ABSTRACT

Title of dissertation: SeSFJava: A FRAMEWORK FOR DESIGN AND ASSERTION-TESTING OF CONCURRENT SYSTEMS

Tamer Elsharnouby, Doctor of Philosophy, 2005

Dissertation directed by: Professor A. Udaya Shankar
Department of Computer Science

Many elegant formalisms have been developed for specifying and reasoning about concurrent systems. However, these formalisms have not been widely used by developers and programmers of concurrent systems. One reason is that most formal methods involve techniques and tools not familiar to programmers, for example, a specification language very different from C, C++ or Java. SeSF is a framework for design, verification and testing of concurrent systems that attempts to address these concerns by keeping the theory close to the programmer's world.

SeSF considers **layered compositionality**. Here, a composite system consists of layers of component systems, and **services** define the allowed sequences of interactions between layers. SeSF uses conventional programming languages to define services. Specifically, SeSF is a markup language that can be integrated with any programming language. We have integrated SeSF into Java, resulting in what we call SeSFJava. We developed a testing harness for SeSFJava, called SeSFJava Harness, in which a (distributed) SeSFJava program can be executed, and the execution checked against its service and any other

correctness assertion. A key capability of the SeSFJava Harness is that one can test the final implementation of a concurrent system, rather than just an abstract representation of it.

We have two major applications of SeSFJava and the Harness. The first is to the TCP transport layer, where service specification is cast in SeSFJava and the system is tested under SeSFJava Harness. The second is to a Gnutella network. We define the intended services of Gnutella – which was not done before to the best of our knowledge – and we tested an open-source implementation, namely Furi, against the service.

SeSFJava: A FRAMEWORK FOR DESIGN AND ASSERTION-TESTING OF CONCURRENT SYSTEMS

by

Tamer Elsharnouby

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:

Professor A. Udaya Shankar, Chair/Advisor
Professor Eyad Abed, Dean’s Representative
Professor Ashok Agrawala
Professor Samrat Bhattacharjee
Professor Atif Memon

To My Parents Amani and Mahmoud,

To Rehab and Grandpa Mohamed

ACKNOWLEDGMENTS

I owe my gratitude to all the people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my adviser, Professor A. Udaya Shankar for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past years. He has always made himself available for help and advice and there has never been an occasion when I've knocked on his door and he hasn't given me time. It has been a pleasure to work with and learn from such an extraordinary individual.

I owe my deepest thanks to my family - my mother *Amani* and my father *Mahmoud* who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them. I would also like to thank my sister Rehab and my grandpa Mohamed.

My friends have been a crucial factor in my finishing smoothly. I'd like to express my gratitude to Khaled Arisha, Ahmed Elgammal, Mohamed Elmohandes, Gehad Galal, Ayman Khalafallah, Tamer Nadeem, Zaki Sharbash and Adel Youssef for their friendship and support. I would also like to thank my friends Mohamed Abdallah, Ahmed Abdel Hafez, Abdel Hameed Badawy, Mona Diab, Tamer Elbatt, Mahmoud Elfayoumy, Ashraf Elmasry, Mohamed Tamer Elrefae, Tarek Ghanem, Walid Gomaa, Yaser Jaradat, Hesham Mahmoud, Anis Valiani and Moustafa Youssef.

It is impossible to remember all, and I apologize to those I have inadvertently left out. Thank Allah and thank you all!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Formal specification and correctness analysis of concurrent systems have been investigated since the mid-1970s. A **concurrent system** is a collection of active entities that execute simultaneously and interact with each other during the course of their lifetimes. Formal specification of a system refers to a description of the desired external behavior of the system in a language with mathematically defined syntax and semantics. Correctness analysis of a system is a proof that the system **satisfies** its specification, i.e., the system does what it is supposed to do. The terms “specification” and “system satisfies its specification” have various interpretations in software engineering, spanning functionality, performance, reliability and resource utilization. Here, we use the terms in the sense of correct functionality. The execution of a concurrent system is represented by the sequence of state changes, and a correctness property is a condition on this sequence of states.

