiveness of I/O commands would be a good candidate. In this chapter, by distilling protocols, we conclude Communicating Sequential Process (CSP) [15] is a suitable model to specify protocol modules. And, a Finite State Machine (FSM) enables CSP to operate in a modal execution fashion. Our resulting computational model suggests that embedding CSP and FSM in DE will be convenient and adequate in expressing communication protocols. In Section 2.1, we provide a detailed view of protocols and identify their key features. Then in Section 2.2 and 2.3 we give brief introduction to CSP, FSM, and DE. Finally, a proposed hybrid architecture for modeling protocols is illustrated in Section 2.4.

## 2.1 Protocol Specification

Conceptually, a communication protocol is a distributed algorithm that coordinates two or more entities to accomplish a shared or collective task. It uses messages passed back and forth among entities, defining both message format and interpretation and conditional sequences of messages. If one would try to give more specific definition, the terms coordinate, entity, task, conditional sequence, and message all have to be carefully defined [51]. This turns out to be non-trivial because it is equivalent to elaborating the details of the combinational structures and computational models of the algorithm [24]. In next section we will see that the issues of selecting an appropriate structure or model does not have a definite answer. Instead, most of time we compromise on the trade-off between mathematical elegance and intuitive perception [14].

To show the importance of this point, let us look at the internal process of a network interface card where the data-link layer protocol has been built in. While analyzing its performance, we treat the full-duplex link as two separate channels and neglect the interference. In addition, upon receiving a packet we assume the process is able to examine its correctness and then take actions in an instant. Such an "idealization" greatly reduces hassles while formulating the metrics of the communication system [52]. However, we know in fact there is only one single coaxial cable connected to the card and the respondence to an incoming packet does take processing time.

A tool to model protocols must compromise on the issue of abstraction level eventually. Therefore, the expressiveness of the tool has been carefully chosen to considerably match our perception of nature but still retain the simplicity and effectiveness for implementation. Before introducing the adopted description language for protocols, let us return to our conceptual definition of protocols and re-explain those fundamental features in detail as follows.

"**entity**": Usually a hardware device or software code. However, while describing a protocol, it is always useful to isolate the embedded control logic from the actuators to identify the imaginary actors of the protocol. An actor here means an agent process that provides communication services to a physical entity. It could be a single reactive module as well as a combinational aggregation of modules. .

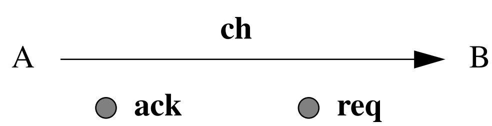
"**conditional sequences**": A series of logical control statements that guard respondent actions. Typical guards are packet arrival, signal triggering, and expiration of timers. A simple form is similar to the IF-THEN-ELSE construct if a specific condition is expected to happen. The CASE-THEN-ELSE is used to switch the execution flow into a certain branch if a corresponding condition in a guard list is satisfied. The switching may be nondeterministic if more than one condition in that guard list are evaluated to be true. In that case, one branch will be chosen from all qualified ones with equal probability.

"**coordinate**": The actions taken by the distributed autonomous processes. There is no overall supervisor directing the interaction among distributed nodes. Instead, each process has a predefined script to decide its response to an event and then enter an appropriate state to keep the system healthy (e.g., no deadlock). By sending out or waiting for a notification event, the physically distant nodes hence coordinate themselves to accomplish data communications.

"**task**": Most of time means to exchange information, i.e., sending and receiving actions plus the data propagation over channels of two communicating processes. Here these actions are described in a high-level sense and we neglect all dependencies of devices and protocols. Specifically, the sending and receiving are just insertion and removal operation respectively  to a queue. However, note that the queue is not necessarily a substantial data structure dedicated to representing the channel. Depending on the synchrony model, it could be a FIFO queue or a collection of separated events on a chronological queue. This flexible definition enables various characterizations of channels such as random order, propagation delay, and packet corruption.

"**message**": The information passing from node to node over a channel. The usual forms of message are packets and signals. A packet usually contains many fields such as control header, data payload, and error detection code. A signal could be a pure triggering or a valued event to notify its counterpart that something is happening.

These explanations characterize the basic requirements of a description language to specify distributed processes. A simple but adequate protocol modeling language, PROMELA, caught our attention because its expressiveness was designed to specify precisely these protocol features. Figure 2.1 gives a PROMELA example specifying a semaphore mechanism that functions as the basis of many asynchronous transmission protocols.




```
PA() {                                   PB() {
loop:                                     loop:
 (req==1)->ch!HEADER,DATA;                req=1;
 ack=1;                                   (ack==1)->ch?h,x;
 (req==0)->ack=0;                         req=0;
 goto loop;                               (ack==0)->goto loop;
}                                        }
```

Figure 2.1 Asynchronous transmission using a semaphore mechainsm.

In this example, **A** and **B** are two distributed nodes connected by a data channel **ch**. PA and PB are two processes built in **A** and **B** respectively which coordinate the transmission. Signals **req** and **ack** are accessible to both PA and PB and are used to inform the other node that system status has changed. Buffer **h** and **x** are the temporary spaces where header and data are stored. A typical scenario starts from setting **req** to 1 by PB. As soon as perceiving the change of **req**, PA sends HEADER and DATA onto channel **ch** and sets **ack** to 1. Seeing **ack** turned on, PB stores HEADER and DATA in buffer and resets the **req** signal. This resetting results in the releasing of signal **ack** by PA, and then allows both PA and PB to return to their original states. At this point, PA and PB are ready for the next iteration.

This example shows how effective PROMELA can express the protocol features discussed. It uses independent processes to represent distributed "entities". The "conditional sequences" guarding the evolution of system state are given by Boolean expressions. The actions updating the system state to "coordinate" processes are done by assignments. The I/O actions to "exchange" data through channels are succinctly abbreviated as ? and !. "Messages" passing over channels are easily formatted by explicitly enumerating all fields in order. Moreover, the sequential specification fits well the convention of designing protocols by examining intended scenarios. We will return to PROMELA in the next chapter.

## 2.2 CSP and FSM

One way to understand PROMELA is to look at its original computational model, CSP. As its name suggests, CSP allows us to describe a concurrent system by a group of sequential processes which take part in sequences of events. Those processes operate independently and communicate with one another over well-defined channels. To justify the appropriateness of specifying protocols using CSP, the rest of this section we will examine a simple polling protocol to highlight the notation and semantics of CSP.

Figure 2.2 shows the CSP specification of a simple polling protocol. The three defined processes are running concurrently and each executes sequentially. Notation c?x:M stands for a guard which waits for a message **x** of type **M** from channel **c**. If that message has not arrived, this statement blocks the execution flow of the process. The symbol "[]" is followed by an alternative to the uppermost condition. Note these collateral conditions are not necessarily mutually exclusive. Notation c!x denotes sending a message **x** onto channel **c**. The "->" symbol simply means "then do".

```
Sender      = (ch?y:{POLL}->dataIn?x:msgOK
               []ch?y:{NACK})->ch!x->Sender
Requester   = (ch!POLL->Receiver)
Receiver    = (ch?x:msgOK->dataOut!x->Requester
               []ch?y:msgOK'->ch!NACK->Receiver)
```

Figure 2.2 CSP specification of a simple polling protocol.

The protocol works in the way that the Requester first sends a POLL message to the Sender. After the Sender have seen the POLL message, it retrieves a protocol data unit (PDU) from the local **dataIn** channel and sends the PDU to the designated channel. Once the Receiver gets the PDU correctly, it delivers the data to local **dataOut** channel and revisits Requester to send another POLL message. If the Receiver receive a corrupted PDU, a NACK message will be sent and the Sender will resend another copy of the PDU on receiving the NACK.

Though the CSP specification is organized and self-explanatory, it lacks hierarchy. Suppose now the Sender has two superstates, running and suspended. Everything works normally in the running state and stops totally in the suspended state. In addition, the transition between these two states can happen at any time [23]. It would be cumbersome to add a Boolean condition (state==running) in front of every single statement in Sender to implement the high-level behavior. This shows a drawback of CSP for its inconvenience in specifying behaviors hierarchically because basically all processes are flattened out [43].

CSP is not the only control-dominated computation model. FSM has long been used to specify intricate control sequences [16]. Some elaborated FSM such as Statecharts [7] allows FSM to be hierarchically and concurrently combined. To examine its expressive power, we redo the modeling by using hierarchical FSM (HFSM) and show it in Figure 2.3. The resulting state diagram models the protocol, but is not as clear as the textual representation in CSP. This impression comes from two side-by-side observations. First, scattered guard/action pairs mess up their relativity as compared with the aligned ones in textual form. Second, unimportant states complicate the diagram as all states at the same level have to be explicitly shown. Such complexity is aggravated when the specification is further elaborated. Though applying more hierarchy helps to simplify the diagram at each level, we lose the sequential continuity of logical statements.

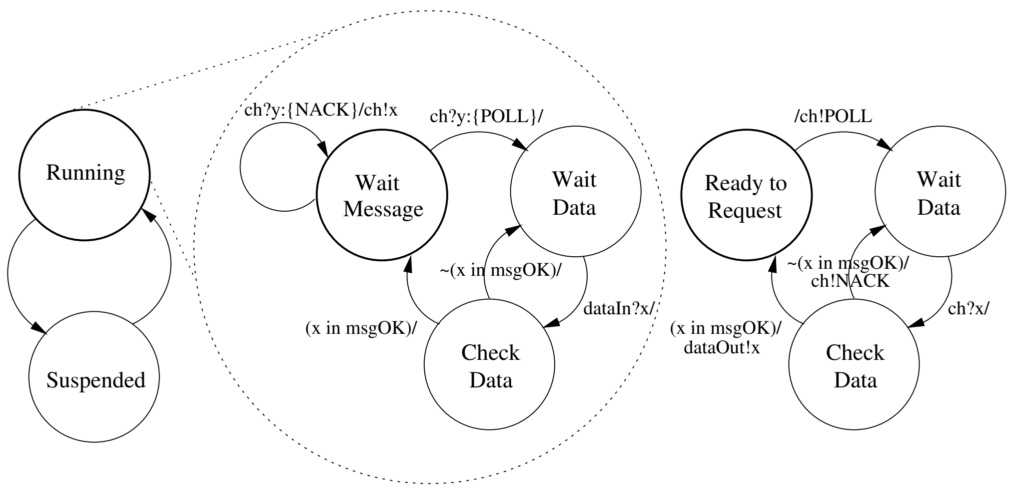



Figure 2.3 An HFSM representation of a simple polling protocol.

Fortunately, W.-T. Chang et al. advocate a new family of models of computation called *charts, which decouples the concurrency model from the hierarchical FSM semantics [12]. Therefore, using *charts allows embedding a CSP model in a hierarchical FSM to solve our dilemma of choosing CSP or FSM in representing a protocol. Specifically, automata which implement the fundamental elements of a protocol are specified in CSP to retain the logical clarity, but they all unavoidably appear as the leaf cells of the hierarchy. HFSMs are then applied to group those automata in consideration of their functionality, geographical location, interface encapsulation, or behavior at higher levels. Since we treat CSP modules as automata, such a heterogeneous hierarchy is straightforward. Figure 2.4 shows an example of embedding a CSP module inside FSM. Simply mapping each Boolean expression and event waiting as a state plus a guard, assignment and event emission as an action, and symbol [] as a new transition, we can always transform a simple CSP module into an FSM. Figure 2.5 shows the resulting hierarchical diagram.

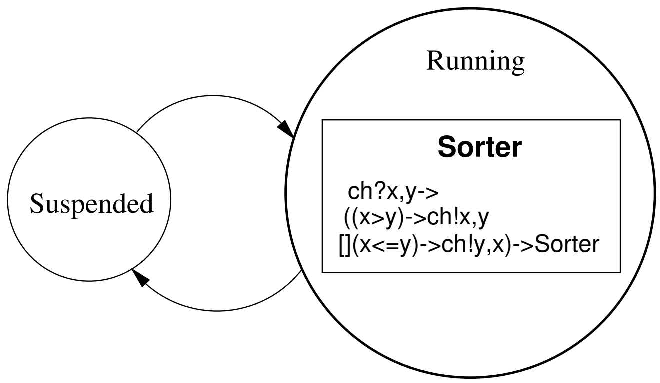



Figure 2.4 Embedding a CSP specification in an HFSM diagram.

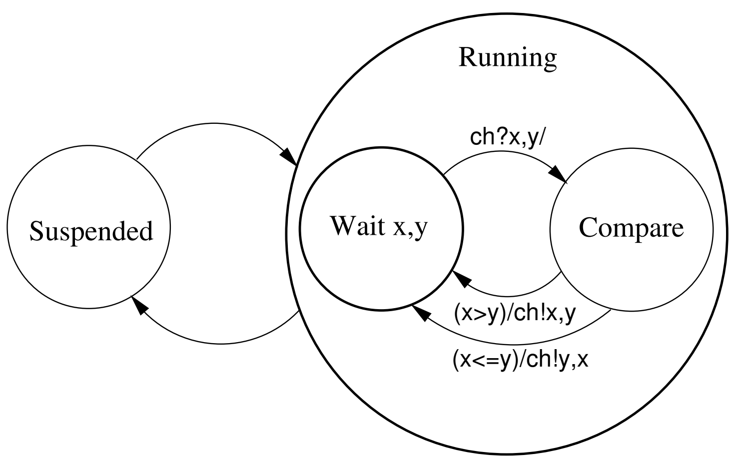



Figure 2.5 A direct transform of a simple CSP specification into an HFSM.

By using this embedding methodology, Figure 2.6 gives our ultimate specification of the simple polling protocol with a succinct and intelligible diagram.

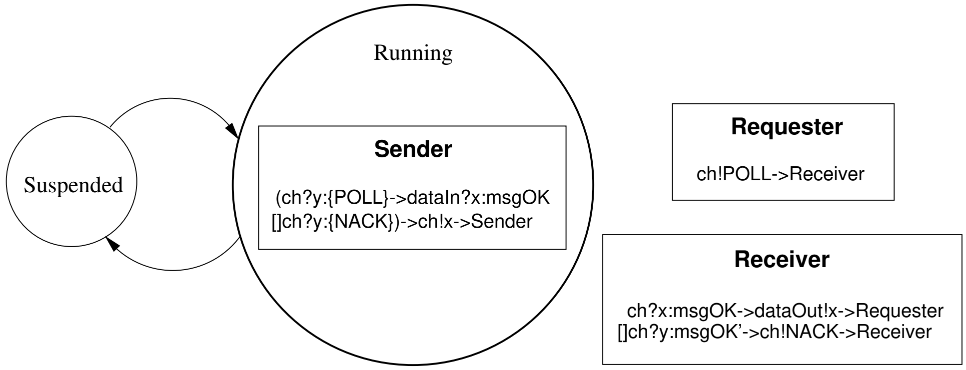



Figure 2.6 A hybrid representation of a simple polling protocol.

## 2.3 Discrete-event Model

The discrete-event model of computation is the most popularly adopted semantics for modeling distributed or parallel systems in computer-aided simulations [5]. This fact results from the trade-off between our perception of nature and the ability of computers. While conducting a simulation using digital computers, computation is inevitably discrete. This limitation leads to the discreteness of state evolution and that contradicts our recognition of temporal continuity. One compromised choice could be simulating the system with condensed source events. We then obtain a discrete version of system state evolution, which is similar to a sampled version from the continuous one [38]. In order to sort those discrete events chronically as well as to synchronize parallel subsystems, the DE model carries a notion of global time to indicate the occurrence of events [26]. These time stamps help to pinpoint system states on the time axis and form a discrete version [53]. As long as the time span between each two consecutive events remain short, the discrete version gives a good approximation to the real world [39].

Since a protocol is a collection of rules guiding the interaction among distributed and parallel processes, the DE concurrency model [20] also applies to the simulation of protocols and their underlying communication infrastructures. However, the semantic subtleties while combining DE, FSM and CSP have to be carefully examined and defined before we can do so. For example, how should a process interact with its counterpart when both the synchronization mechanisms of DE and CSP are acting? How should a signal be converted while it is running through the interface between DE and CSP models?

In order to embed CSP inside DE, we examine how a CSP module refines a DE computation unit (actor). When a DE actor fires [19], which occurs when there is an event at one of its inputs carrying the earliest time stamp, the CSP module imitates the DE actor and responds to the environment. Several data associated with the event are used to update the state of the CSP module:

1. The time stamp of the event is used to adjust the timers declared in the CSP module.
2. The "present" indicator corresponding to the input port where the event arrives is set.
3. A valued event uses its accompanied value to update the internal CSP variable designated to the input port.
4. A message event forwards the message to the internal CSP channel designated to the input port.

After updating, the CSP module examines its currently blocked condition and executes statements as many as the encountering conditions are non-blocking. The execution is regarded as an instantaneous action that takes zero time. Upon reaching the first unexecutable statement, the CSP module outputs new events, if any, and surrenders control to the DE environment to finish one iteration. Each of these outputting new events could be a pure event, a value, or a message generated by an assignment or sending command during the iteration. However, in DE, they must be assigned a time stamp to denote their birth times. Recall our assumption of zero-delay execution, these events are assigned the same time stamp as the input that triggered the reaction.

Consider the example shown in Figure 2.7. Suppose that an event **p** with a earliest time stamp **t** arrives at port **a** of process **A**, and both process **A** and **B** are in their initial states. The DE system reacts as follows:

1. Fire **A**: The waiting for event **p** is satisfied after forwarding the pure event **p** from input port **a** to internal channel **a**. **A** then sends a pure event **q** to channel **c** and that is immediately wrapped with time stamp **t** and put onto the output port **c**. After that **A** still tries to execute more statements, but there is no **reset** event shown on channel **b**. This blocking forces **A** to surrender control to the DE environment.

2. Fire **B**: **B** takes event **q** from channel **d** and sends out event **r** with time stamp **t** to channel **e**. After that **B** returns to its initial state, i.e., waiting for another event **q** from channel **d**. Since there is no more event in channel **d**, the statement is unexecutable and **B** hence surrenders the control.

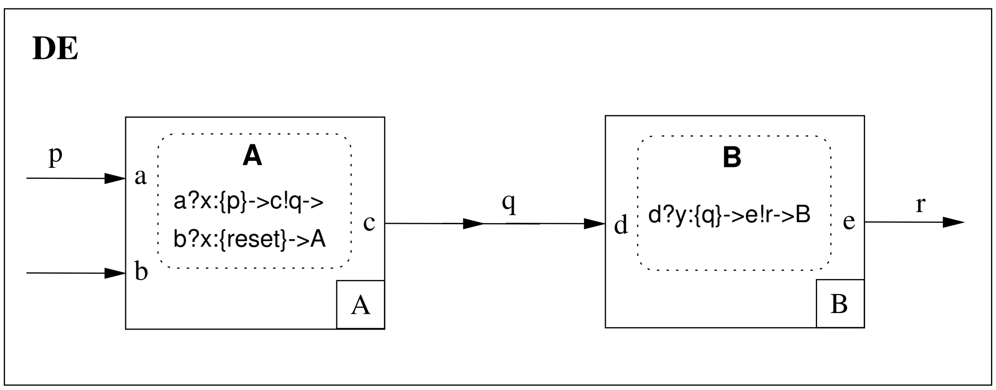



Figure 2.7 Two CSP modules that refine DE actors.