Many elegant formalisms have been developed for specifying and reasoning about concurrent systems, for example, Lynch and Tuttle’s I/O automata [49], Chandy and Misra’s UNITY [12, 58], Lamport’s TLA [44], Milner’s CCS [57], Hoare’s CSP [33, 69], Manna and Pnueli's Temporal Logic [50, 51], and Lam and Shankar's relational notation [43]. However, these formalisms have not been widely used by developers and programmers of concurrent systems. One reason is that most formal methods involve techniques and tools that are not familiar to programmers, for example, a specification language that is very different from Java, C or C++.

SeSF is a framework for design, verification and testing of concurrent systems [78] that addresses these concerns by keeping the theory close to the programmer's world. This dissertation presents an implementation of SeSF in Java. SeSF uses the term **service specification** to refer to a formal specification of a system, and the term **system specification to** refer to the description of the system itself.

The system specification is intended for execution. Hence, it is defined by programs written in an implemented programming language. Furthermore, it must satisfy the computational, synchronization, and other constraints of the underlying platform– for example, does the platform have a single processor, a multi-processor with shared memory, or a set of loosely-coupled message-passing processors.

The service specification is a description of the external behavior of the system, capturing all (and only) the desired properties and unencumbered by implementation issues and internal structure. Its primary goal is to be easy to understand. This invariably means that the service specification assumes much more powerful atomicity, memory, and computation than is required by the system specification.

In addition, to systems and services, SeSF also formalizes the notion of a system satisfying its service. Informally, this holds if (1) the system is ready to accept any input allowed by the service, and (2) any output the system does is allowed by the service.

Like most formalisms, SeSF provides **compositionality**. This means that in a composite system, if a component system is replaced by another system that satisfies the service of the original component system, then the composite system continues to work properly. In most formalisms, the service defines the permissible interactions between the system and its environment. However, our interest is in **layered compositionality**. Here, a composite system consists of layers of component systems, and services define the allowed sequences of interactions between layers.

1.1 SeSFJava

Services can be defined using one of the following: (1) an abstract non-executable formalism, e.g., TLA [44] and CSP [33, 69]; (2) a high-level specification language that can be compiled and executed, e.g., PAISLey [88] and Statemate [29]; and (3) a conventional programming language, e.g., C, C++ and Java.

The second and third options lead to what we call **executable services** (or executable specifications). The adoption of executable services, in general, and in SeSF in particular, has the following consequences. First, the notion of a system satisfying a service is equivalent to the composite program of the system and service satisfying certain correctness properties. Second, developers can *test* a concurrent system against its service simply by executing the composite program of the system and the service, and checking whether those properties are violated.

Compared to the second option, the third option has certain advantages and disadvantages. One advantage of the third option is that the service specification language is familiar to programmers, perhaps even the same language as that of implementation. This reduces the possibility of the service specification being misunderstood by implementors. Another advantage is that it allows actual implementations to be tested, rather than an abstract model. One disadvantage of the third option is that service specifications are invariably larger in size, making “mechanical verification” practically infeasible, although we think that this is not a big loss because mechanical verification is currently impractical for unbounded-state models. Another concern is that the service specification language may suffer from inconsistencies and ambiguities that plague most conventional programming languages.

Our approach is really a mix of the second and third options. SeSF itself is a high-level executable language, but it is not compiled. Instead, we treat SeSF as a markup language that can be integrated with any programming language. We integrate SeSF with Java, resulting in what we call **SeSFJava**. We choose Java because of its relatively precise semantics, popularity and built-in concurrency constructs. A SeSFJava program is a Java program with SeSF tags inserted as Java comments. Both systems and services are specified by SeSFJava programs.

Because the SeSFJava tags are Java comments, a SeSFJava program can be compiled and executed as a Java program. Thus, for implementation purposes, a SeSFJava system program is identical to the original Java program. But because of the SeSF tags, it can also be tested against its service and other correctness assertions. We developed a testing harness, called **SeSFJava Harness**, that can execute a (distributed) SeSFJava system program and check whether the resulting execution satisfies the relevant SeSFJava service program and any other desired correctness assertions.