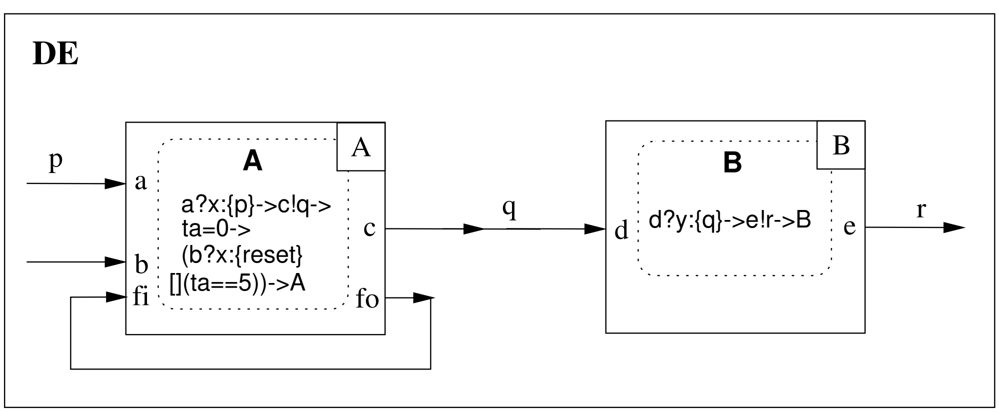



Figure 2.8 Timed-CSP and untimed-CSP modules refine DE actors.

Now, suppose that **A** has a timer **ta** which synchronized with the environment to measure the time **A** has waited for the **reset** event. A timeout event happens when **ta** reaches 5 time units and the **reset** event has not arrived at channel **b**. In this case, **A** resets itself anyway as having received the **reset** event. The new specification of process **A** is given in Figure 2.8. Note that we have to equip **A** a pair of feedback ports **fi** and **fo** for self-triggering. A detailed explanation of implementing such timing features in CSP [41][55] is discussed in the next chapter.

## 2.4 A Hybrid Architecture

Summarizing the proposed structure of domains in above sections, we depict an ultimate modeling architecture in Figure 2.9 using the same simple polling protocol example. The DE model serves as the host environment where CSP modules and hierarchical FSMs containing CSP leaf cells sit in. Three features make this hybrid architecture a compelling model for protocol modeling and simulation:

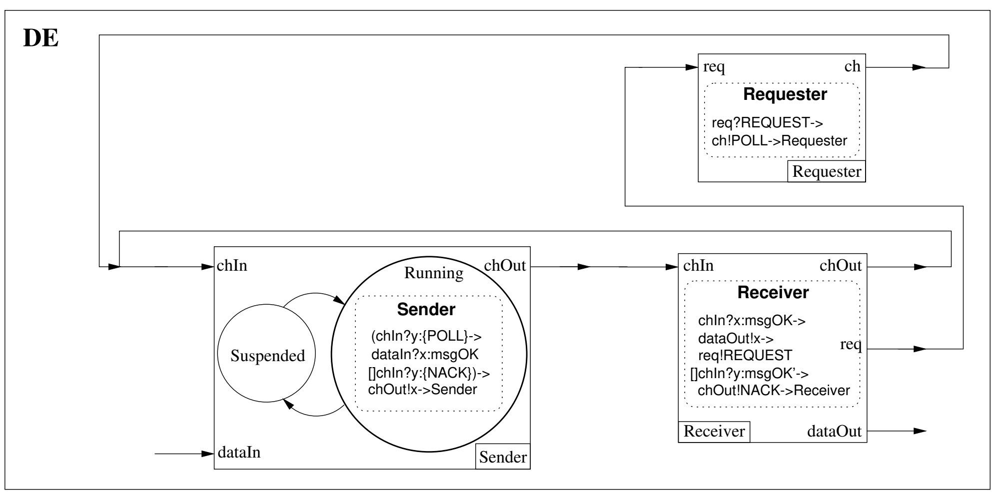



Figure 2.9 Proposed hybrid architecture for modeling protocols.

First, specifying protocol elements in CSP matches our intuitive perception in distributed communication and parallel tasking. Its textual representation also retains a clear and intelligible form in notation.

Second, hierarchical FSMs enable a modal execution of CSP modules. Similar to the command "watch" in some synchronous languages [11], embedding CSP in FSM allows activating and suspending CSP modules at any state. It also helps to represent a protocol at different levels of abstraction.

Third, DE model provides a global system time which facilitates the adjustment of the timers in timed-CSP modules. It therefore proliferates the timing statements in timed-CSP that remarkably reduce the burden in specifying timing behaviors.

# 3

# Software Implementation

In this chapter, we present the details of the software tool that realizes our proposed hybrid architecture for protocol modeling and simulation. To leverage on existing tools, we integrate SPIN, the interpreter of PROMELA, into Ptolemy, a framework providing many domains including FSM and DE. The integration faces many challenges such as coordination of two simulation kernels, event conversion and forwarding, implementation of timed-CSP statements, and scheduling of CSP and DE.

We give a brief overview of Ptolemy in Section 3.1 and point out a possible niche in its structure to accommodate an external tool. Then we explain the simulation kernel of SPIN and discuss the extension of its input language PROMELA to include temporal expression in Section 3.2. Section 3.3 describes the implementation considerations while embedding SPIN in Ptolemy. Finally, we introduce our tool SiP (SPIN in Ptolemy) in Section 3.4.

## 3.1 An Overview of Ptolemy

Ptolemy is a modeling and simulation framework for heterogeneous systems. It covers many aspects of designing signal processing and communication systems, ranging from algorithms, system modeling, simulation, through parallel computing, software/hardware synthesis, and real-time applications. The non-dogmatic kernel of Ptolemy allows users to freely choose a best matched domain to specify each of the subsystems from many built-in domains including synchronous/Boolean/dynamic dataflow, discrete-event, process network, etc. Ptolemy also functions as a coordination framework that deals with the scheduling of simulation across all mixing domains.

The basic computation unit of modularity in Ptolemy is the Block. A system modeled by Ptolemy can thus be viewed as an interconnected block diagram. Blocks communicate one another by propagating streams of messages/data through links among them. Derived from Block, a Star is the lowest level object in Ptolemy which contains a module of code that is invoked at run-time. Also derived from Block, a Galaxy may hierarchically contain both Galaxies and Stars to form a computation unit at a higher level. As expected, Universe is the name of the object that contains a complete system.

Every Star in Ptolemy contains a "go()" method which will be executed every time the Star is triggered. Typical scenario of the "go()" method is first examining Particles present at the input ports of the Star, getting Particles and performing computation, and then generating new Particles on the output ports. We found the "go()" method is actually a great niche to store the code for communicating with an external tool. Such a bridging "go()" method contains 3 parts in our design:

1. Get data from input ports and convert them into the format used by the external tool.
2. Call an external procedure to perform an iteration of computation.
3. Wrap up the computation results and put them on output ports.

This is our main idea of the agent star described in Section 3.3. By wrapping up a SPIN process to imitate a Star, we enable Ptolemy kernel to execute external computation without modifying the kernel..

## 3.2 SPIN and PROMELA

SPIN is a tool allows simulating and validating distributed modules of concurrent systems. Actually, it was originally designed to perform simulation and verification of communication protocols. In this report, we only focus on its ability of simulation and try to modify and integrate it into Ptolemy.

The input language to SPIN is called PROMELA, which is a description language for extended FSMs. Its syntax loosely bases on Dijkstra's guarded command language notation and C.A.R. Hoare's language CSP [30]. PROMELA supports only three types of objects: processes, variables, and channels. Processes are like C functions in design, and like UNIX processes in behavior. The body of a process is a sequence of CSP-like statements that specify the behavior of a distributed entity. Variables can be global or local, and can be given values by assignment or receiving statements within proper scopes. Supported types are Boolean, bit, byte, integer, and user-defined structures. Channels are essentially queues that shared among processes. A channel is declared to pass a certain type of message, and is given a fixed finite length.

One obvious shortage of PROMELA, similar to most reactive model description language, is the lacking of temporal statements. However, the correct functioning of a distributed real-time system depends on the timely coordination of its interacting components [22]. The protocol elements thus inevitably have to react according to those timing requirements. In Section 3.2.1 we propose several temporal statements and their reacting semantics.

### 3.2.1 Extending PROMELA's Expressiveness

The original PROMELA grammar has neither timer data type nor timing commands. The SPIN simulation kernel regards the execution time of each atomic PROMELA statement, a single command or an atomic block of commands, as one iteration. Therefore, every time span between two consecutive atomic statements is considered as a universally equal and indivisible duration. This assumption looks awkward while coupling SPIN and DE domain because DE model requires each operation having been assigned an execution duration. For regular DE computation units, the duration could be either constant or variable, and is assigned as the total consuming time of the executed commands in one iteration. This is acceptable if the operation is similar in each iteration such as parity bit checking or extracting header from a packet. However, during each iteration a protocol module could execute a very different set of commands and hence a fluctuating execution time. One usual way to work around it is to define the duration of each executed atomic statement as one time unit. But, this assumption seems too coarse since it may regard a long arithmetic computation and a simple register shifting taking same operation time.

We adopt a more flexible approach to specify the execution time of PROMELA code. A Programmer could place a **delay**(*duration*) command after each atomic statement whose execution time is not negligible and assume those ahead it are zero-delay. For example, suppose during some state a protocol module needs to perform two register shifting and one shortest path searching, we could place **delay**(10) after that searching procedure to indicate the aggregate duration of these three statements is 10 time units. Besides, the **delay**( ) command can also be used to assure the correctness of received data if signal settle time and bus skew time are taken into consideration while modeling a bus I/O protocol.

Another temporal event in protocol specification is time-out. A typical case is to start a timer after sending a packet and retransmit the packet if an acknowledgment has not been received after a predefined duration. To specify this timing mechanism in PROMELA, we need to create the timer data type. A timer can be reset to any starting time at any place of codes by assigning a value. Programmers are allowed to use as many timers as necessary and have them running simultaneously. The **expire**(*timer*, *target-time*) command is used to check if a specific timer has expired as well as to register a likely time-out event in the future. Note the registered time-out event is not deterministic to happen since other events could abort the waiting state or a timer reset command could change the target-time.

Routine state checking is also useful while specifying a protocol. A protocol module may enter an idle state for a long time and be unaware of something going wrong. In this case, programmers could use a timer and set a target-time to inspect states again. Or, implicitly, using command **return**(*duration*) will register a promissory return time to invoke the module again.

To enhance PROMELA with these temporal features, we have to modify the parsing rules of PROMELA and give corresponding execution codes in SPIN. Our current implementation already includes all the features mentioned above. In addition, we allow timers to be mixed with or assigned by other arithmetic expression. This requirement comes from that fact that timing is usually a parameter to other functions and target-times are often calculated by some formulas. Moreover, It is also permitted to apply timers to comparison operations such as >, <, ==, etc. This facilitate programmers verifying the timing at any moment before the timer has reached its target-time.

### 3.2.2 The Simulation kernel of SPIN

The simulation kernel of SPIN is implemented as an interpreter of PROMELA. It relies on yacc to build a parse tree before the simulation can be started. Also, many symbol tables will be established to facilitate the evaluation of variables, operation of queues, and control of program flow at run-time. Figure 3.1 gives a high-level view of the parse tree where two processes and their variables and statements are shown.

The scheduler of SPIN randomly picks one sequential statement from one of all non-blocking processes and calls the evaluator to execute that statement. The evaluator then updates variables if the statement contains assignments, or decides next statement to be executed if the statement is a control flow command. Assume the initial value of PC (next process to choose) in Figure 3.1 points to process A, and the internal PC of A and B point to statement P and Z respectively. Possible execution scenarios are PZQR, PQZR, and PQRZ if the control flows in A and B remain sequential and the execution blocked after R or Z.

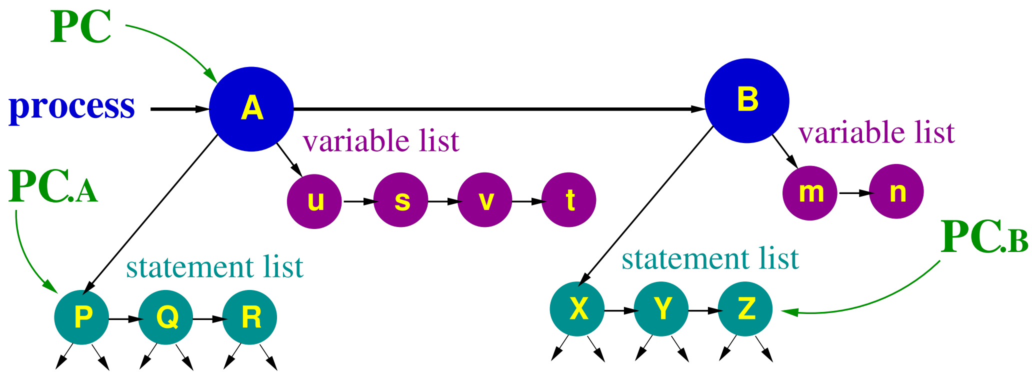



Figure 3.1 The parse tree of processes built by SPIN simulation kernel.

Most of our modification is made to the scheduler and evaluator of the SPIN simulation kernel. We disabled the nodeterministic scheduler of SPIN and let the Ptolemy DE scheduler take over the scheduling. We also rewrote the core subroutines of the evaluator so that timing statements, floating-point operations, and external C function calls could be understood by SPIN.

## 3.3 Integrating SPIN into Ptolemy

The way we integrate SPIN into Ptolemy is to have both their simulator kernel running at the same time. This approach requires an interface to interchange data, events, timings and other more subtle information such as pointers of functions between SPIN and Ptolemy. Our idea is to create an agent star for each protocol module written in PROMELA. An agent star is regarded as a regular DE star by Ptolemy and is in charge of passing all information back and forth between SPIN and Ptolemy such as propagating data through the input and output channels, so-called ports, of a star and a protocol module.

To choose a suitable class from DE stars, we at first enumerate the features of a PROMELA protocol module, so-called process, and select the star class which quite describes those attributes. A process has I/O channels, parameterizable states and it is able to re-invoke itself after a specific duration. In addition, it always consumes all simultaneous incoming events before reacts to the environment like outputting data or emitting new events. Also, it is possible that many processes trigger one another simultaneously without a deterministic order. We soon found the DERepeatStar class with Phase mode and Delay type fits these requirements very well. Therefore, we let all agent stars be derived from the DERepeatStar class and tuned to Phase mode and Delay type immediately after construction.

### 3.3.1 Communication Ports

In order to bind ports and states, we need to understand the data structure of local variables of a process in SPIN. They could be single-space variables, arrays, FIFO (first-in-first-out) queues, or arrays of FIFO queues and they are all allowed to be ports or states. To improve execution efficiency, we create a pointer for each port and state variables and make the links at the first visit to the agent star. Also, at first visit, state variables are assigned the values which were parameterizable from Ptolemy environment as their initialization. After then, data arriving star ports are written to the data structure of SPIN through their corresponding pointers.

The FIFO queues are accessed by using queue functions provided by SPIN. Specifically, an agent star repeatedly gets a data unit, so-called particle, from an input port and forwards it to the corresponding FIFO queue in SPIN. In the other hand, if a port variable has been updated during an iteration, its updated value will be emitted to the output port of the agent star with an appropriate time stamp. Similar actions apply to the FIFO queues if there are some data having been inserted into the queue during an iteration. Figure 3.2 illustrates how a SPIN process is bound with a Ptolemy star and Figure 3.3 shows the forwarding paths between them.

A process not only cares the value of an incoming data, but also needs to know if there are new events, i.e. new data, arriving a specific port. We introduce command **present**(*port*) to test if a new particle has arrived at the given port and

**turnoff**(*port*) to turn off that indication. The command **admit**(*port*) is used as shorthand of testing and turning off immediately. An input port is also allowed assigning the sustainment of present indication. The attributive keyword **persist** indicates the present indication is persistent until it has been turned off explicitly. Without declaring **persist**, an input port is considered volatile which retains the indication only at the arrival time of a particle. As time proceeding, it will be deactivated automatically.

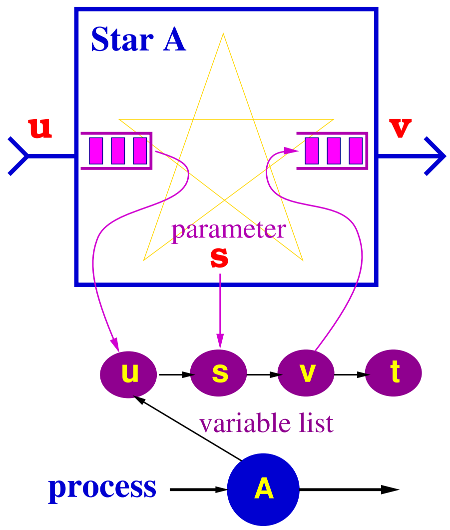



Figure 3.2 Extra pointers are used to bind the I/O prots of Ptolemy with the corresponding variables in SPIN.

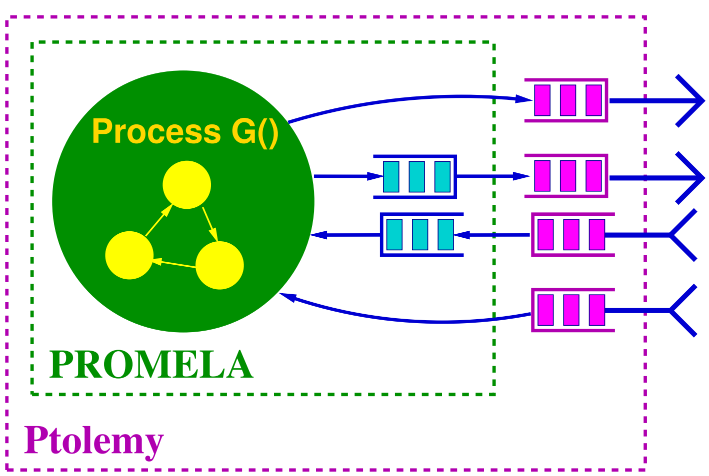



Figure 3.3 Particles are forwarded between Ptolemy and SPIN.

### 3.3.2 Discrete-event Agent Star

Recall that an agent star only bridges ports between SPIN and Ptolemy, itself does not perform any operation specified in the protocol module it serves. For any incoming event, the agent star first forwards the particle, indicates signal present, and then calls a PROMELA interpretation procedure in SPIN to take over execution. Given the process identification number passed by the agent star, SPIN locates the desired process and reloads its program counter to resume interpretation. As we defined in last section, all consecutively executable PROMELA statements without a **delay**(*duration*) command beneath are considered zero-delay. Therefore, SPIN always processes PROMELA codes continuously until an unexecutable statement is reached and then it returns control to Ptolemy. However, this does not imply that statement is forever unexecutable because other processes may change the situation. If it does never get through, it is most likely an incorrect protocol design which leads the system entering a deadlock or an abnormal termination.