This is not straightforward because the SeSF tags are at a much higher level than most programming languages, including Java. In particular, to test a system against its services, the Harness must construct the composite program of system and services, which is not trivial in the context of dynamically created objects and processes. SeSFJava Harness is able to handle general Java programs (e.g., not restricted to finite-state programs) and general services with arbitrary safety and progress assertions.

The development of SeSFJava and SeSFJava Harness is motivated to a large extent by the desire to eliminate errors that are introduced in going from formal specifications to implementations, due to the developer's lack of expertise with the specification language or formal methods in general. By defining specifications in conventional languages, SeSFJava frees the programmer from having to understand two different languages. SeSFJava Harness allows developer to test a system on its actual platform.

As mentioned earlier, using Java as a specification language exposes specifications to the flaws and ambiguities of Java. For example, Java has an ambiguous memory model [67, 52], and different Java implementations have different memory models. For another example, Java Virtual Machines running on Windows have three priority levels, whereas those running on SunOS have thirty one priority levels.

1.2 Bank example

We illustrate the discussion above with a very brief example of a Bank system and Client systems as shown in figure 1.1. This example is expanded upon in chapters 4 and 5. The clients request to update (either by depositing to or withdrawing from) a shared account in the Bank. The Bank informs the client of the transaction outcome whether it is accepted (ack) or rejected (nack). The service Account specifies the acceptable sequences of interactions (update, ack and nack) the lower layer of the Bank and the upper layer of the Clients. Interactions are . In particular, (1) each client has at most one update pending, and (2) the outcome of an update is nack iff the update is a withdrawal for an amount greater than the balance in the account.

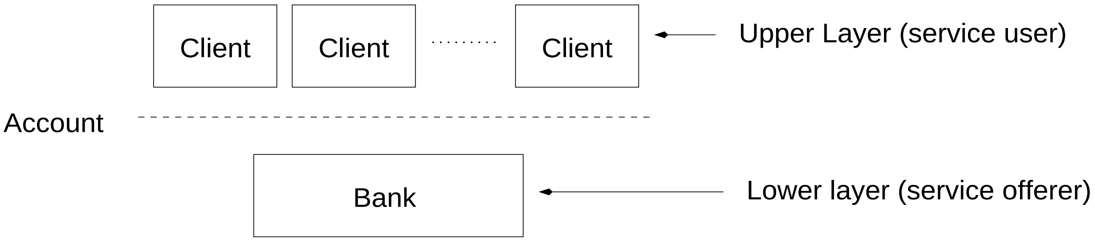


Figure 1.1: Example: systems and services

The Bank and Client systems are specified by SeSFJava system programs. System programs are similar to *classes* in programming languages, and systems are *instances* of the classes. A system program defines constants, variables, functions that are initiated (callable) by the environment, referred to as **xc events** (externally controlled events), and functions that are initiated by the system itself, referred to as **lc events** (locally-controlled events). Each event has an enabling condition which has to hold for its execution to be correct (e.g., a client can initiate an update only if it currently has no outstanding balance). SeSF does not impose any particular behavior when a system's xc event is called when not enabled. The system can block, or respond with an error message, but it may also not check (and perhaps behave unpredictably later on).

The Account service is specified by a SeSFJava service program that defines the events of the lower layer callable from the upper layer, referred to as **dnw events** (downward events), and the events of the upper layer callable from the lower layer, referred to as **upw events** (upward events). So Account defines dnw event update and upw events ack and nack. SeSF also defines the conditions for system Bank to satisfy service Account, and the conditions for system Client to satisfy service Account.

The Bank and Client system programs as well as the Account service programs are in SeSFJava, that is, they are Java programs with SeSFJava tags. These programs are compiled using a standard Java compiler. The SeSFJava tags are inserted as comments. For example, to specify that event update of system Bank is callable from the environment (clients in this case), the programmer inserts tag "//# xc_event;" just before the method's header.