There are four cases of unexecutability. First, a **delay**(*duration*) command is always unexecutable because it will not be satisfied until global time has proceeded by that duration. Second, an **expire**(*timer*, *target-time*) command will not be executable until the timer reaches its target-time. The third situation is the most usual one, a logical false condition. For example, an expression *ACK==1* is regarded as an unexecutable statement if *ACK* is not equal to 1 at that moment. The event present test command **present**(*port*) is considered as a logical expression as well as all Boolean-typed functions. The last case is executing a **return**(*duration*) command. This is obvious as the function of **return**( ) is just to register a future visit before it yields current control to Ptolemy simulation kernel.

### 3.3.3 Firing Mechanism

So far we have solved the semantics and implementation details related to port binding and execution control transferring. However, they are not the main reason we choose the DERepeatStar class as the base class of agent stars. The specialty of a DERepeatStar is that it is equipped a pair of feedback I/O ports by default. By placing a particle with an appropriate time stamp onto its feedback output port, a DERepeatStar is able to re-invoke, so-called refire, itself at any future time. This is because that particle will follow the feedback link back to the star's feedback input port and become a triggering event when the global time reaches the moment as the time stamp of the particle. Therefore, for the cases that SPIN yields control caused by timing commands such as **delay**( ), **expire**( ) and **return**( ), the agent star is able to schedule itself a future refiring by using feedback ports. As for the logical unexecutability, the agent needs not schedule any refiring since that will eventually be solved by some input events sending from other processes if the protocol was correctly designed [50].

The refiring time is assigned the earliest expected epoch when the unexecutable statement may become executable. As a result, it is true that we can not estimate the time when a logical unexecutability would be solved, and hence we do not schedule a refiring for it. Nevertheless, we are able to schedule the refirings for timing conditions. For example, a **delay**(*duration*) command definitely suspends the process for *duration* time units. The refiring time stamp is simply current global time + *duration*. This also applies to the command **return**(*duration*). Their semantic difference is that **delay**( ) absolutely stops the evolution of process during suspended time while **return**( ) allows other triggering to awake the process prior guaranteed reentry. Command **expire**(*timer*, *target-time*) leads to schedule a refiring at current global time + *target-time* - current value of *timer*. Note this scheduling will keep updating as *timer* and *target-time* may vary before timeout. Briefly, the principle of timing refiring is to invoke the process exactly at the time it becomes executable. Otherwise, the evolution of that process is delayed and thus the simulation violates the definition of concurrency. Such being this case, the modeling of current processes is distorted and the simulation result is incorrect.

In addition to regular event firing and timing refiring, the broadcasting event firing also awakes agent stars. It is often used in specifying protocol modules communicating through shared media such as a topology with a shared bus and radio broadcasting via atmosphere. Since construction, every agent star is endued a state parameter *medium* which could be freely designated. During simulation, any agent star could listen and/or broadcast events to all other members on the same medium, and have them be invoked to check if any further state transition is possible. We propose two ways to specify the medium an agent star belongs to: explicit assignment and implicit scope. The former method categorizes agent stars into different medium groups according to the given medium names throughout hierarchy. The implicit scope method defines medium groups by the hierarchical levels of the protocol structure. Based on the Ptolemy design paradigm, the level of a star is uniquely determined by its parent compositional blocks, so-called galaxies. Specifically, the compositional architecture decides the scopes of media.

## 3.4 The Tool, SiP

SiP (SPIN in Ptolemy) is a preliminary software implementation of the protocol modeling and simulation methodology proposed in this report. Its experimental prototype is announced in SRC Annual Review, Austin, March 1998. The first version SiP1.0, as a patch package supplemented with Ptolemy, was released on June 29. SiP1.1, supporting C++ function calls in PROMELA, was released on August 1. And SiP1.2, which allows floating-point operations in PROMELA, was released on September 10. All packages and their installation instructions could be downloaded from the URL of http://ptolemy.eecs.berkeley.edu/dgm/protocol/.

SiP contains four major components:

1. A Ptolemy language code generator for agent stars, called **ppl2pl**.
2. Add-on Ptolemy source codes supporting agent stars.
3. A Ptolemy-supported SPIN package.
4. A protocol element library.

A typical scenario of protocol modeling and simulation using SiP is described as follows.

1. Specify each newly defined protocol module by PROMELA (enhanced with new features).
2. Use **ppl2pl** to generate the Ptolemy language codes of the agent star for each new protocol module.
3. Make the agent stars of new protocol modules under Ptolemy environment. After doing this, a reusable icon for each new protocol module is created as the modules in build-in library.
4. Specify the architecture of protocols and the connections between protocol modules and system elements. Protocol modules can be grouped to form more abstract compositional blocks as galaxies in Ptolemy.
5. Perform system-level simulation to verify the functionalities of the testing protocols.

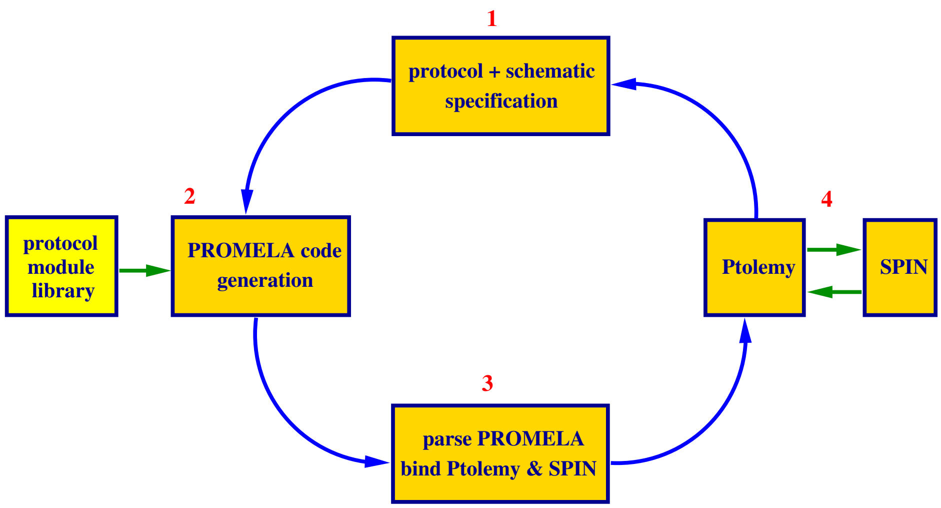



Figure 3.4 Four phases of SiP's running cycle.

Figure 3.4 gives a closer view of the running cycle of SiP, which can be categorized into four phases. Phase 1 is the Specification Phase indicating the editing of protocol modules as well as system construction. After received a simulation request, SiP enters Phase 2 to generate and preprocess PROMELA codes of all the protocol modules on the system schematic. Once completing PROMELA code generation, in Phase 3 SiP first has the SPIN parser to construct the parsing tree for each module, and then it binds all interfaces between each pair of agent star and PROMELA process. Figure 3.5 gives the detailed view of a pair of PROMELA process and its corresponding parse tree in SPIN at this point. Next, SiP starts Phase 4, the Ptolemy-SPIN Co-simulation Phase, to perform system-level simulation. Finally, the result of simulation feedbacks to Phase 1 to help the designer validate the functionalities of protocols and evaluate the system-level performance.

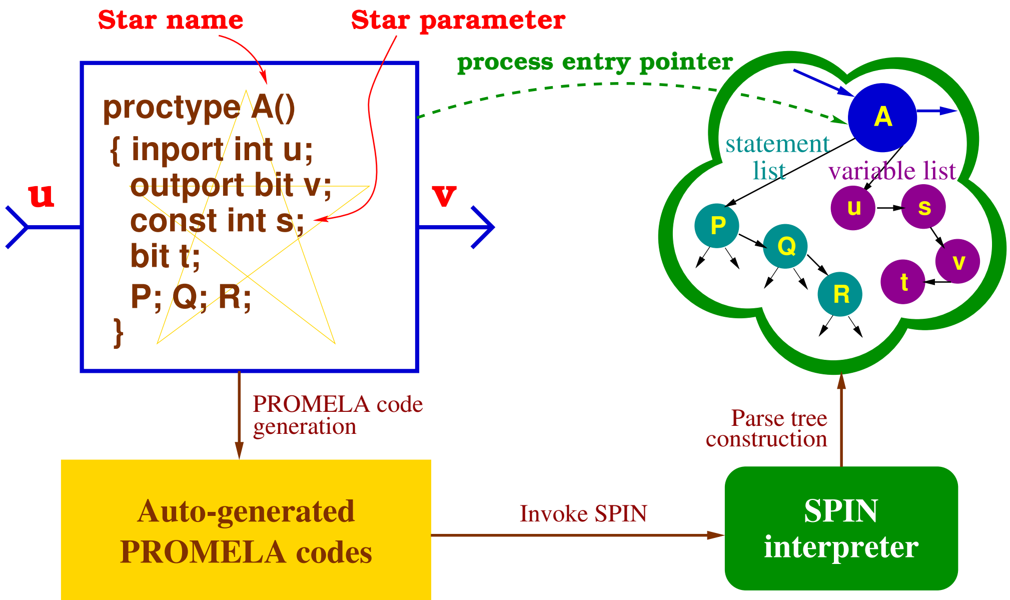



Figure 3.5 The side-by-side comparison of a PROMELA process and its corresponding parse tree built by SPIN.

# 4

# Elements of Network Protocols

To justify the effectiveness of our tool, in this chapter we examine and specify several network protocols using SiP. Although the following cases are fundamental building elements of protocols, they become reusable modules after represented in SiP. Leveraging on the cumulative designs of new modules, one can always construct more sophisticated protocol elements by exploiting the hierarchy capability of the tool.

## 4.1 Connections

Data communication services in a network can be categorized into two types, connectionless and connection-oriented. Typical examples are the Public-Switched Telephone Network (PSTN) and the Internet Protocol (IP) switching network respectively. A connectionless service allows a node sending data packets to another node without having obtained a permission from it previously, while a connection-oriented service needs a connection setup phase to guarantee the quality of service (QoS) [28].

Specifically, in a connectionless communication, the switching process PA can transfer a packet to its counterpart process PB at another switch each time the packet is ready to be sent out. Both of the two switches have no information about whether if the conducting packet is belonged to a certain data stream. Also, they have no knowledge of the traffic of the subsequent packet flow. Consequently, PB may start discarding packets when it runs out of buffers.

Unlike the connectionless communication, a connection-oriented service requires PA to establish a connection to PB by a setup procedure before it sends first data packet to PB. The established connection can later be disconnected by a disconnection procedure similar to the connection procedure. We call the rules of the procedures to establish and disconnect a connection as a connection protocol [14].

Briefly, to establish a connection, PA first sends a connection request (CON_REQ) to PB and waits for its response. After received the request, PB checks the availability of its resource and replies PA with a positive acknowledgment (CON_ACK) or a negative rejection (CON_REJ). Once the connection has been established, data packets can be transferred continuously from PA to PB. To disconnect the connection, the disconnection request (DIS_REQ) can be issued by either PA or PB. And, to confirm that request, the one received DIS_REQ replies CON_REJ as a confirmation.

We first use SiP to model the connection requesting side, i.e. PA, as follows.

```
#define IN_BUFF 32
#define OUT_BUFF 32
#define DATA_BUFF 256

mtype = { CON_REQ, CON_ACK, CON_REJ, DATA, DIS_REQ, IDLE, SETUP,
  CONNECTED, TEARDOWN }

proctype PA()
{
 inport chan pktIn = [IN_BUFF] of { int };
 inport chan dataBuf = [DATA_BUFF] of { int };
 outport chan pktOut = [OUT_BUFF] of { int };
 const int SETUP_TIMEOUT=64;
 const int DATA_TIMEOUT=5;
 int tempPkt;
 int state=IDLE;
 timer t1;
 do
   ::(state==IDLE)->
    if
```

```
    ::(len(dataBuf)>0)->pktOut!CON_REQ; state=SETUP; t1=0;
    ::(len(pktIn)>0)->pktIn?tempPkt;
    ::else->return(0);
   fi;
  ::(state==SETUP)->
   if
    ::pktIn?CON_ACK->state=CONNECTED; t1=0;
    ::pktIn?CON_REJ->state=IDLE;
    ::(len(pktIn)==0)->
     if
      ::expire(t1,SETUP_TIMEOUT)->state=IDLE;
      ::else->return(0);
     fi;
   fi;
  ::(state==CONNECTED)->
   if
    ::pktIn?CON_REJ->state=IDLE;
    ::(len(dataBuf)>0)->dataBuf?tempPkt; pktOut!DATA; pktOut!tempPkt; t1=0;
    ::(len(dataBuf)==0)->
     if
      ::expire(t1,DATA_TIMEOUT)->pktOut!DIS_REQ; state=TEARDOWN;
      ::else->return(0);
     fi;
   fi;
  ::(state==TEARDOWN)->pktIn?CON_REJ->state=IDLE;
 od;
}
```

Figure 4.1 SiP specification of the connection process at requesting side.

Process PA uses a pair of I/O ports, **pktIn** and **pktOut**, to communicate with process PB. It also provides a service access point (SAP), **dataIn**, for the entity it served, **A**, to input data packets. In above specification, we adopt a timer **t1** to simplify the interface between PA and **A** instead of having one explicit control port and one status feedback port. The former method lets timeout event initiate a disconnection request automatically while the later method requires an explicit external controlling signal.

Process PA has four states, IDLE, SETUP, CONNECTED, and TEARDOWN. Initially, the state is set to IDLE. Once PA gets the first data packet from A, it enters the SETUP state and starts to establish a connection with PB by sending CON_REQ to it. At the same time, a timer is started to prevent PA from waiting for PB's reply forever. If PA does receive a correct reply, it will be either CON_ACK or CON_REJ and that decide the next state of PA to be CONNECTED and IDLE respectively. Or, the reply is lost and t1 expires. In this case, PA sends CON_REQ again and reset t1 to start another trial of connection.

When a connection has been established, PA sequentially forwards data packets from **A** to PB. If there is no more data packet in **dataIn** for DATA_TIMEOUT time units, we assume this is the case that **A** has already sent all data and a disconnection request DIS_REQ should be sent immediately. By doing that, PA enters the TEARDOWN state and wait for CON_REJ from PB to confirm the disconnection. In fact, a CON_REJ from PB at any moment will force PA back to the IDLE state.

Compared with PA, PB is simpler in the connection protocol as it only has two states, IDLE and CONNECTED. Its SiP specification is listed below.

```
#define IN_BUFF 32
#define OUT_BUFF 32
#define DATA_BUFF 256

proctype PB()
{
 inport chan pktIn = [IN_BUFF] of { int };
 outport chan pktOut = [OUT_BUFF] of { int };
 outport chan dataBuf = [DATA_BUFF] of { int };
 int tempPkt;
 int state=IDLE;
 do
   ::(state==IDLE)->pktIn?CON_REQ->
    if
      ::(len(dataBuf)<DATA_BUFF/2)->pktOut!CON_ACK; state=CONNECTED;
      ::else->pktOut!CON_REJ;
    fi;
   ::(state==CONNECTED)->
    if
      ::pktIn?DIS_REQ->pktOut!CON_REJ; state=IDLE;
      ::pktIn?DATA->pktIn?tempPkt->
       if
         ::(len(dataBuf)<DATA_BUFF)->dataBuf!tempPkt;
```

```
        ::else->pktOut!CON_REJ; state=IDLE;
      fi;
    fi;
  od;
}
```

Figure 4.2 SiP specification of the connection process at receiving side.

Initially set to the IDLE state, PB acknowledges the connection request from PA only when at least half of its local buffer is empty. Otherwise, it replies with CON_REJ to reject the connection. Once the connection has been established, PB is in CONNECTED state and it forwards every incoming packet to its local buffer. If the incoming rate is much higher than the processing rate and thus the local buffer is exhausted, PB will send PA a disconnection notification, CON_REJ, and interrupt connection immediately. After that, PB returns to the IDLE state.

Although we now have built the two communicating modules of the connection protocol, it only models a simplex connection. Specifically, these two modules only allow establishing a connection from PA side to PB side but not the other direction. However, as explained in Chapter 2, more complicated protocols can always be constructed if we have taken the reusability into account. For example, the middle part of Figure 4.3 shows a design of half-duplex protocol where each side consists of both PA and PB blocks. Since a half-duplex protocol at most allows one connection from one side to the other, each side needs an extra control input to switch between transmitting and receiving modes. The switching control here is similar to the "push-to-talk" button on a talk radio whose position decides the radio to send out or receive from a channel. As shown in the figure, we implemented this mechanism by using a relay and a multiplexer to direct and merge packet streams. For each side, PA block is activated and PB is shut off when the control is on and inversely when it is off. A galaxy icon representing the schematic is shown on the right. It can be reused to build even higher layer schematics such as an N-port half-duplex module.

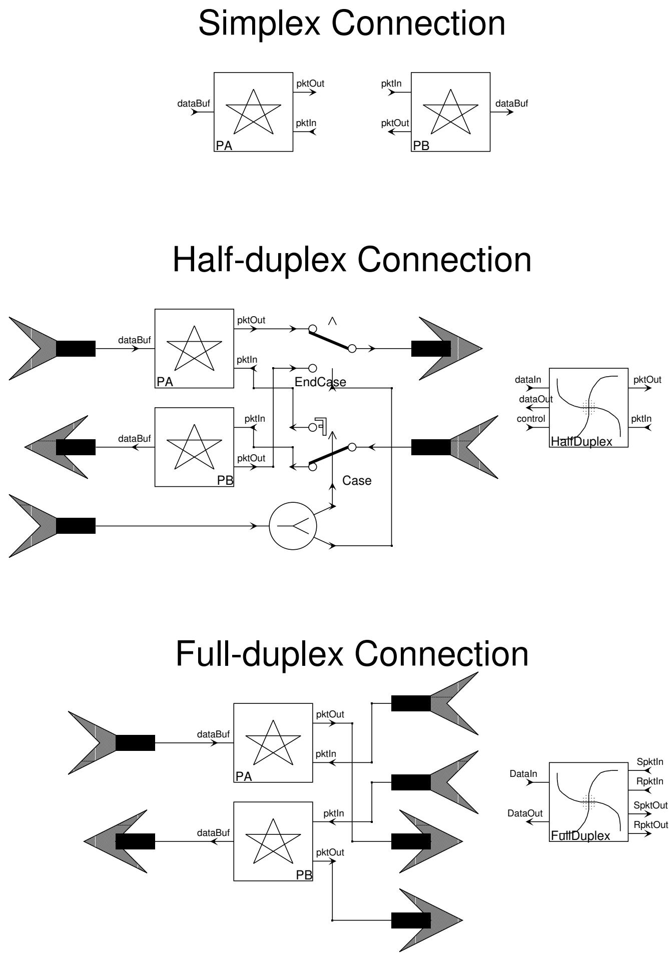



Figure 4.3 Construct duplex connection protocols using simplex blocks.