For implementation, the programmer can treat the Bank and Client programs as Java programs, compile them, and run them as illustrated in figure 1.2(a).

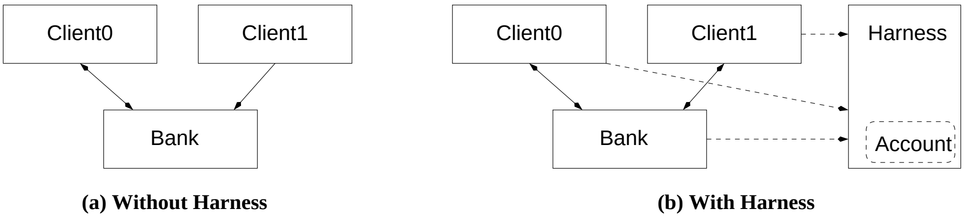


Figure 1.2: Composite system of bank and two clients

For testing, the programmer uses a SeSFJava preprocessor to process the SeSFJava tags inside the SeSFJava programs. This preprocessor instruments the program so as to connect the Client and Bank systems to the Harness, and send the local snapshots (data and control variables) of the systems to the Harness at predetermined breakpoints. After preprocessing, the systems and the Harness, which includes the preprocessed Account program, are compiled using a standard Java compiler. Then the Clients, Bank and Account are executed together under the control of the Harness (figure 1.2(b)). During the execution, the Harness constructs a global snapshot from these local snapshots states, and records any violations to properties stated in the systems and services.

1.3 Applications

We have done two major applications of SeSFJava and the Harness. The first is to the TCP transport protocol, where the service specification is cast in SeSFJava and the system is tested under SeSFJava Harness. The second is to a Gnutella network. We define the intended services of Gnutella – which was not done before to the best of our knowledge – and tested an open-source implementation, namely Furi, against the services.

The TCP transport protocol application was also done in the context of the introductory networks course (CMSC417) at the Department of Computer Science of the University of Maryland. This educational use of SeSFJava and Harness was motivated by our desire to expose students to formal methods and to see its effectiveness in a "real-life" situation. Networking course projects are usually described by an informal specification and a collection of test cases. Students often misunderstand the specification or oversimplify it to fit just the test cases. Using formal methods, in general, eliminates these misunderstandings and allows the students to test their projects thoroughly, but at the expense of learning a new specification language within the tight time schedule of the semester. Using SeSFJava eliminates such expense. The use of SeSF significantly increased the percentage of students who completed the projects, reduced their email queries about the specification, and reduced the grading time.

1.4 Structure of the Dissertation

Chapter 2 presents related work. Chapter 3 describes SeSF. Chapter 4 uses the Bank example to introduce SeSFJava. Chapter 5 applies the SeSFJava Harness to the Bank example. Chapter 6 describes the SeSFJava Harness and assertion checking more extensively. Chapter 7 describes the application of the Harness to the data transfer protocol. Chapter 8 describes the application of the Harness to the connection management protocol. Chapter 9 describes how we used SeSFJava in an introductory networking course, and summarizes our classroom experience. Chapter 10 illustrates the service specifications of Gnutella peer-to-peer and how we tested an open-source Gnutella implementation against these services. Chapter 11 concludes.

Chapter 2

Related Work

This discussion of related work is in three parts. First, we describe formalisms for design and verification of distributed systems. Second, we present tools that depend on runtime monitoring of programs. Third, we present tools and techniques for model checking of Java programs.

2.1 System modeling approaches

A rich set of formalisms has been developed in the past thirty years for compositional modeling and verification of concurrent systems. Compositionality, in general, requires two steps: (1) verifying the component programs of a composite system individually, and (2) constructing the properties of the composite system from the component system properties. Unlike SeSF, which adopts of layered compositionality, the formalisms presented in this section adopt traditional compositionality. We group those formalisms according to the mathematical and logical foundations adopted by them.