A full-duplex connection protocol can be similarly constructed as shown in the bottom of Figure 4.3. Note that in this case we do not have an extra control line because now PA and PB blocks are allowed to interact with their counterparts simultaneously. That is, data packets now can propagate in both directions by establishing a two-way connection.

## 4.2 Error Detection and Recovery

In last section, we assume the channel between two distant processes PA and PB is perfect. That is, through that channel packets can always arrive the other side correctly. The real world, however, is not such perfect. Three different types of errors could happen during the delivery: corruption, loss, and out-of-order arrival [27]. The sender or receiver hence has to detect whether some error happened during the transmission, and then either correct it or initiate a retransmission procedure.

Due to electrical interference or thermal noise, bits may be altered at any point of the medium thorough the connection. To recover the corrupted bits, the error control code accompanied with the erroneous packet needs to contain enough information for the correction. The price of this recovery ability is that more bits are required for the error control code and thus less efficient in conveying data [27]. The bit error rate (BER) of the medium and end-to-end latency are two major considerations while making the trade-off between recovery ability and data efficiency. Intuitively, low BER requires less protection bits and small latency afford multiple trials of transmission so that simpler protection techniques are preferred.

Today's wired networks, especially the optical links, suffer from very low BER and moderate latency. Instead of trying to recover corrupted bits, more efficient technique such as cyclic redundancy check (CRC) is widely used in the data-link layer to detect bit corruption. For example, an Ethernet frame carries up to 1,500 bytes of data requiring only a 4-byte CRC code. Besides, BISYNC by IBM, DDCMP by DEC, IMP-IMP used in ARPANET, HDLC, FDDI and ATM all adopt the CRC algorithm [2]. However, this protection code is only for detecting the occurrence of bit corruption but not able for recovery. Once a receiver detects a bit error, it immediately discards the frame and executes a predefined routine to inform the sender that a retransmission of that frame is required.

The elaboration of detection techniques is more like refining an algorithm rather than a protocol design issue. Alternatives such as two-dimension parity and Internet checksum algorithms also try to correlate data bits with much shorter redundant bits. These techniques manipulate the data packets themselves but do not involve in the interaction of distributed processes, which is the core issue of specifying protocols. In fact, while designing a reliable communication protocol, we often assume that an error detection technique has been chosen and concentrate on working out the routines to recover errors. Specifically, we enumerate possible scenarios of errors, and then define corresponding recovery rules to resend corrupted and lost packets and reorder out-of-order packets.

Recall the PA and PB processes in last section, the issue now is to have the packets sent by PA be delivered at PB without corruption, loss or reorder. The usual approach of error recovery is having PB reply PA an acknowledgement packet in response to the received data packets sent by PA. Then PA examines the received acknowledgements to perceive which packets has lost or discarded by PB due to corruption. After that PA can either resend those missing packets or on a batch basis depending on the consideration of complexity and efficiency.

Three features are generally shared in the error recovery protocols. First, each data packet sent by PA includes a sequence number field. Therefore by examining the numbers, PB is able to reorder those out-of-order packets. Second, data packets received by PB is acknowledged by replying PA an acknowledgement packet. This response could be taken with respect to each individual data packet or a block of them. Third, a number is predefined to limit the maximal amount of data packets PA can send without receiving acknowledgement regarding any of them. This upper bound is usually called window size. Error recovery protocols with this feature are hence named sliding-window protocols.

A typical sliding-window protocol works as follows. At sender's side, PA continuously sends data packets containing increasing sequence numbers to PB as long as allowed by the window size. Whenever PA receives acknowledgement, the window is moved ahead of all acknowledged data packets. If the earliest sent packet within the window has not been acknowledged for a predefined timeout duration, PA resends that packet using its original sequence number. At receiver's side, PB replies all correct packets and discards corrupted ones. For those correctly received packets, PB only stores the unacknowledged ones because it recognizes that the rest in fact have already been saved but whose past acknowledgements were lost. Besides, PB uses sequence numbers to store out-of-order packets in correct order.

A practical specification of the sliding-window protocol using SiP is shown below.

```
#define IN_BUFF 32
#define OUT_BUFF 32
#define DATA_BUFF 256
#define SWINSIZE 17

utility {
enCRC; deCRC; Max;
}

proctype PA()
{
 inport chan pktIn = [IN_BUFF] of { int };
 outport chan pktOut = [OUT_BUFF] of { int };
 inport chan dataBuf = [DATA_BUFF] of { int };
 const int PKT_TIMEOUT=10;
 int lar=-1, lps=-1, dataBkup[SWINSIZE];
 int tempPkt, tempSN, tempCRC;
 timer tm[SWINSIZE];
 do
   ::((lar+SWINSIZE > lps) && (len(dataBuf)>0))->
    dataBuf?tempPkt; lps++;  pktOut!tempPkt; pktOut!lps; pktOut!enCRC(tempPkt, lps);
    dataBkup[lps%SWINSIZE]=tempPkt; tm[lps%SWINSIZE]=0;
   ::((lps > lar) && expire(tm[(lar+1)%SWINSIZE], PKT_TIMEOUT))->
    pktOut!dataBkup[(lar+1)%SWINSIZE]; pktOut!lar;
    pktOut!enCRC(dataBkup[(lar+1)%SWINSIZE], lar+1);
    tm[(lar+1)%SWINSIZE]=0;
   ::(len(pktIn)>0)->pktIn?tempPkt; pktIn?tempSN; pktIn?tempCRC;
    if
      ::deCRC(ACK,tempSN,tempCRC)->lar=Max(lar, tempSN);
      ::else->skip;
    fi;
od;
}
#define RWINSIZE 17
#define ACK 255


proctype PB()
{
```

```
inport chan pktIn = [IN_BUFF] of { int };
outport chan pktOut = [OUT_BUFF] of { int };
outport chan dataBuf = [DATA_BUFF] of { int };
bool needAck=0, rcvInd[RWINSIZE];
int npe=0, dataBkup[RWINSIZE], i;
int tempPkt, tempSN, tempCRC;
i=0;
do
  ::(i<RWINSIZE)->rcvInd[i]=0; i++;
  ::else->break;
od;
loop:
 pktIn?tempPkt; pktIn?tempSN; pktIn?tempCRC;
 if
   ::deCRC(tempPkt,tempSN,tempCRC)->
     if
       ::(tempSN < npe)->needAck=1;
       ::((npe <= tempSN) && (tempSN < npe+RWINSIZE))->
         rcvInd[tempSN%RWINSIZE]=1; dataBkup[tempSN%RWINSIZE]=tempPkt;
         do
             ::rcvInd[npe%RWINSIZE]->dataBuf!dataBkup[npe%RWINSIZE];
               rcvInd[npe%RWINSIZE]=0; needAck=1; npe++;
             ::else->break;
         od;
       ::else->skip;
     fi;
     if
       ::needAck->pktOut!ACK; pktOut!npe-1; pktOut!enCRC(ACK, npe-1); needAck=0;
       ::else->skip;
     fi;
   ::else->skip;
 fi;
 goto loop;
}
```

Figure 4.4 SiP specification of a sliding-window protocol (PA for transmitting side; PB for receiving side)..

Variables, constants and auxiliary procedures used in these two processes are defined as follows:

**lar**: last acknowledgement received
**lps**: last packet sent
**npe**: next packet expected
SWINSIZE: sending window size
RWINSIZE: receiving window size

enCRC: encode CRC
deCRC: decode CRC

PA has three major states, sending a packet, acknowledgement timeout, and receiving an acknowledgement packet. The prerequisite to send a packet is that **lps** has to be still within the sending window size and there must exist pending packets to be sent. After sending a packet, PA resets and starts a timer to keep track of the time it has been waiting for that packet. If the timer of the oldest unacknowledged packet, which was sent PKT_TIMEOUT time units ago, expires, PA sends that packet again and resets the timer. After receiving an acknowledgement packet, PA first verifies its correctness by using CRC checking, and moves the sending window ahead of the acknowledged packet index if the verification is positive.

PB will be triggered only when a data packet arrives. After verifying its correctness, PB takes actions with respect to the sequence number of the packet. If the number is smaller than **npe**, PB recognizes that one earlier copy of this packet has been successfully received but all of its acknowledgements have been lost. PB then sends another acknowledgement for this packet again. If the number is equal to or greater than **npe** and less than **npe**+RWINSIZE, the packet is said to be within the receiving window. The data bytes contained in the packet will then be stored but not yet delivered because they could be out-of-order packets. After that, PB examines the receiving indications starting from npe and delivers the corresponding data bytes sequentially until the first negative indication is reached. Finally, an acknowledgement is sent to acknowledge all packets prior the next expected packet. The last case of a correctly received packet is that the packet has a sequence number larger than the upper bound of receiving window. In that case, PB has to discard the packet because it does not have spare buffer to store the data bytes of the packet. As for corrupted packets, PB simply discards all of them.

## 4.3 Flow Control

At the end of last section, we mention that the receiving process PB has to discard packets due to running of out buffer, even though those packets have been correctly received. To avoid wasting transmission bandwidth like this, the sending process PA should control the flow of its outgoing packet stream so that it will not overwhelm PB's handling capability. However, on the other hand, PA should also try to send PB as many packets as possible for maximizing efficiency. The requirement of such trade-off leads to the development of various flow control schemes [48].

The sliding-window protocol discussed above in fact has a very primitive design of flow control. Its sending window size, SWINSIZE, prevent PA sending further packets if the number of unacknowledged packets already reaches the size. This blocking remains until PA receives an acknowledgement for some packet within the sending window.

There are three defects of the sliding-window protocol in terms of controlling the packet flow. First, large SWINSIZE makes the control ineffective due to the rare blocking on PA side. This happens when the round-trip time (RTT) of PA->PB->PA is long and we try to "fill the pipe" to achieve higher efficiency. Second, fixed SWINSIZE disables PA from adapting the sending window size to reflect current situation of PB. Intuitively, one would like to shrink SWINSIZE when PB is very busy and enlarge it when PB is close to idle. Third, in sliding-window protocol, an acknowledgement bundles both the information of confirming reception and allowing further sending, which makes PA less perceivable to the actual status of PB. For example, PB may want to acknowledge some packets but still keep PA blocked because it is currently too busy to accept any new packets. The bundling, however, is unable to differentiate this situation.

For the rest of this section, we will discuss a modified version of the sliding-window protocol that allows PA to change SWINSIZE depending on the frequency of discarding packets [14]. In the modified protocol, the constant SWINSIZE is replaced with a variable **swinsize** whose value ranges from MINSWS to MAXSWS. Depending on the occurrence of packet discard, **swinsize** has the flexibility to be adjusted within that range and which actually tunes the tightness of sending window. Specifically, whenever an acknowledgement timeout happens and PA resends the corresponding packet, **swinsize** is reduced by a factor of 2 if the new value is not less than MINSWIN. On the other hand, every time PA successfully sends THRSWS packets to PB without having to resend any one of them, **swinsize** is incremented by 1 if the new value is not greater than MAXSWS.

The adaptation algorithm of this scheme, though effective, turns out too pessimistic when timeouts happen consecutively. Say the waiting for acknowledgements of packet 3, 4, 5, . . . expires one by one, **swinsize** will be decreasing exponentially. However, we know that most timeouts are not caused by the transmission error which rarely happens in today's wired media. Most of time those missing packets are discarded by the receiver due to insufficient buffer of processing ability. Therefore, it is expected that the syndrome of packet loss appears in a burst fashion. In this case, after resending packet 3 and halving swinsize, PA may not want to decrease **swinsize** again when the acknowledgement of packet 4 also expires later.

One way to work around the problem is to keep track of an "unlikely window" that specifies a continuous list of possibly discarded packets. After PA resending a packet, it checks whether the sequence of the packet falls in the unlikely window. If yes, **swinsize** remains unchanged; otherwise it is halved. The updating of the unlikely window is done whenever the **swinsize** is reduced.

Figure 4.5 shows the PA process accomplished the flow control scheme described above. Since the controlling is totally done by PA, in this scheme PB is the same as the one in last section.

```
utility {
enCRC; deCRC; Max; Min;
}

#define SWINSIZE 17
#define MINSWS 2
#define MAXSWS 17
#define THRSWS 2
```

```
proctype PA()
{
 inport chan pktIn = [IN_BUFF] of { int };
 outport chan pktOut = [OUT_BUFF] of { int };
 inport chan dataBuf = [DATA_BUFF] of { int };
 const int PKT_TIMEOUT=10;
 int lar=-1, lps=-1, uws=-1, uwe=-1, cap=0, swinsize=(MINSWS+MAXSWS)/2;
 int tempPkt, tempSN, tempCRC, dataBkup[SWINSIZE];
 timer tm[SWINSIZE];
 do
   ::((lar+swinsize > lps) && (len(dataBuf)>0))->dataBuf?tempPkt;
    lps++; pktOut!tempPkt; pktOut!lps; pktOut!enCRC(tempPkt, lps);
    dataBkup[lps%SWINSIZE]=tempPkt; tm[lps%SWINSIZE]=0;
   ::((lps > lar) && expire(tm[(lar+1)%SWINSIZE], PKT_TIMEOUT))->
    pktOut!dataBkup[(lar+1)%SWINSIZE]; pktOut!lar;
    pktOut!enCRC(dataBkup[(lar+1)%SWINSIZE], lar+1); tm[(lar+1)%SWINSIZE]=0;
    if
      ::((uws<=lar+1) && (lar+1<=uwe))->skip;
      ::else->swinsize=Max(swinsize/2, MINSWS); uwe=lps;
    fi;
    uws=lar+2; cap=0;
   ::(len(pktIn)>0)->pktIn?tempPkt; pktIn?tempSN; pktIn?tempCRC;
    if
      ::deCRC(ACK,tempSN,tempCRC)->
        cap=cap+Max(tempSN-lar, 0); lar=Max(lar, tempSN);
         if
           ::(cap>=THRSWS)->swinsize=Min(swinsize+1, MAXSWS);
             cap=cap-THRSWS;
           ::else->skip;
         fi;
      ::else->skip;
    fi;
 od;
}
```

Figure 4.5 SiP specification of a modified sliding-window protocol for flow control at transmitting side.

Variables and constants used in above specification are defined as follows.

**swinsize**: sending window size
**cap**: consecutively acknowledged packets
**uws**: unlikely window start
**uwe**: unlikely window end
MAXSWS: maximum sending window size

MINSWS: minimum sending window size
THRSWS: threshold of consecutive acknowledgements to increase swinsize

Note that PA now has an adaptive window size, **swinsize**, which is updated whenever timeout happens or **cap**>=THRSWS. These two cases in fact can be regarded as the implicit and explicit status feedback from PB. Beside, the two bounds of unlikely window, **uws** and **uwe**, are updated only after processed a time-out event. Being such case, **cap** is reset because the earliest unacknowledged packet has been assumed discarded and thus the acknowledgement is no longer continuous.

## 4.4 Routing

Generally, networks are constructed to allow distributed end-users to convey information one another. Such data interchanging would be trivial if the intended communicating partner is always an adjacent node of the sender. In fact, for above sections of this chapter, the protocols we mentioned will only work in this trivial way if no effort is made to implement a forwarding mechanism between the two communicating entities. This is where routing protocols enter the picture to form a full circle of end-to-end connection.

The fundamental idea of routing is to attach an address tag of the destination node to the data packet and let the intermediate nodes figure out a way to forward the packet. In a packet switching network, the routing process is done in a hop-by-hop fashion, i.e., each router only decides which neighboring router connected to it would probably be the best (fast and cheap) next hop in terms of forwarding a packet to its destination. A direct question arisen from this approach is "Which neighbor should a router choose to forward a packet?". The answer is the prosperity of current designs of routing protocols.

A common necessity of routing processes is to establish and maintain a routing table. The table lists the current best next hop from the router to all of its reachable nodes. Note that information may have to be updated as the status of the links belonged to the network changes. A router hence has to keep gathering status reports from other nodes and recalculating those best next hops periodically [48]. Two common problems associated with routing tables are:

1. A routing table will not be scalable if it enumerates all reachable nodes in the table. Some simple designs use an entry for each possible destination in the network. Doing this requires a table to be large enough to accommodate the number of nodes in the network and that is usually not feasible in consideration of memory requirement.

2. A routing table needs to store up-to-date information to reflect any changes in the network topology and in the connection status of links. The first consideration of the changes makes a router more robust to tolerate the failure of other nodes and to support the mobility of end nodes. The second leads to the design of an intelligent router that avoids congested links while routing a packet.

Various routing protocols, such as hierarchical, random, distributed, backward learning, source, and mobile routing [14] have been proposed to solve above problems. Because most of these designs have the complexity and subtlety beyond the scope of this thesis, we will only discuss a simplified hybrid routing protocol in this section. Nevertheless, it does partially solve the two problems and provides an overview of the issues while designing a routing protocol.

Our hybrid routing protocol (HRP) mixes part of hierarchical, distributed, and source routing protocols. It is hierarchical because the whole network is partitioned into several subnetworks and each subnetwork contains several hosts. The address representation therefore has 2 fields for two levels of resolution. We let the routing process only consult the information at subnetwork layer and thus each entry in the routing table now stands for a subnetwork instead of an individual host. Doing this remarkably saves the memory requirement of the routing table. Our routing protocol also allows a router to inform others about its connection status with adjacent nodes, which is the sole feature of the distributed routing. Finally, the source routing assumes that all hosts have current and complete information about the network topology, so do the routers (but not end-hosts) in our assumption. In the source routing, when a packet is generated, the process calculates the best route for the packet to reach its destination and attaches the route to the packet. When a node in that route receives the packet, it simply looks its successor in the route and forwards the packet. This approach would be infeasible if the route information is too long to be included in a packet. In our protocol, the routing works on a hop-by-hop basis at each router although the complete information is available to computer the whole route. One advantage of the hop-by-hop routing is that no route is attached to a packet and thus less overhead is introduced.

Figure 4.6 shows the design of our hybrid routing protocol using SiP.

```
utility{
netproc;
}

#define IN_BUFF 32
#define OUT_BUFF 32
#define DATA_BUFF 256

#define DATA_TYPE 10
#define RTT_TEST 11
#define RTT_REPLY 12

proctype HRP()
{
 inport chan pktIn = [IN_BUFF] of { int };
 outport chan pktOut1 = [OUT_BUFF] of { int };
 outport chan pktOut2 = [OUT_BUFF] of { int };
 inport chan InDataBuf = [DATA_BUFF] of { int };
 outport chan OutDataBuf = [DATA_BUFF] of { int };
 const byte SUBNET_ID=1;
 const byte HOST_ID=1;
 int LONG_ID;
 int type,srcid,dstid,data;
 int nbr, nextid, cost;
 timer upd, rtt[2];
 LONG_ID=SUBNET_ID*256+HOST_ID;
 upd=0;
```

```
 loop:
  do
    ::expire(upd, 200)->
       pktOut1!RTT_TEST; pktOut1!SUBNET_ID;
       pktOut1!netproc(1, SUBNET_ID, 1); pktOut1!0;
       pktOut1!RTT_TEST; pktOut1!SUBNET_ID;
       pktOut1!netproc(1, SUBNET_ID, 2); pktOut1!0;
       upd=0; rtt[0]=0; rtt[1]=0;
    ::(len(InDataBuf) > 0)->InDataBuf?dstid; InDataBuf?data;
       type=DATA_TYPE; srcid=LONG_ID; break;
    ::(len(pktIn) > 0)->pktIn?type; pktIn?srcid; pktIn?dstid; pktIn?data; break;
    ::else->return(0);
  od;
  if
    ::(type==RTT_REPLY)->nbr=netproc(2, SUBNET_ID, srcid);
      cost=rtt[nbr-1]/2; netproc(3, SUBNET_ID, srcid, cost);
    ::(type==RTT_TEST)->
      if
        ::(netproc(2, SUBNET_ID, srcid)==1)->
          pktOut1!RTT_REPLY; pktOut1!SUBNET_ID; pktOut1!srcid; pktOut1!0;
        ::(netproc(2, SUBNET_ID, srcid)==2)->
          pktOut2!RTT_REPLY; pktOut2!SUBNET_ID; pktOut2!srcid; pktOut2!0;
      fi;
    ::(type==DATA_TYPE)->
      if
        ::(dstid/256==SUBNET_ID)->OutDataBuf!data;
        ::else->nextid=netproc(4, SUBNET_ID, dstid/256);
          if
            ::(netproc(2, SUBNET_ID, nextid)==1)->
              pktOut1!DATA_TYPE; pktOut1!srcid; pktOut1!dstid; pktOut1!data;
            ::(netproc(2, SUBNET_ID, nextid)==2)->
              pktOut2!DATA_TYPE; pktOut2!srcid; pktOut2!dstid; pktOut2!data;
            ::else->skip;
          fi;
      fi;
    ::else->skip;
 fi;
 goto loop;
}

int netproc(int* args)
{
  #define ROUTER_NUM 4
  #define id2idx(ID) (ID-1)
  static int net[ROUTER_NUM][ROUTER_NUM]={{1, 2, 3, -1}, {2, 1, 4, -1},
                                          {3, 1, 4, -1}, {4, 2, 3, -1}};
```

```
  static int cost[ROUTER_NUM][ROUTER_NUM]={{0, 90, 50, -1}, {90, 0, -1, 20},
                                          {50, -1, 0, 30}, {-1, 20, 30, 0}};
  int pathcost[ROUTER_NUM], tmpcost, src_sn, dst_sn, bestnext, node2, done=0, i, n;
  switch (args[0]) {
   case 1: return(net[id2idx(args[1])][args[2]]);
   case 2: for(i=1; i<ROUTER_NUM; i++)
              if (net[id2idx(args[1])][i]==args[2]) return i;
   case 3: cost[id2idx(args[1])][id2idx(args[2])]=args[3];
             cost[id2idx(args[2])][id2idx(args[1])]=args[3]; return 1;
   case 4: src_sn=args[1]-1; dst_sn=args[2]-1; bestnext=dst_sn;
    for(i=0;i<ROUTER_NUM;i++)
     pathcost[i]=(cost[src_sn][i]>-1)? cost[src_sn][i]:MAXINT;
    while (!done) {
     done=1; n=1;
     while((n<ROUTER_NUM)&&(net[src_sn][n]>0)) {
      node2=id2idx(net[src_sn][n]);
      if (cost[node2][dst_sn]>-1) {
       tmpcost=pathcost[node2]+cost[node2][dst_sn];
       if (tmpcost<pathcost[dst_sn]) {
        pathcost[dst_sn]=tmpcost;
        bestnext=node2;
        done=0;
       }
      }
      n++;
     }
    }
    return(net[bestnext][0]);
  }
}
```

Figure 4.6 SiP specification of the Hybrid Routing Protocol with its auxiliary C procedure netproc().

HRP models a router having multiple input ports and 2 output ports. Normally, it takes DATA_TYPE packets and use Bellman-Ford algorithm to decide to which neighbor it should forward the packet for minimizing the latency of the complete route. The implementation of the Bellman-Ford algorithm [2] is shown in the above auxiliary C procedure. Besides, in order to gather up-to-date status of links used by the algorithm, a router generates extra packets, with type RTT_TEST, and sends them to its neighbors to request a RTT measurement testing every 200 time units. Whenever receives a RTT_TEST type packet, a router immediately sends a RTT_REPLY type packet back to the one initiated the testing. Therefore, on receiving the RTT_REPLY type packet, the testing initiator is able to obtain the RTT and update the current "cost" of a specific link with value RTT/2. Note in HRP we assume this update is made globally and simultaneously to all routers for simplicity. However, we know that actually this has to be done using some broadcasting mechanism.

The auxiliary C procedure netproc implements more than one function. The first argument serves as the index of intended function and the rest arguments have different meanings under different functions. A detailed description is given below.

| Index | Arguments | Return | Function Description |
|---|---|---|---|
| 1 | host ID,<br>neighbor index | neighbor ID | Return the ID of a neighbor by giving its index. |
| 2 | host ID,<br>neighbor ID | neighbor index | Return the index of a neighbor by giving its ID. |
| 3 | host ID,<br>neighbor ID, cost | 1 | Update the cost of the link between two giving hosts. |
| 4 | source subnet ID,<br>destination subnet ID | best next ID | Decide the best next hop and return its ID. |

## 4.5 Multiple Access

The protocols considered so far are designed for point-to-point communication links, which assume that on a specific medium there is only one host sending signal at a time. Under such assumption, a receiver needs only consider the transmitted signal from some peer and noise on the link, but not signals from other peers which shared the same medium.

However, there are many widely used communication media such as radio broadcast, satellite, and multitap bus [28], which may have two or more hosts are sending out signals simultaneously. In this case, different data streams will interfere with one another if no measures are taken to regulate the use of the medium. Such a collision can be avoided if only one host is permitted to use the medium for transmission at a time. Two usually adopted approaches are used to implement this feature: token and carrier sense. The single token in a network allows only one host transmitting at a certain moment, and all other hosts have to wait until they hold the token. Two popular protocols based on this approach are Token Bus and Token Ring protocols [27].

Carrier sense, as implied by its name, requires a host listening to the medium before it can send out signals. As long as the medium is in use, existing carrier signal, the rest hosts refrain from transmitting and remain waiting. One disadvantage of this approach is that a collision may still occur when two or more hosts start sending at almost the same time such that they all thought the medium is cleared. The occurrence of a collision requires all receiving hosts throwing away whatever they have received recently. In addition, all sending hosts have to stop transmitting and, after some time, retransmit the same message. However, more collisions may happen following the same scenario again and again [24]. A protocol based on this approach is called a CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol. In this section we will look at a simple protocol using the CSMA/CD technique, which is actually is simplified version of the most popular local area network (LAN) protocol, Ethernet protocol. Two key features of CSMA/CD are preserved in our Simple Ethernet Protocol (SEP): carrier sense (to make sure the medium is free) and collision detection (to see if any other host is also transmitting). Data frame and jam frame are the only two types of frames defined in this protocol. Under normal condition, a host broadcasts data frames onto the medium for conveying information. The jam frame will be transmitted when a collision has been detected, which is meant to garble all frames on the medium so that all hosts will be aware of the occurrence of the collision.

The SiP specification of the transmitting and receiving processes of SEP are shown in Figure 4.7(a) and Figure Figure 4.7(b) respectively. In the protocols, we assume that a frame is transmitted byte by byte at a speed of one byte per time unit. The collision detection thus can be done between the transmission of each two consecutive data bytes. The CRC computation procedures have been modified to operate in a byte-by-byte fashion due to the same reason. Their first argument has value 0, 1 and 2 to differentiate the cases of starting, intermediate and end of a frame. We also define an end-to-end propagation delay, PROP_DELAY, which results in the possible unawareness of far-end transmission by using carrier sense. The share medium is modeled by a broadcasting channel BUS which is declared as a wireless port.

```
utility {
 enCRC; deCRC; randNum; Max;
}

#define DATA_BUFF 256
#define MAX_BYTES 1024
#define PL_BYTES 64
#define JAM 127

proctype XMT()
{
 inport chan dataBuf = [DATA_BUFF] of { int };
 wireless chan BUS = [MAX_BYTES] of {int};
 const byte HOST_ID=1;
 const int PROP_DELAY=5;
 const int RAND_WAIT=32;
 bool rexmt=0;
 int i, dstid, data, bidx, bmax=0, dataBkup[PL_BYTES];
 car_sen:
 do
   ::(rexmt || (len(dataBuf)>0))->bidx=0;
     if
       ::rexmt->skip;
       ::else->dataBuf?dstid;
     fi;
     do
       ::(len(BUS)>randNum(PROP_DELAY)+2)-> delay(1+randNum(RAND_WAIT));
       ::else->break;
    od;
```

```
   BUS!dstid; BUS!HOST_ID; enCRC(0,dstid); enCRC(1, HOST_ID);
   do
    ::(bidx<PL_BYTES)->
     if
      ::(rexmt && (bidx<bmax))->data=dataBkup[bidx];
      ::else->dataBuf?data; dataBkup[bidx]=data;
     fi;
     BUS!data; enCRC(1, data); bidx++; delay(1);
     if
      ::(len(BUS)>bidx+2)->i=0;
       do
         ::(i<PROP_DELAY)->BUS!JAM; delay(1);
         ::else->break;
       od;
       delay(PROP_DELAY+randNum(RAND_WAIT));
       rexmt=1; bmax=Max(bmax,bidx); goto car_sen;
      ::else->skip;
     fi;
    ::else->BUS!enCRC(2, 0); rexmt=0; bmax=0; delay(1); goto car_sen;
   od;
  ::else->return(0);
 od;
}
```

Figure 4.7(a) SiP specification of the transmitting process of the SEP.

```
proctype RCV()
{
 outport chan dataBuf = [DATA_BUFF] of { int };
 wireless chan BUS = [MAX_BYTES] of {int};
 const byte HOST_ID=1;
 int i, srcid, temp, bidx, dataBkup[PL_BYTES];
 wait_frm:
 BUS?HOST_ID->BUS?srcid;
 deCRC(0, HOST_ID); deCRC(1, srcid); bidx=0;
 do
   ::(bidx<PL_BYTES)->(len(BUS)>0)->
    if
     ::BUS?[JAM]->delay(PROP_DELAY);
      do
        ::(len(BUS)>0)->BUS?temp;
        ::else->break;
      od;
      do
        ::(len(BUS)==0)->return(0);
        ::((len(BUS)>0) && (BUS?[JAM]))->BUS?temp;
```

```
            ::else->goto wait_frm;
          od;
        ::else->BUS?temp; dataBkup[bidx]=temp;
            deCRC(1, temp); bidx++;
      fi;
    ::else->BUS?temp;
      if
        ::deCRC(2, temp)->i=0; dataBuf!srcid;
          do
            ::(i<PL_BYTES)->dataBuf!dataBkup[i];
            ::else->break;
          od;
        ::else->skip;
      fi;
      goto wait_frm;
 od;
}
```

Figure 4.7(b) SiP specification of the receiving process of SEP.

In the transmitting process, XMT, dataBuf is the service access point to the higher layer protocol, Logical Link Control (LLC) sub-layer protocol, for inputting data frames. The buffer dataBkup has a size of PL_BYTES bytes to store the payload of the currently conducting frame. During the carrier sense phase, if XMT has a frame to send but the medium is in use, it waits a random duration of time equiprobably between 1 and RAND_WAIT. XMT sends JAM signal for PROP_DELAY time units when it is transmitting data bytes and detects that extra bytes from other host(s) have been inserting into BUS. Indices **bidx** and **bmax** are used to indicate the position of next byte to be transmitted and the position of the byte where a collision was detected in last transmitting trial. Flag **rexmt** indicates whether if the transmitting process is in the retransmission phase.

As for the receiving process RCV, dataBuf is the SAP to the LLC sub-layer where the correctly received frames are delivered. Normally RCV receives and stores data bytes in buffer dataBkup for later CRC verification. When a JAM signal is detected, RCV discards all received bytes of the currently receiving frame and removes all the following corrupted data bytes and JAM signal until the medium is cleared again.

# 5

# Application Example

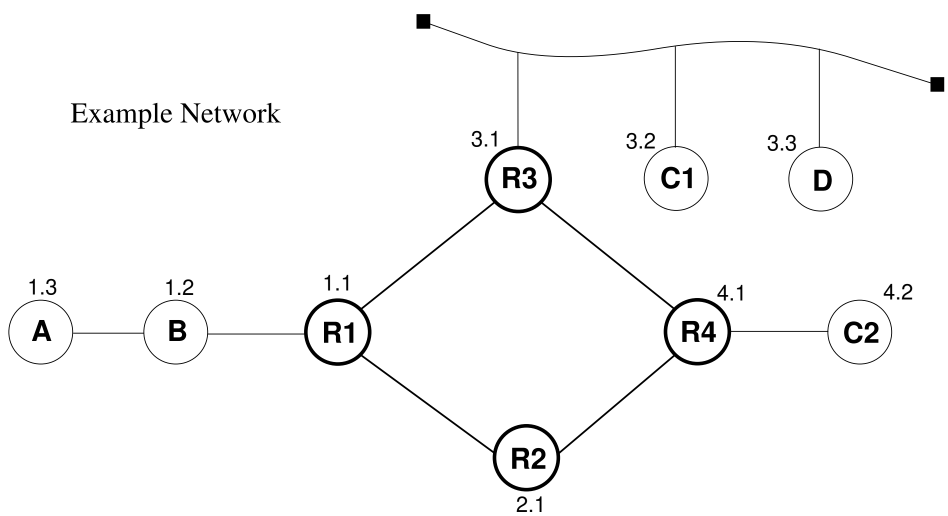



Figure 5.1 Graph diagram of the example network used in this chapter.

In order to validate our design of protocol elements discussed in last chapter, we construct an example network system by reusing those modules to model the communication protocols running on the hosts and routers in the network. Figure 5.1 depicts a connected graph diagram showing the nodes and the links of the network. The nodes are numbered in the format of hierarchical address, which is represented as **n1.n2** to denote the sub-network index **n1** and the host index **n2**. Aliases **R1**, **R2**, **R3**, and **R4** are four routers located in sub-networks 1, 2, 3, and 4 respectively. Except sub-network 3, where all nodes share the same medium, all nodes in other sub-networks are connected by point-to-point links. In the rest of this chapter, we will demonstrate how to build such a network system using SiP and perform simulation of a streaming video application running on it.

Figure 5.2 shows a SiP schematic that models the network in Figure 5.1. Recall the concept we stated in Chapter 2, that each network node in SiP represents a single protocol module or a group of them. In other words, we perceive the activities of a network as the interactions among protocol modules. Therefore, the blocks with names ROUTER and HOST shown in Figure 5.2 are all protocol modules instead of physical hardware entities. Starting from the left, HOST_A (alias **A** in Figure 5.1) is a video encoder that generates and transmits a variable-bit-rate packet stream to a multicast server [45], HOST_B (alias **B**). The server then duplicates and broadcasts the stream to two designated client hosts, HOST_C1 (alias **C1**) and HOST_C2 (alias **C2**). For not overwhelming server **B**, the transmission from **A** to **B** can not start until **B** confirms the connection request from **A** using the connection protocol we discussed in Section 4.1. In addition, **A** and **B** follow a flow control protocol to regulate their traffic using the sliding-window technique introduced in Section 4.3. Figure 5.3 gives a detailed views of the structures of protocols inside **A** and **B**. **A** contains a video encoder (VENC), an address attaching process (AttAddr), a flow control and a connection protocol of transmitting side (SWPA and PA). **B** has a connection and a flow control protocol of receiving side (PB and SWPB), a multicasting process (Distrb), and an SWPA.

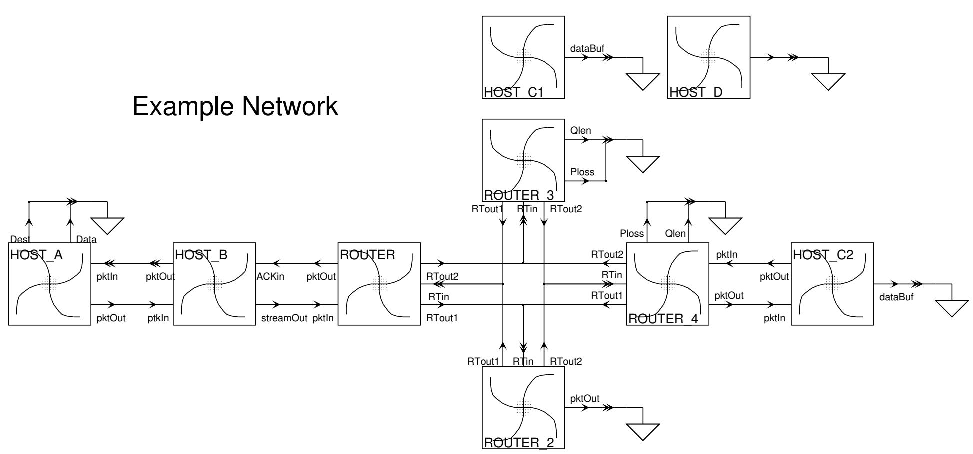



Figure 5.2 A SiP schematic modeling the example network in Figure 5.1.

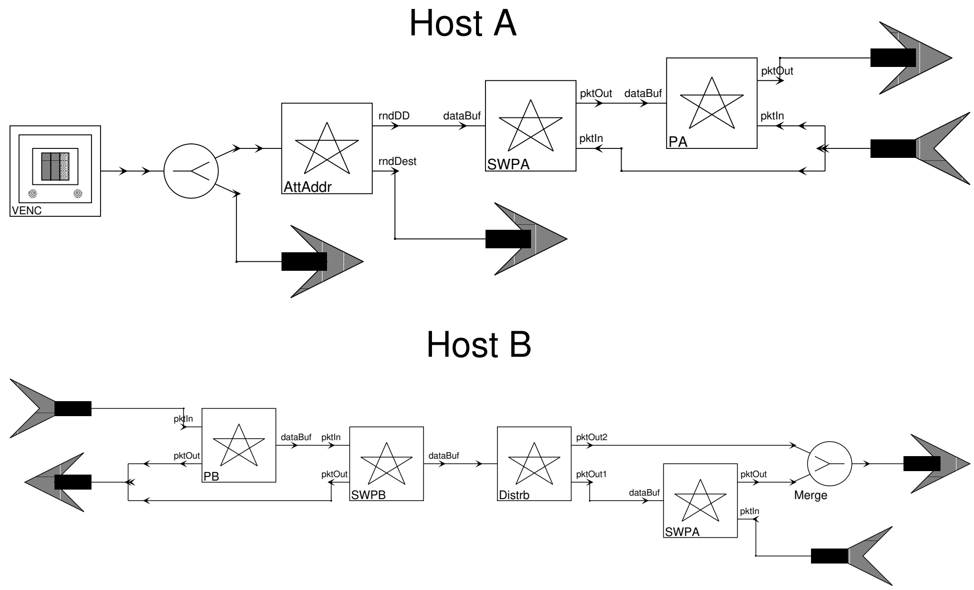



Figure 5.3 The internal structures of galaxies HOST_A and HOST_B.