Temporal logic refers to all approaches for specifying temporal information within logic frameworks (i.e., well-formed formulas, axioms and inference rules). Temporal logic was first introduced by Rescher in 1971 [68], and Pnueli [64, 65] pioneered the use of temporal logic for reasoning formally about the properties of concurrent systems. Since then, various assertional methods based on temporal logic formalisms have been proposed.

Lynch and Tuttle introduced I/O automata [49]. An I/O automaton is a labeled transition system, which consists of a set S of states, a set A of actions divided into input, output and internal actions, and a set T of transitions ($\mathsf{S} \times \mathsf{A} \times \mathsf{S}$). Like SeSF, the semantics of an I/O automaton is described by executions and its external behaviors by traces. Unlike SeSF, input events of an I/O automaton are always enabled. Hence, if an input is to be not valid in certain states, the natural way to capture this is to have the action of the input check whether the current state is valid, and if not, transition appropriately (e.g., ignore input, go to an “error” state, etc.). But this requires the implementation to check the validity condition, which may be expensive (e.g., checking the primality of an input number). Also, unlike SeSF, input events can not return a value, and this prevents I/O automaton from modeling atomic read-modify-write constructs, which is an important class of synchronization constructs. A formal language for I/O automata is described in [25].

DIOA (Dynamic I/O Automata) is a process algebra for I/O automata. It extends I/O automata with the ability to change their signatures (states, transitions, actions) dynamically, and to create other I/O automata [5]. Similar to process algebra, DIOA uses parallel composition operator to check that the traces generated by an implementation automaton are equivalent to those generated by the specification automaton. The differences between DIOA and SeSF are the same as those between I/O automata and SeSF.

Unity [12], developed by Chandy and Misra, uses a rich set of temporal logic operators. A Unity program consists of a collection of guarded commands that are repeatedly selected and executed under some fairness constraints. Unlike SeSF, process interaction is based on shared variables rather than coupled actions. Thus, Unity has a single global state shared by all processes. In order to partition a Unity program into processes, instructions for partitioning are given outside the program. Proving the correctness of such a program requires one to prove the correctness of the abstract program and then to prove the correctness of the partitioning [12]. Unity also does not support dynamic creation and termination of processes.

Seuss [58] is the object-oriented extension of Unity. Similar to SeSF, it has the concept of a program, called box, and an instance, called clone. A Seuss program can be divided into sub-components that communicate with each other via procedural calls. The Seuss sub-components have no shared variables. Thus, compositionality proofs are easier in Seuss than in Unity. Similar to SeSF, Seuss specifies explicitly whether a procedure is callable by the environment or internally controlled. Unlike SeSF, when a caller calls a procedure while its enabling condition is not satisfied, the call is rejected and the caller tries again later till the call is accepted. Consequently, what is considered a fault in SeSF (calling an event with an unsatisfied enabling condition) is just a rejected event call in Seuss.

Temporal Logic of Actions (TLA) [45], developed by Lamport, uses primed expressions to indicate updates, e.g., $\mathsf{x}' = \mathsf{x}+1$ to denote $\mathsf{x} := \mathsf{x}+1$. Similar to Unity, TLA uses a single shared global state. Similar formalisms are developed by Manna & Pnueli [50, 51], Owicki & Lamport [61], Lam & Shankar [43], Back & Kurki-Suonio [6, 7] and Schneider

& Andrews [73].

Process algebra approaches model concurrency by using a collection of operators and algebraic representation of processes. The two archetypical process algebras are CSP (Communicating Sequential Processes) of Hoare [33, 69], and CCS (Calculus of Communicating Systems) of Milner [57]. In CSP, processes are defined by all finite behaviors, for example,

$$\mathsf{meet} = \mathsf{hi} \rightarrow \mathsf{talk} \rightarrow \mathsf{bye} \rightarrow \mathsf{STOP}$$

Recursion is used to define long (including unbounded) behaviors, e.g., $\mathsf{Clock} = \mathsf{tick} \rightarrow \mathsf{Clock}$. Unlike SeSF, CSP allows input actions to be blockable. Composition in CSP is quite complex; for example, CSP has both external nondeterminism ($\mathsf{S} = \mathsf{P} \Box \mathsf{Q}$ means that S can be P or Q depending on the environment's choice) and internal nondeterminism ($\mathsf{S} = \mathsf{P} \sqcap \mathsf{Q}$ means that S can be P or Q and the environment has no control on this choice). CCS defines similar operators and semantics. Both of them compose by synchronizing external actions. Both CSP and CCS check correctness by checking whether the traces generated by implementation process are equivalent to those generated by the specification process. They differ in how to calculate process equivalences. LOTOS is the most common specification language based on process algebras.