Router **R1** takes the both broadcasting streams from **B** and route them to their designated destination sub-networks using the HRP routing protocol explained in Section 4.4. Our initial parameters make **R1** choose **R3** to direct both the packets for **C1** and **C2**. This turns out congesting **R3** quickly and slowing down both streams to **C1** and **C2**. Fortunately, HRP updates the routing table periodically and soon figures out that, from **R1**, **R2** may be a better choice to deliver packets to **C2**. After receiving packets from **R1**, **R2** realizes the destination of those packets is sub-network 4 and thus it redirects them to **R4**. The simulation result of this feature is given in Figure 5.4. As one can see, the shooting packet flow via **R3** turns flatter after **R2** started to share the traffic at time 240.

To make the simulation more informative, we build flow control capability in **R4** but not **R3** to evaluate the importance of traffic regulation in a connectionless network. Figure 5.5 shows the internal structure of **R4**, which consists of a routing protocol (HRP), a finite-length queue (FINITE_Q), a flow control protocol of receiving side (SWPB), and a selective acknowledgement protocol of transmitting side (SAPA, which is a slight variation of SWPA). The SWPB module here will interact with the SWPA in host **B** to adaptively adjust the size of sending

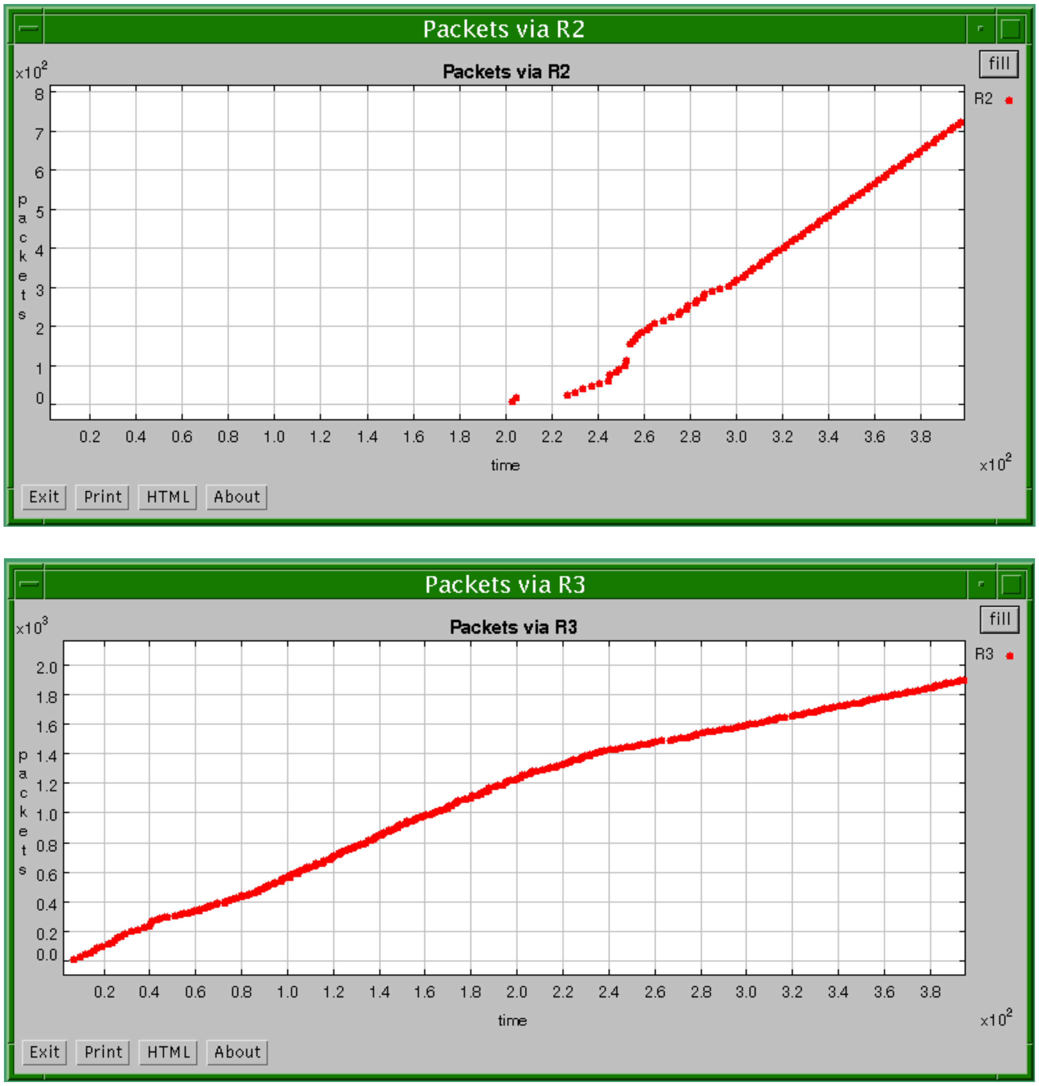



Figure 5.4 Simulation result shows that using adaptive routing table optimization can resolve traffic congestion at routers.

window and it hence mitigates bursting traffic between them. As for **R3**, no effort is made to deal with a variable-bit-rate packet flow and thus all processed packets will be discarded when its output buffer is filled. Figure 5.6(a) shows the simulation result of the output queue lengths of **R3** (no-SW, the dots) and **R4** (SW, the crosses). We set both their maximum queue sizes as 18, so **R3** discards packets whenever its queue length exceeds 18. **R4** has a zigzag shape of queue length

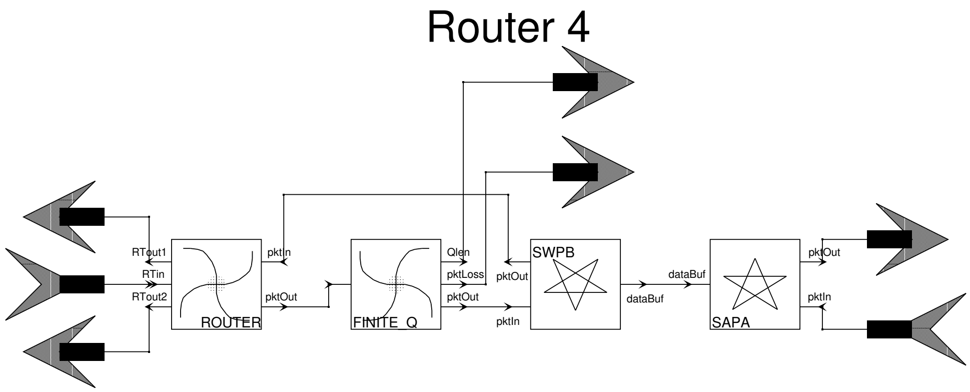
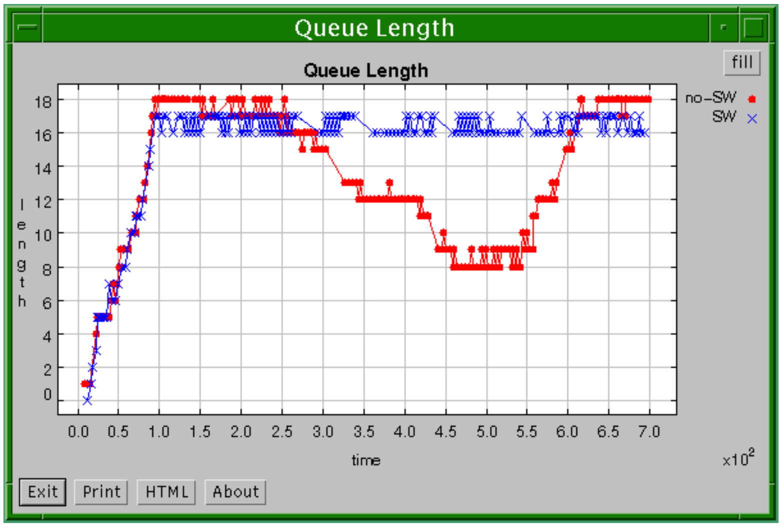



Figure 5.5 The internal structure of the galaxy ROUTER_4.

around the maximum size of its queue because, once its queue gets too long, SWPA cuts down its sending window size or even stops transmission to let **R4** digest its queued packets. And then, after receiving more acknowledgements from the SWPB in **R4**, SWPA enlarges its sending window size gradually and that leads to the rising of the queue length of **R4** again. Since we set the maximum sending window size as 17, one can see from the figure that the queue length of **R4** stop growing at that number. Therefore, R4 is able to guarantee that no packet would be discarded due to an output buffer overflow. Figure 5.6(b) shows the cumulative number of packets discarded at **R3** and **R4**. Not surprisingly, only **R3** suffers from buffer overflows.

Note that so far we have used the SWPA/SWPB pair many times. This is the major advantage of using SiP to model network protocols because reusing protocol modules remarkably reduces the burden in specifying the same or similar protocols repeatedly. Moreover, since Ptolemy follows object-oriented paradigm, all replicated modules of a specific protocol are actually different objects derived from the same class. They are the same protocol, but they can be given different parameters and then evolve independently. The encapsulation property of object-oriented design also assures the state of these sibling modules not being garbled one another.



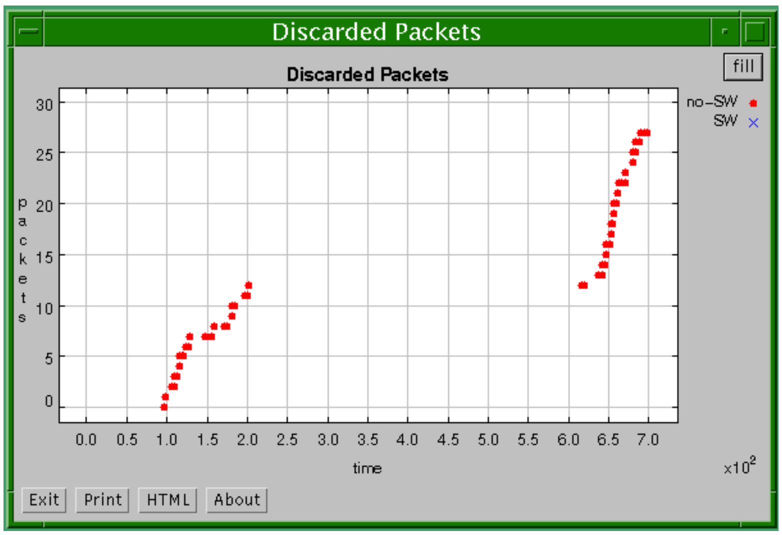




Figure 5.6 (a)Upper: Simulation result shows that using flow control protocol can constrain the queue length within maximum size. (b)Lower: Not using flow control protocol results in queue overflows and thus some packets have been discarded.

Return to our discussion of packet forwarding, which has been described all the way from the video encoder to the destination sub-network of the packet. We now consider the scenario of delivering a packet from a router to the destined host within the same sub-network. Since **C2** has a dedicated link directly connecting it with **R4**, we simply build a selective acknowledgement protocol of receiving side (SAPB) in **C2** to coordinate its counterpart SAPA in **R4** to accomplish the communication. Unlike the above simple point-to-point communication, sub-network 3 needs a more elaborated protocol to deal with its multi-tap bus. As shown in Figure 5.1, router **R3** shares the same medium with other two peers, including **C1**. We use the multiple access protocol SEP, a CSMA/CD protocol introduced in Section 4.5, to implement the communication process of all peers on the bus. Figure 5.7 shows the internal view of SEP, which consists of a transmitting and a receiving process (SEPXMT and SEPRCV). Note that in our design, there is no link among all the SEPXMT and SEPRCV processes in all peers on the bus. This because we adopt "wireless" I/O ports while specifying the SEP protocol. The broadcasting nature of these ports appropriately imitates the topology of shared medium and also comparatively simplifies the wiring. Following the SEP protocol, **R3** finally delivers the packets to **C1** via the shared bus.

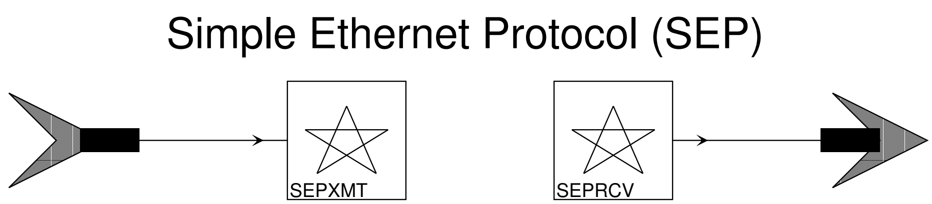



Figure 5.7 The internal structure of galaxy SEP.

The last issue to consider in this simulation is, *why not using flow control protocol everywhere so that no entity would ever discard a packet?* This answer is, not all application can afford long latency during transmission [44]. For direct-immediate applications, such as videoconferencing, Internet phone, whiteboard, talk, etc., long delay between continuous or consecutive information is not tolerable. Packets that successfully arrive their destinations with old time stamps will be useless. As a result, the flow control protocol though effectively smoothes the traffic, the latency introduced by it leads to packet discard, too. Figure 5.8 gives a simulation result that serves as a good example to account for this phenomenon. Each symbol (dot or cross) in the figure indicates an event of receiving a packet. The vertical axis in the figure represents the sequence number of a received packet and the horizontal axis gives the time when the packet was received. Obviously, from the simulation result, using flow control (cross) suffers longer latency, although it is free from packet loss. On the other hand, without using flow control (dot), one can receive packets earlier at the price of discarding packets. Nevertheless, one can always adopts some error (loss) concealment algorithm to enhance the reconstruction quality of the information. This latency consideration also leads to the general adoption of UDP (User Datagram Protocol, which does not includes flow control mechanism) for delivering real-time and delay-sensitive information in the Internet realm. As for applications carrying deferrable information, such as FTP (File Transfer Protocol), Web browser, etc., the TCP (Transmission Control Protocol, which has a built-in sliding window protocol for flow control) is widely used to reduce the impact to the Internet while conducting busty packet streams.

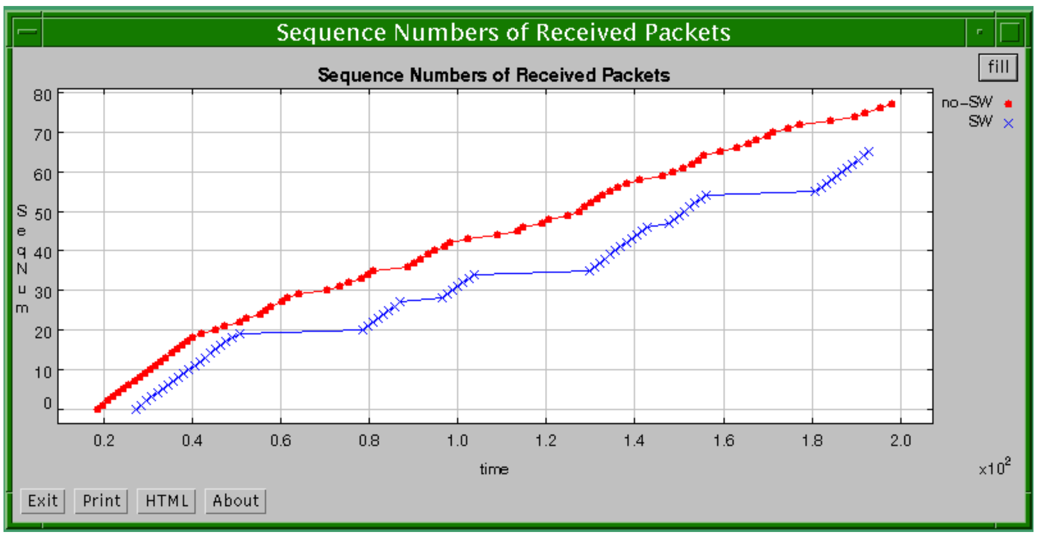



Figure 5.8 Simulation result shows that using flow control protocol could lead to longer lantency due to buffering bursty traffic.

# 6

# Conclusions

To support communication protocols modeling in a system-level design, in this report we have proposed a hybrid model of computation to allow mixing protocol modules with other subsystems. By embedding CSP in FSM and DE, we found a protocol can be succinctly specified and effectively simulated in a system context. The resulting architecture using such integration of domains has been investigated to clarify and define its semantics of concurrency and synchrony.

Base on the proposal, we have prototyped a supporting software infrastructure, SiP, by leveraging on two existing tools, SPIN and Ptolemy. The consideration of combining these two tools originates from the fact that SPIN is designed for protocol specification and Ptolemy supports heterogeneity in system-level modeling. We examine the internal data structure of SPIN and elaborate a specific actor class in Ptolemy to accomplish data sharing and simulation scheduling. The resulting software implementation not only enriches the expressiveness of the input language of SPIN, PROMELA, in temporal statements, it also lightens a niche in Ptolemy to accommodate an auxiliary co-simulation tool.

The testing of SiP starts from the attempt at specifying several fundamental communication protocols such as connection, error detection and recovery, flow control, routing, and multiple access. Because the supporting commands of SiP well cover the necessary expressions in protocol specification, we efficiently built a reusable module for each of these protocols. In addition, we also examine the extension of these modules by mixing them with other Ptolemy actors or external

C language subroutines. An early observation of the reusability of the tool has also been identified, when we were building a duplex connection module using simplex ones and coupling the modules originally designed for error recovery and flow control respectively.

To evaluation the capability of SiP in modeling a complete network system, we construct an example network on which a streaming video application multicasts a packet flow to two remote client hosts. By reusing all protocol elements we mentioned in last paragraph, with very little extra effort we finish the modeling of the system. For this example, it proves the reusability of SiP does remarkably reduce the burden in specifying similar protocol modules repeatedly. As for system-level simulation, many interesting results also have been discussed using this example. We have observed how the fluctuation of queue length and transmission latency affect the behavior of flow control, connection, and routing protocols. And, how the parameters and adaptation schemes of these protocols impact the quality of services over a network.

One open issue of SiP is, to exploit the formal verification capability of SPIN and provide a model checking [46] tool using Ptolemy's graphical user interface. In [42], a compiler that translates Statecharts into PROMELA has been proposed. It is reported to facilitate the modeling and performing partial order reduction [31][32] of a large number of reactive modules by using Statecharts plus SPIN. Actually, we believe that by leveraging on the object-oriented kernel and the well-developed code generation domain in Ptolemy, SiP should be able to provide similar or even more superior features.

# Appendix A: SiP v1.2 User's Manual

## A.1 Introduction

SiP (SPIN in Ptolemy) is a system-level protocol modeling tool developed at University of California at Berkeley. It relies on a translator **ppl2pl** and a SiP kernel to cooperate with the Ptolemy environment. The translator converts the input language to SiP, called Ptolemy-supported PROMELA Language or **ppl**, into the description language of Ptolemy Stars, called Ptolemy Language or **pl**. The automatically generated **pl** code can then be used to build an agent Star in Ptolemy to make a connection with its original **ppl** code. The SiP kernel is bulit into Ptolemy environment to accomplish a Ptolemy-SPIN co-simulation involving both agent Stars of protocol modules and built-in Ptolemy Blocks (Stars and Galaxies). Figure A.1 illustrates the two phases of system specification while using SiP.

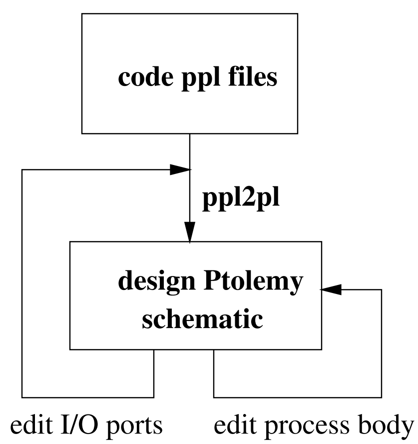



Figure A.1 SiP's two phases of system specification.

The first phase is to code protocol modules in **ppl**, and the second phase is to design a system schematic in Ptolemy using the agent Stars of these modules and Ptolemy's built-in Blocks. Since agent Stars are regualr Stars in Ptolemy DE domain, the second phase is almost the same as the usual way to construct a Ptolemy schematic. The detailed explanation of how to construct a schematic in Ptolemy environment can be found in the Volume I of Almagest, the Ptolemy's User's Manual (ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/www/papers/almagest/user.html). Also shown in the above figure, one minor difference in the second phase is the extra step to execute **ppl2pl** if the declaration of ports in a protocol module has been changed. This assures the consistency of an agent Star and its corresponding protocol module. Note that if the change was the process body of the protocol module, no addition step needs to be taken because the agent Star contains no information about the process body.

This manual will focus on the first phase, the design of protocol modules. Specifically, we will introduce the syntax and semantics of all constructs of **ppl** in the Section A.2 and demonstrate an example to walk through the two design phases in Section A.3.

## A.2 The Elements of ppl

In SiP, every leaf process of a protocol is coded in **ppl**. The **ppl** file should contain exact one process and have a file name as same as its process name. Generally, a **ppl** file is organized as follows.

```
/* This file, PP.ppl, shows the basic structure. */
utility {
func1;
private:
func2;
}

#define QSIZE 10

proctype PP() {
 inport int IN;
 outport int OUT;
 const int WSIZE=5;
 < statements; >
}
```

In the rest of this section, we will give an introduction to the use of **ppl** for specifying protocol processes. While designing a process, one principle a user should keep in mind is to minimize the specialness and complexity of the process. This would facilitate the reusability of the process and reduce burden to debug it.

### A.2.1 Data Types, I/O Ports, and Constants

Six basic data types, bool, bit, byte, short, int, and double, are supported by **ppl**. Following the same order, they occupy 1, 1, 8, 16, 32, and 64 bits respectively. A variable is declared similarly to the syntax of C language. The declaration below creates a byte array and two initialized variables in Boolean type and double type respectively.

```
byte frame[32];
bool done=0;
double RTT=60.514;
```

Variables are regarded as local to a process, so their names can be reused in other processes. One way to make a variable accessible by another process is to declare it as a wireless port like:

```
wireless int votes;
```

In this case, all processes within the same scope can read/write value from/to the variable votes. The scope is a parameter of every agent Star. It can be explicitly specified a channel name to force different processes listening to the same channel.

The timer data type has the same resolution as double but it has very different behavior to previous basic data types. A timer counts up automatically as the simulation time proceeds. It can be set to any floating-point number at any moment as if it is manually adjusted to that epoch. The declaration 'timer t1=0.0' creates a timer t1 and resets it initially.

The user-defined data type follows the C-like syntax. For example, the following declaration defines a new data type PDU and a variable packet in that type.

```
typedef PDU {
  int header;
  byte data[1024];
  int checksum;
}
PUD packet;
```

The same as the syntax of C, the third byte in the data field of variable packet is expressed by packet.data[2].

I/O ports are the interface enables a process to communicate with another process. They can be categorized into two types, signal channels and message channels. A signal channel is an extension of a variable, which allows to send/ receive a value to/from a port. A typical declaration is given below.

```
inport byte Data;
outport double Result;
```

A message channel is unidirectional and first-in-first-out (FIFO). For instance,

```
inport chan dataIn = [256] of { int };
outport chan dataOut = [1] of { double };
```

declares an input message channel in int type with a buffer space of 256 slots and an output message channel in double type with single buffer space.

Two adjective keywords multi and persist are used to specify the properties of I/O ports. multi declares a multi-port that allows multiple connection to different processes. persist declares a persistent input port that keeps the present indicator on even if the data arrived the port at an earlier time. Two examples are given as follows.

```
multi outport bit chipSelect;
persist inport double batchMeasure;
```

A const is used to specify a parameter of a process. It becomes a state of the agent Star belonged to the process. It has an initial value but can be given a new value at run-time. A typical declaration looks like:

```
const int myID=123;
```

### A.2.2 General Statements

The arithmetic operation and Boolean expression of **ppl** are exactly the same as C language. They include +, -, *, /, %, ++, and -- for arithmetic; >, >=, ==, <=, <, and != for comparison; &&, ||, and ! for Boolean expression; &, |, ^, ~, <<, and >> for bitwise operation. Two consecutive statements are separated by a semicolon ';' or an arrow '->'. In **ppl**, Boolean expressions are simplified as regular statements. Therefore, the condition ((a==b)&&(c>d)) can be an independent statement. It will be either executable or blocked at run-time depending on the values of variables. An unexecutable statement will block the process until the condition becomes true later. This is the most common approach to synchronize with another process.

By definition, an assignment '=' is always executable. Assigning a value to an output signal channel implicitly issues a data output event. For example,

```
chipSelect=1;
```

sends out a bit '1' to the output port chipSelect. As for message channels, operators '?' and '!' are used to receive and send data respectively. For example,

```
dataIn?radius; dataOut!(radius*radius*3.14);
```

reads the head element from channel dataIn and writes it to variable radius. After that a computation result is sent out to channel dataOut. The operator '?' can be also used to test the head element of an input channel. The expression

```
dataIn?[5];
```

is not a reading operation. Instead, it is a Boolean condition that checks if the head element of channel dataIn has a value of 5. The command len(dataIn) is another way to check the status of a message channel, which returns the current number of elements queued in a message channel. The condition (len(dataIn)>=5) is executable when there are at lease 5 elements hold by dataIn.

Three commands are used to check and change the present indicator of a input port. present(dataIn) is executable if there is at least one new arrival at the input port dataIn. turnoff(dataIn) is always executable that turns off the indicator. admit(dataIn) functions as same as present(dataIn), but it turns off the indicator after the checking if there is indeed a new arrival. It is equivalent to the statements present(dataIn)->turnoff(dataIn).

To assure a process operates correctly, a user could use the printf command to print out the run-time values of specific variables. Its syntax is the same as the printf command in C. Another way to detect design faults is to place assert statements at some checking points. For instance,

```
assert( (a>b) || (c<d) );
```

takes no effect when the condition is true, but a violating condition will immediately stop the simulation and respond a warning message to the user.

### A.2.3 Control Flow

There are four control flow constructs in ppl: case selection, repetition, watching guard, and unconditional jump. The general form of a case selection is

```
if
  ::(condition 1)-> statements;
  ::(condition 2)-> statements;
  ::else-> statements;
fi
```

Exact one branch will be selected and executed at one time. If more than one condition are executable, one of them will be picked with an equal probability. On the other hand, if no condition is satisfied, the statements on else branch will be executed. Furthermore, suppose under the same case and the else option is absent, the process will be blocked until at least one condition becomes executable. To avoid the blocking, usually the else branch is given as 'else->skip;', which means to skip the whole selection construct if no condition is satisfied.

The second control flow is do loop. A do loop has exactly the same structure as the if construct. It will be executed repeatedly until it encounters a statement break. A factorial function can be implemented as follows.

```
f=1;
do
  ::(n>1)-> f=f*n; n--;
  ::else->break;
od
```

The watching-guard construct 'unless' has a structure shown below.

{ statement block 1 } unless { statement block 2}

Before each statement in block 1 is executed, the first statement in block 2 will be checked. If the later is unexecutable, statements in block 1 are executed repeatedly. Once the first statement in block 2 happens to be executable, the execution of statements in block 1 stops immediately. Note that a statement break in block 1 would also exit the loop.

The unconditional jump goto functions as it does in common computer languages. For example, 'goto Waiting' forces the program counter switching to the statement below label Waiting. Note that **ppl** identifiers cannot be labels.

### A.2.4 Timing Commands

The command delay(*duration*) suspends the process for *duration* time units. It is always unexecutable if the *duration* is positive, because the system time will not be advanced during an iteration. After having slept for *duration* time units, the process wakes up again and continues executing the statements after the delay command.

To model the time-out checking mechanism in a protocol, the command expire(*timer, target-time*) is used to check if a specific timer has expired. It also registers a likely time-out event in the future. Note the registered time-out event is not deterministic to happen since other events could abort the waiting state or a timer assignment could change the *target-time*.

Routine state checking is useful while specifying a protocol. A protocol module may enter an idle state for a long time and be unaware of something going wrong. A programmer could use a belatedly refiring command return(*duration*) to register a promissory return time to invoke the process again. Besides, to suspend execution immediately but not to register any return time, the command return(0) is usually used to yield control of execution.

### A.2.5 External Function Calls

To declare external C++ functions, the utility construct is used to specify the function names as well as their scopes. For example,

```
utility {
  pubFOO;
private:
  prvFOO;
}
```

declares two external functions pubFOO and prvFOO. The function pubFOO is public and could be shared with other processes, while the function prvFOO is private and not accessible by other processes. Suppose the **ppl** file containing above utility construct is named testFOO.ppl, the templates of these functions will be created in the names with pathes as './utility/pubFOO.cc' and './utility/testFOO/prvFOO.cc' after executing **ppl2pl**. It is user's responsibility to fill the code in these templates. A typical template looks like:

```
// Arguments stored in args[0], args[1], args[2], ...
// Do not erase the remark symbol ahead function name.
// int pubFOO(int* args)
{

 }
```

Note the declaration in the utility construct does not include the augments of functions. Therefore, given all variables are not double type, following forms of callings are all valid.

```
a=pubFOO(1, 2, i); b=pubFOO(2, k+2); c=prvFOO(5, 6, 7, 8); d=prvFOO();
```

A Ptolemy Block could be also used as an external function. The command extoper(*outport*, *inport*) fires a Ptolemy Block and waits a reply from it. The command sends out a triggering signal to *outport* and waits a new arrival, usually the computation result from the Block, at *inport*. This command enables using existing Ptolemy built-in Blocks as the computation subroutines of a **ppl** process.

## A.3 A Simple Example

In this section, we use a simple example to walk through the design procedures of SiP. The following two processes implement a redundant transmission protocol. Upon receiving a POLL request, the Sender process sends an integer and a redundant copy of the number to the Receiver. The Receiver checks if the two copies have the same value to decide whether the data have been corrupted during transmission. If the two copies are the same, the Receiver sends out the number to dataOut channel and sends another POLL request to the Sender; otherwise, the Receiver sends a NACK notification to the Sender for requesting a retransmission. To differentiate the iterations of transmission, the Sender increases the sending number whenever it receives a POLL request.

**Sender.ppl:**

```
#define POLL 1
#define NACK 2
proctype Sender()
{
 inport chan chIn = [10] of { byte } ;
 outport chan chOut = [10] of { int };
 int x=0;
 do
   ::chIn?POLL->x++; chOut!x; chOut!x;
   ::chIn?NACK->chOut!x; chOut!x;
 od;
}
```

**Receiver.ppl:**

```
proctype Receiver()
{
 inport chan chIn = [10] of { int };
 outport chan chOut = [10] of { byte };
 outport chan dataOut = [10] of { int };
 int x1,x2;
 chOut!POLL;
 loop:
  chIn?x1->chIn?x2;
  if
    ::(x1==x2)->dataOut!x1; chOut!POLL;
    ::else->chOut!NACK;
  fi;
```

```
goto loop;
}
```

Applying **ppl2pl** to both processes, we obtain two **pl** files DESender.pl and DEReceiver.pl in directory './ptolemy/' as the source code of agent Stars. We then create two agent Stars in the facet './user.pal' by using the 'make star' function in Ptolemy environment. Now we are ready to open a new facet and specify the schematic of our system. However, it is perceivable that the transmission will be error-free if we directly connect the I/O ports of the two agent Stars. To implement an unreliable channel, we adopt the built-in AWGN (Additive White Gaussian Noise) Galaxy to add noise to data. We also include a Delay Star to model the propagation delay. Figure A.2 shows the design of the channel model.

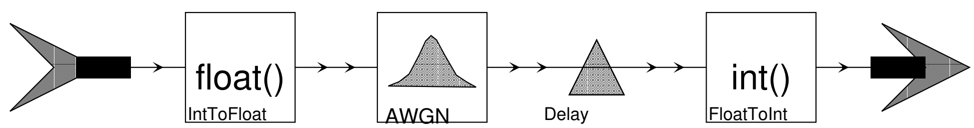



Figure A.2 The model of an unreliable channel with propagation delay.

The schematic of our system is shown in Figure A.3. The XMgraph Star is used to display the received data.

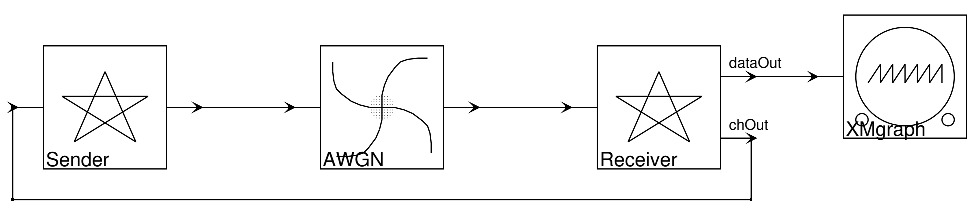



Figure A.3 A communication system over an AWGN channel.

Figure A.4 gives a simulation result of the above system. Note that there was a transmission error at time 9 and a later retransmission made it up. However, the redundant transmission protocol cannot guarantee the correctness of received data. Suppose the two copies of data were both corrupted during transmission and happen to have the same value when the Receiver reads them, the Receiver will regard them as a correctly received pair. As shown in Figure A.5, the Receiver made a wrong decision at time 5.

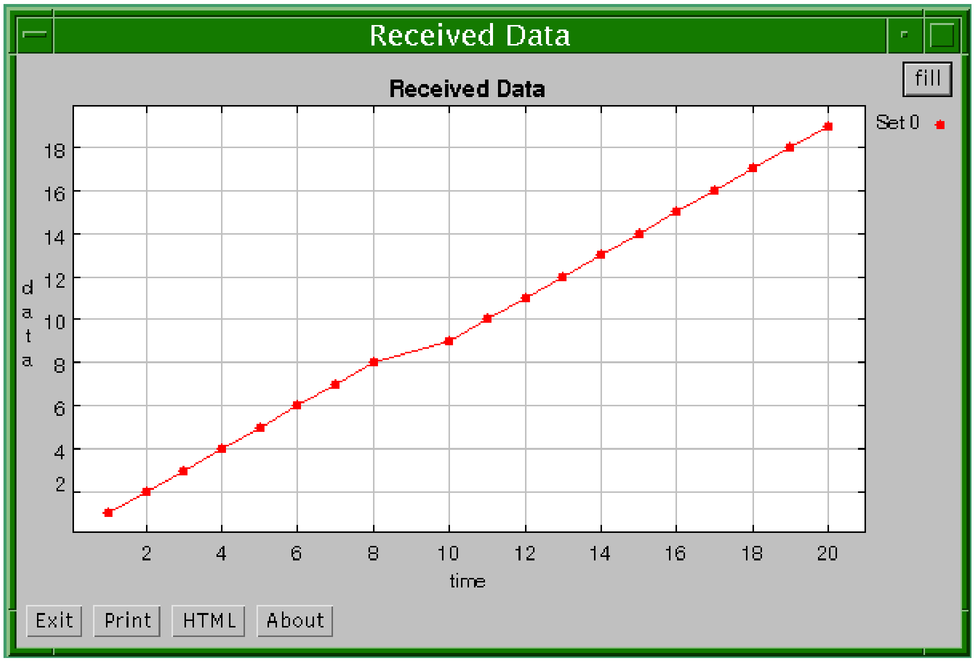



Figure A.4 A simulation result shows the error recovery ability of the redundant transmission protocol.

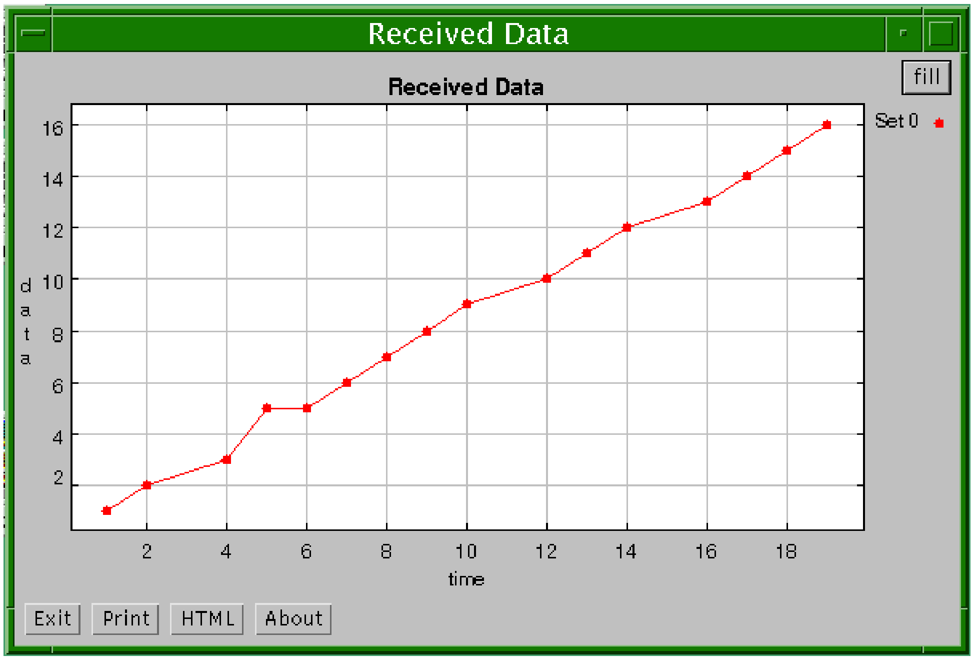



Figure A.5 A simulation result shows that the redundant transmission protocol cannot guarantee an error-free transmission.

# Appendix B: SiP v1.2 Programmer's Manual

## B.1 Introduction

SiP (SPIN in Ptolemy) is a system-level protocol modeling tool developed at University of California at Berkeley. Its software package includes a stand-alone executable file **ppl2pl**, a Ptolemy-supported SPIN package (modified from SPIN v3.0 by Lucent Technologies - Bell Labs), and several supplemental files to the Ptolemy DE domain. We assume in this manual that the readers are thoroughly familiar with the DE domain and know how to write a DE Star. Refer to the Chapter 12 in Volume II of Almagest, the Ptolemy's Programmer's Manual (ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/www/papers/almagest/prog.html). Readers are also encouraged to read SiP User's Manual to have the background knowledge of the Ptolemy-supported PROMELA Language, called **ppl**. In following sections, we will focus on the implementation issues of **ppl2pl**, agent Star, and the Ptolemy-supported SPIN kernel.