Finite State Machines (FSM) are usually used for verification of protocols. The most common variation of FSM is *Communicating FSM* (CFSM) [10, 63]: CFSM defines a tuple of machines, channels, initial set and transition systems; one per machine. SDL and Estelle are the popular specification languages of FSM models. FSM models are inadequate for specifying general programming models. This is because capturing the dynamics of any non-trivial program generally results in the state explosion problem.

2.2 Runtime monitoring of programs

Computer systems are often monitored during their execution for performance measurement, evaluation and enhancement, and debugging and testing [75]. We focus on monitoring for testing purposes.

One of the earliest systems is Anna (Annotated Ada) [71], which was developed to continuously monitor an executing Ada program for specification consistency. Anna annotations are inserted within the comments in Ada programs. Anna transforms the annotations into checking functions. It instruments calls to these functions into code areas that may cause specification violations.

MaC (Monitoring and Checking) framework [41, 46] provides assurance on the correctness of an execution of a system at run-time. MaC has two phases: before and during execution of the system. Before system execution, system requirements are formalized and monitoring scripts are constructed. Scripts instrument code into Java bytecode, and map from low-level information (e.g., variables changes) into high-level events (e.g., predicates). During run-time phase, the instrumented code extracts the low-level information and passes this information to a monitoring component. This component determines whether the generated events satisfy the formal specifications of the system. MaC is applied to a single system and does not support distributed systems. Although MaC does not handle progress assertions, it can be extended easily to do so as with SeSF.

Reference [14] tests multi-threaded applications by using DejaVu [4, 13], a capture/replay tool for the Jalapeno JVM (Java Virtual Machine) [3]. During execution of a Java program, DejaVu records all the thread switches that take place. DejaVu can replay the original thread schedule back and thus it can execute the original program deterministically. Thus, invariants can be tested whenever a thread switch takes place without needing an external module to ensure atomicity. As in previous approaches, this is limited to concurrent systems, not distributed ones. Further, it does not work with other JVMs.

Temporal Rover [21] is a specification based verification tool for applications written in C, C++, Java, Verilog and VHDL. It generates executable code from LTL (Linear Temporal Logic) and MTL (Metric Temporal Logic) assertions written as comments in the source code. These comments are compiled and linked as part of the application under test. During execution, the generated code validates the executing program against those specified assertions. Similar to MaC, it does not handle distributed systems.

Passive testing [55, 56] inserts *observers* at specific locations in CFSM models. One can determine the correctness of the protocol by checking generated executions at those observers.

Programmers' Playground [26] uses a language with formal semantics expressed in terms of I/O automata. The Playground separates communication from computations using I/O abstraction, which is a model of interprocess communication. A module defines three parts: data structures which are externally visible, reactive actions that start upon any change in the external data structures, and active actions that access other modules. The Playground compiles these modules, sets up the communication channels between each pair of processes that have common access to certain data. The Playground ensures atomic access to these external data structures. Playground has a *connection manager* which is a central runtime module that sets up the interprocess communication channels between all the defined Playground modules. Although Programmers' Playground is intended for implementation and not just testing, the system generated by the Playground is similar to how SeSFJava Harness works.

2.3 Model checking and theorem proving

Model checking is one of the techniques used to determine whether a system specification possesses a property expressed as a temporal logic formula. Model checking algorithms rely on state-space exploration in order to determine whether a system satisfies a temporal formula. Model checkers accept system specifications in some formal language L, for example, TLA+ or IOA. Next, they construct a finite state transition system which is checked against property P.