## B.2 Translator ppl2pl

Each leaf cell of a protocol module is specified by a **ppl** process. It will neither be understood by Ptolemy nor SPIN without translation. SiP provides the **ppl2pl** translator to generate a **pl** (Ptolemy Language) file of the customized agent Star from a **ppl** process, such that the Ptolemy kernel can access to that process through its agent Star.

Since an agent Star only customizes the mechanism to read/write Particles from/to the internal channels in SPIN, the **ppl2pl** only processes the declaration of I/O ports, parameterizable constants, and external functions of a **ppl** process to generate the code of its agent Star. The detailed actions taken by **ppl2pl** to deal with these three types of declaration are listed as follows.

- **I/O ports**: The `inport` and `outport` in a **ppl** process are converted into the `input` and output constructs in the **pl** file respectively. A port with data types bool, bit, byte, short, and int in the **ppl** process are all specified as an integer-typed port in the **pl** file, and the double is mapped to the float. Keyword multi is retained to indicate the port is a multiple I/O port, while keyword wireless tells **ppl2pl** to neglect the port because a wireless port is not implemented as a regualr Ptolemy I/O port. Usually in go() an extra statement is applied to each input port to maintain the present indicator of the port. The keyword persist disables that maintenance to keep the incoming Particles persistent. In go(), each input port relies on a loop to forward all arriving Particles to its corresponding channel in SPIN. Each output port also has a loop to flush out all queued elements in its corresponding channel in SPIN. These forwarding loops are added in go() while an I/O port declaration is detected in the **ppl** process. Because SPIN uses a special data structure to access its channels, in a **pl** file some temporary variables in that data structure are included in the constructor construct and are deleted in the destructor construct.

- **parameterizable constants**: The const declaration in the **ppl** process is converted into the defsate construct in the **pl** file. Its data type and initial value are specified by corresponding fields in the defstate construct. Since the value of a state could be changed at run-time, an extra statement is added in go() to set the new value of the const in SPIN at the beginning of the first arrival to the agent Star.

- **external functions**: The **ppl** allows including public and private external C++ functions. A public function is included in the code construct in the **pl** file so that it is accessible by other agent Stars. A private function is included in the method construct to add a new member function to the class of the agent Star (each agent Star is a new class derived from DESiPStar) so that the function is only accessible by the agent Star itself. Since a function is tracked by a pointer to make it accessible by the SPIN evaluator, for each function **ppl2pl** creates a pointer in the protected construct and assigns its value in the constructor construct.

The **ppl2pl** also auto-documents the generated **pl** file by filling its author and descriptor constructs. This will require executing some UNIX system commands.

## B.3 Agent Star

The best way to understand the functions of an agent Star is to do a side-by-side comparison of its code with its original **ppl** process. An example is given below to be referred by the discussion in the rest of this section.

### ppl process (MOD.ppl):

```
#define Base 100
proctype MOD()
{
 wireless bool running=1;
 inport int value;
 outport int result;
 const int modulus=10;
 loop:
  running->admit(value)->result=Base+(value-Base)%modulus;
 goto loop;
}
```

### generated pl file (DEMOD.pl):

```
defstar
{
   name { MOD }
   domain { DE }
   derivedfrom { SiPStar }
   author { Shang-Pin Chang }
   descriptor { DEMOD.pl is an agent star associated with PROMELA file MOD.ppl. It was
generated by spchang@coulomb.eecs.berkeley.edu on Sun Dec 13 01:38:00 PST 1998. }
   copyright { Copyright (c) 1990-1998 The Regents of the University of California. All
rights reserved. See the file $PTOLEMY/copyright for copyright notice, limitation of liability,
and disclaimer of warranty provisions. }
   output {
      name { result }
      type { int }
   }
   input {
```

```
   name { value }
   type { int }
}
defstate {
   name { scope }
   type { string }
   default { "" }
   desc { Scope of the wireless ports used in file MOD.ppl }
}
defstate {
   name { instOp }
   type { int }
   default { 0 }
   desc { Set to 1 for an instant operation module, otherwise 0. }
}
defstate {
   name { modulus }
   type { int }
   default { "10" }
   desc { Initial value of modulus used in file MOD.ppl }
}
protected {
   Symbol *sym_result;
   Symbol *sym_value;
   double lastP_value;
}
destructor {
   unregisterPXPStar();
}
method {
   name { getScope }
   access { protected }
   arglist { "(void)" }
   type { "const char*" }
   code { return (const char*)scope; }
}
code {
   extern RunList* findproc3(int ID);
   extern Symbol* findloc2(RunList *x, const char *s);
}
setup {
   delayType = instOp? FALSE:TRUE;
}
begin {
   SPINReady = FALSE;
   DERepeatStar::begin();
```

```
      outputPML(((strlen(scope)>0)? 1:0),"/export/coulomb/coulomb1/spchang/thesis/
examples/",fullName(),name(),className(),(const char*)scope);
      registerPXPStar();
   }
   go {
    if (!SPINReady) {
     run_Proc = findproc3(sipID);
     run_Proc->host = this;
     sym_result = findloc2(run_Proc,"result");
     sym_value = findloc2(run_Proc,"value");
     lastP_value = 0.0;
     ((int *)(findloc2(run_Proc,"modulus")->val))[0] = (int)(modulus);
     SPINReady = TRUE;
    }
    if (lastP_value<arrivalTime) sym_value->present = 0;
    while (value.dataNew) {
     ((int *)(sym_value->val))[0] = (int)(value.get());
     sym_value->present = 1;
     lastP_value = arrivalTime;
    }
    if (arrivalTime >= completionTime) {
     NxtFireDur = RunSpin(run_Proc,arrivalTime);
     if (sym_result->updated) {
      result.put(arrivalTime) << ((int *)(sym_result->val))[0];
     }
     setNextFiring();
     GlobalUpdate();
    }
   }
}
```

### B.3.1 PROMELA Code Generation outputPML()

To generate the PROMELA code of all agent Stars on a schematic before the first Star is fired, we include the function call *outputPML()* for PROMELA code generation in the begin construct in the **pl** file. Since the begin method is executed exactly once, all agent Stars would also output their PROMELA code exactly once. However, there may be many cloned agent Stars on a schematic and it results a very large PROMELA file containing repeated processes. This would then reduce the efficiency while generating the parse tree. Benefitted from SPIN's ability to support multiple instances of a process, we only let the first visited agent

Star generate the code and force its clones within the same scope be distinct instances of the same process. We will discuss the scope of an agent Star later.

Opposite to the function of **ppl2pl**, *outputPML()* processes the **ppl** process body instead of its I/O ports, constants, and external functions. The major task of *outputPML()* is to set all defines and wireless ports in the generated PROMELA code having a correct scope. For example, suppose the agent Star of the above **ppl** process MOD has a clone in both galaxy G1 and G2, its statement

```
#define Base 100
```

will appear in the generated PROMELA code twice, which is not a desired result. This is solved by casting the define with the scope of its agent Star as follows.

```
#define G1_Base 100
#define G2_Base 100
```

Such scope casting also applies to their process names and wireless ports. Therefore, the generated PROMELA code contains following statements:

```
proctype G1_MOD(int _SIP_ID)
  .....
 G1_running->admit(value)->result=Base+(value-Base)%modulus;
  .....
proctype G2_MOD(int _SIP_ID)
  .....
 G2_running->admit(value)->result=Base+(value-Base)%modulus;
```

Note that in this case their wireless ports listen to different channels G1_running and G2_running. This is the usual case when the clones of an agent Star are embedded in different galaxies. To force them having the same scope, explicitly give the same name to the scope state of both clones (see defstate scope in above DEMOD.pl). For example, let the name of scope be ALL and the generated PROMELA code would contain only one process and two instances like:

```
#define ALL_Base 100
proctype ALL_MOD(int _SIP_ID)
{
  .....
 ALL_running->admit(value)->result=Base+(value-Base)%modulus;
```

```
}

init
{
 run ALL_MOD(1);
 run ALL_MOD(2);
}
```

### B.3.2 Pointer Binding

The pointer binding of the ports/states of an agent Star and the channels/ constants in its corresponding SPIN process is accomplished during the first visit to the go() method of the agent Star. The code within the (!SPINReady) block at the beginning of the go() method in the above DEMOD.pl is the additional code executed during the first visit to the go() method. The agent Star uses function *findproc3()* to locate the pointer of its corresponding SPIN process and locates two signal channels value and result within that process by using function *findloc2()*. These pointers will facilitate the access to the process and channels in SPIN during subsequent visits to the go() method of the agent Star. Note that the pointer of const modulus is also located to set its initial value using the value of state modulus of the agent Star. At this point, the agent Star lets SPINReady be true to indicate the pointer binding is finished.

### B.3.3 The Scenario of go() Method

Except the extra code executed at the beginning of the first visit to the go() method, a regular execution scenario of the go() method is described as follows. Readers should refer to the code inside go() method after the (!SPINReady) block in the above DEMOD.pl while reading this section.

The *lastP_value* is used to denote the arrival time of the previous arrival to port value. If it is earlier than the current system time, the present indicator of port value is turned off before the testing of new arrivals at the port. Then the agent Star uses a loop to get all Particles in port value and writes them to the signal channel value in its SPIN process. At the same time the arrival time is recorded in *lastP_value* and the present indicator is turned on if any Particle has been detected at the port.

After forwarded all Particles, the agent Star is ready to call SPIN kernel to execute its corresponding SPIN process for one iteration. However, the prerequisite is the *arrivalTime* must be later or equal to the *completionTime*. This would not be satisfied when the process is executing a delay() command and the duration has not expired yet. Otherwise, the process will be executed for one iteration by calling *Run_Spin()*. After that the agent Star checks whether if the outport result has been updated during that iteration. If yes, the updated value is sent out to the output port result of the agent Star to form a new Particle.

The last two steps are *setNextFiring()* and *GlobalUpdate()*. The former estimates the refiring time to the agent Star and sends out a dummy Particle with a future time stamp to the *feedbackOut* port. The later checks whether if any wireless outport has been updated during the iteration at SPIN kernel. If so, it will fire all agent Stars having a wireless inport listening to the same channel within the same scope.

## B.4 Ptolemy-supported SPIN Kernel Run_Spin()

*Run_Spin()* allows the SPIN kernel to resume the interpretation of the process from the last unexecutable statement at its previous iteration till the first unexecutable statement at the current iteration (could be the same statement). Before the interpretation begins, two routines have to been done. First, all update indicators of outports are turned off. Thus an agent Star could check which ports have been updated during the iteration and generate new Particles for them. Second, all timers in the process are advanced by the elapsed time from the last iteration to the current system time. This is essentially important to the correct functioning of timing commands in the **ppl** specification. Besides, the interpretation greatly relies on the SPIN evaluator *eval()* defined in *DESiPrun.cc*, where all SiP command are explicitly listed and self-explanatory to show their detailed steps of evaluation.

# Bibliography


[1] O. Tanir, "Modeling Complex Computer and Communication Systems -- A Domain-Oriented Design Framework," McGraw-Hill, NY, 1996.

[2] L. L. Peterson, Bruce S. Davie, "Computer Networks -- A Systems Approach," Morgan Kaufmann Publishers, CA, 1996.

[3] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," Revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California at Berkeley, April 1998.

[4] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," Int. Journal of Computer Simulation, special issue on "Simulation Software Development," vol.4, pp. 155-182, April, 1994.

[5] C. Cassandras, "Discrete Event Systems, Modeling and Performance Analysis," Irwin, Homewood IL, 1993

[6] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, "Hardware- Software Codesign of Embedded Systems," IEEE Micro, pp. 26-36, August 1994.

[7] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," Sci. Comput. Program., vol. 8, pp. 231-274, 1987.

[8] F. Belina, D. Hogrefe, A. Sarma, "SDL with Applications from Protocol Specification," Prentice Hall International (UK), Hemel Hempstead, 1991.

[9] N. A. Lynch, "Distributed Algorithms," Morgan Kaufmann Publishers, CA, 1996.

[10] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," Proceedings of the IEEE, Vol. 79, No. 9, pp. 1270-1282, 1991.

[11] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semamtics, Implementation," Science of Computer Programming, 19(2):87-152, 1992.

[12] W.-T. Chang, S. Ha, E. A. Lee, "Heterogeneous Simulation -- Mixing Discrete-Event Models with Dataflow," invited paper, "Journal on VLSI Signal Processing, Vol. 13, No. 1, January 1997.

[13] R. Cleveland, S. A. Smalka, et al., "Strategic Directions in Concurrency Research," ACM Computing Surveys, Vol. 28, No. 4, December 1996.

[14] M. G. Gouda, "Elements of Network Protocol Design," John Wiley & Sons, 1998.

[15] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, August 1978.

[16] J. Hopcroft and J. Ullman, "Introduction to Automata Theory, Language, and Computation," Addison- Wesley Publishing Company, 1979.

[17] M. Ben-Ari, "Principles of Concurrent and Distributed Programming," Prentice Hall International (UK) 1990.

[18] R. Sharp, "Principles of Protocol Design," Prentice Hall International (UK) 1994.

[19] E. A. Lee, "A Denotational Semantics for Dataflow with Firing," Memorandum UCB/ERL M97/3, Electronics Research Laboratory, University of California at Berkeley, January 1997.

[20] E. A. Lee, "Modeling Concurrent Real-Time Processes Using Discrete Events," Invited paper to Annuals of Software Engineering, Special Volume on Real-Time Software Engineering, to appear, 1998. Also UCB/ERL Memorandum M98/7, March 1998.

[21] E. A. Lee, T. M. Parks, "Dataflow Process Networks," Proceedings of the IEEE, Vol. 83, No. 5, pp. 773-801, May 1995.

[22] Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems," Springer-Verlag, 1991.

[23] F. Maraninchi, "Operational and Compositional Semantics of Synchronous Automaton Compositions," CONCUR 92, Third International Conference on Concurrency Theory," Vol. 630 of Lecture Notes in Computer Science, pp. 550-564, Springer-Verlag, August 1992.

[24] Edited by S. Mauw, G. J. Veltink, "Algebraic Specification of Communication Protocols," Cambridge University Press, 1993.

[25] Asawaree Kalavade, "System Level Codesign of Mixed Hardware-Software Systems," Tech. Report UCB/ERL 95/88, Ph. D. Dissertation, Dept. of EECS, University of California at Berkeley, September 1995.

[26] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21, No. 7, July 1978.

[27] G. E. Keiser, "Local Area Networks," McGraw-Hill 1989.

[28] D. Bertsekas, R. Gallager, "Data Networks," Prentice Hall, NJ, 1992.

[29] D. L. Dill, A. J. Drexler, A. J. Hu, C. H. Yang, "Protocol Verification as a Hardware Design Aid," IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 522-525, 1992.

[30] G. J. Holzmann, "Design and Validation of Computer Protocols," Prentice-Hall, 1991.

[31] G. J. Holzmann, Doron Peled, "Partial Order Rduction of the State Space," SPIN Workshop, Montreal, Quebec, October 1995.

[32] P. Godefroid, "The ULG Partial Order Package for SPIN," SPIN Workshop, Montreal, Quebec, October 1995.

[33] G. J. Holzmann, "An Analysis of Bitstate Hashing," Protocol Specification, Testing and Verification, 15th International Conference, pp. 301-314, 1995.

[34] C. N. Ip, D. L. Dill, "Better Verification Through Symmetry," 11th International Conference on Computer Hardware Description Languages and their Applications, pp. 97-111, 1993.

[35] U. Stern, D. L. Dill "Combining State Space Caching and Hash Compaction," Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop, pp. 81-90, 1996.

[36] J. Billing, M. C. Wilbur-Ham, M. Y. Bearman, "Automated Protocol Verification," Protocol Specification, Testing, and Verification, V, pp. 59-70, 1986.

[37] S. Ha, "Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs," Ph. D. Dissertation, Dept. of EECS, University of California at Berkeley, 1992.

[38] J. Misra, "Distributed Discrete-Event Simulation", Computing Surveys, Vol. 18, No. 1, November 1985.

[39] R. Righter, J. C. Walrand, "Distributed Simulation of Discrete Event Systems," IEEE Proceedings, Vol. 77, No. 1, pp. 99-113, January 1989.

[40] S. Budkowski, E. Najm, "Structured Finite State Automata -- A New Approach for Modelling Distributed Communication Systems," Protocol Specification, Testing and Verification, III, Elsevier Science Publishers B. V (North-Holland), 1983.

[41] M. Daniele, P. Renditore, R. Manione, "Interactive Timed Simulation of Distribution Systems -- from PROMELA to PROMELA+," SPIN Workshop, Montreal, Quebec, October 1995.

[42] E. Mikk, Y. Laknech, M. Siegel, "Toward Efficient Model Checking Statecharts: A Statecharts to PROMELA Compiler," SPIN Workshop, Rutgers University, NJ, August 1996.

[43] D. Clark, D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," Proc. ACM SIGCOMM '90, Philadelphia, PA, September 1990.

[44] D. Clark, S. Shenker, L. Zhang, "Supporting Real-Time Applications in an Integrated Packet Services Network: Architecture and Mechanism," Proc. ACM SIGCOMM, 1992.

[45] S. McCanne and M. Vetterli, "Joint Source/Channel Coding for Multicast Packet Video," Proceedings of the IEEE International Conference on Image Processing, Washington, DC, Vol. 1, pp. 25-28, October 1995.

[46] O. Grumberg, D. E. Long, "Model Checking and Modular Verification," ACM Trans. on Programming Languages and Systems, 16(3):843-871, 1994.

[47] R. Milner, "Calculi for Synchrony and Asynchrony," Theoretical Computer Science, 25(3):267-310, 1983.

[48] M. Schwartz, "Telecommunication Networks: Protocol, Modeling, and Analysis," Addison-Wesley, 1987.

[49] J. B. Stefani, L. Hazard, F. Horn, "Computational Model for Distributed Multimedia Application Based on a Synchronous Programming Language," Computer Communications, 15(2), March 1992.

[50] P. Chou, E. A. Walkup, G. Borriello, "Scheduling for Reactive Real-Time Systems," IEEE Micro, 14(4):37-47, August 1994.

[51] M. B. Abott, L. L. Peterson, "A Language-Based Approach to Protocol Implementation," IEEE/ACM Trans. on Networking, Vol. 1, No. 1, February 1993.

[52] A. Hanish, T. Dillon, "Object-Oriented Modelling of Communication Protocols for Re-Use Static/Dynamic Modelling Aspects," Dept. CSCE, TR 2-3/95, La Trobe University, Australia, 1995.

[53] R. E. Nance, "The Time and State Relationships in Simulation Modelling," Communications of the ACM, 24(4):173-179, 1981.

[54] R. E. Nance, "On Time Flow Mechanisms for Discrete System Simulation," Management Science, 18(1):59-73, September 1971.

[55] J. W. Davies, S. A. Schneider, "A Brief History of Timed CSP," Technical Monograph PRG-96, University of Oxford (UK) 1992.