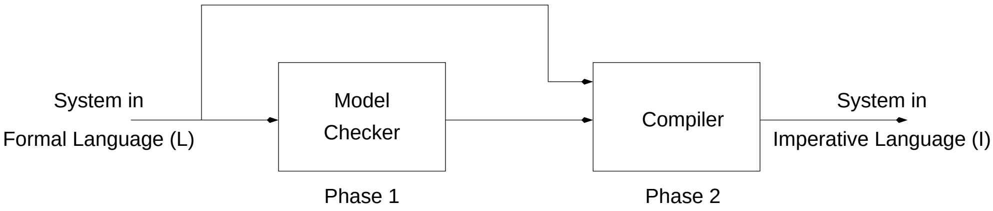


Figure 2.1: L-to-I category

To apply model checking to a program written in a conventional programming language, I, there are two techniques: L-to-I and I-to-L. L-to-I technique (figure 2.1) specifies the system using a formal language L, verifies the correctness of L using model checking tools, and then converts L to I using a compiler. One major disadvantage of this approach that it is very restrictive in the implementations it produces, and the resulting performance is questionable.

IOA-to-Java [84] is a compiler developed at MIT to compile programs written in IOA [25]. A user writes an algorithm in IOA, and then verifies that this algorithm satisfies its properties using tools available to I/O automata, e.g., theorem provers, simulators and model checkers. After the end of this phase, the algorithm is converted to Java using IOA-to-Java compiler.

A series of tools have been developed based on the CCS process algebra, called Concurrency Workbench [18] and Concurrency Factory [17]. These tools analyze systems expressed as CCS expressions, and include model checking, simulation and translation to C++ [28].

On the other hand, I**-to-**L technique (figure 2.2) constructs a finite state model that approximates the executable behavior of the software system of interest (phase 1). In phase 2, this finite state model is verified using one of the many model checkers. The major disadvantage of this approach is that extracting a faithful FSM (phase I) is very difficult. However, phase 2 can be automated. Most frameworks adopt this technique.

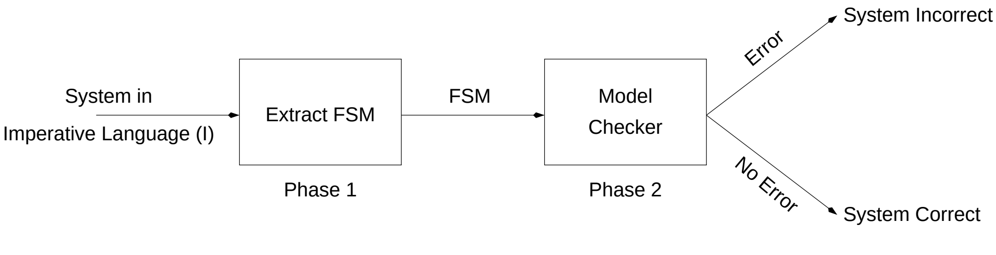


Figure 2.2: I-to-L category

Bandera enables the automatic extraction of compact finite-state models from program source code [20, 30]. It takes Java as input and generates a program model in the input language of one of the several existing verification tools. Bandera supports

SMV [54], **PVS** (guarded statements) [62, 70, 79] and **SPIN** [34] model checkers.

SAL (Symbolic Analysis Laboratory) [9] is a framework for combining different tools for program analysis, theorem proving and model checking toward the calculation of properties. The main part of the SAL is an intermediate language for specifying the concurrent systems. Translators extract transition systems from languages like Java, and convert those transition systems to SAL's intermediate language. Afterwards, the generated code is translated by the SAL environment to inputs to other systems, for example, PVS or SMV.

Java PathFinder [31] translates a given Java program to PROMELA [35] which is the input language to SPIN [34]. The generated PROMELA model has the same state space characteristics as the Java program; that is, it operates at the bytecode level (it emulates the bytecode).

There are various techniques to handle the state explosion problem which usually results when extracting a finite-state model from a Java program. These techniques involve hand-construction of models, which is expensive, prone to errors, and difficult to optimize.

Bandera and Java PathFinder alleviate the state explosion problem by eliminating components (classes, variables, code) that are not relevant to the property being verified; of course, identifying this is non trivial. For example, selecting a certain menu item is likely to be independent of the code. If the state explosion problem persists, the developer can limit the number of components or variables that participate in analysis, for example, bounding the number of objects that can be created.

Abstraction is used when some components contain more details than necessary for the property being verified. The range of such components can be abstracted to a smaller set [15, 11, 27]. For example, given two integers x and y and a property $x < y$, one can abstract the two integers by a boolean variable $z := x < y$, and thus it is represented by a boolean instead of two “practically unbounded” integers.

Static analysis of a program scans this program without executing it in order to construct state transition systems to be used in model checking [48]. *Runtime analysis* tools constructs transition systems from recorded executions, for example, Eraser [72]. Reference [74] adopts a technique that combines testing and abstraction. It first defines an interface I between two CSP processes which are tested against I. Then, it uses the generated runtime execution to abstract the model.

Theorem proving is the technique of finding a proof of a property from the axioms of the system, where both the system and its desired properties are expressed as formulae in some mathematical logic. Theorem proving can be combined with model checking to reduce the effect of state explosion, or to reduce the human intervention in the process. *Deduction* is used to construct valid finite-state abstractions of the system [80]: simple assertions can be deduced and proved using the theorem prover, for example, if predicate p satisfies state s and there is a transition R to state s', then predicate q satisfies s'. Thus, the checker needs not to explore states that have been already proven.

Reference [86] uses theorem proving to refine the abstractions applied to the programs. If an abstraction A of system S does not satisfy property p but no *concrete* counter example of S is generated, one can refine A to get another abstraction A' and recheck P. This process continues till abstraction A' satisfies p, or a concrete counter example of S is generated.

In summary, applying model checking to large software systems is an art. It needs experienced model builders who can abstract/eliminate most of the details of these programs, leaving only what is essential to verify a specific property.

Part I

SeSFJava

Chapter 3

SeSF Overview

SeSF is a framework for compositional design and implementation of concurrent systems. It formalizes the notions of processes, systems, services, system satisfying services, and compositionality. It uses temporal logic to specify safety and program assertions. It attempts to stay close to the programmer's world.

SeSF focuses on layered compositionality. Here, a composite system consists of layers of component systems, with services defining the allowed sequences of interactions between systems in different layers. Thus a system, in general, is "encapsulated" by services above and below. When component systems are composed to form a composite system, services between components become internal to the composite system and the remaining services encapsulate the composite system.

Roughly speaking, a system "satisfies" its encapsulating services if the interactions it initiates are allowed by the services, *assuming* the interactions initiated by the system's environment are allowed by the services. Given a system M and services U and V, we say M **satisfies** U **above and** V **below**, or as we prefer say, M **offers** U **uses** V, to mean that M is encapsulated by U above and V below and satisfies the services. Typically system M is a distributed system, and, U and V each is a distributed service.

Our **compositionality property** is that, given a composite system consisting of layers of component systems with services in between, if every component system in isolation satisfies its services, then the composite system as a whole satisfies its services.

3.1 Atomicity and the interleaving model

A key feature of SeSF is the explicit treatment of atomicity. When a process executes a statement, it affects its state (values of its data and its control) and also perhaps the state of another process. A statement is **atomic** if once a process starts executing it, the environment of the process cannot influence the execution or observe intermediate states. Thus, the atomic statements of a process define when its state can be altered or observed by other processes. Atomicity is essential to understanding a concurrent system, and yet most concurrent programming languages do not explicitly indicate atomicity in their specification. We emphasize that *executions of atomic statements can overlap in time*; that is, atomicity does not imply mutual exclusiveness in time, although the converse is true.

An **interaction** happens when a process executes an atomic statement that affects the state of another process; we say the first process does an **output** and the second process does an **input**.

Atomic execution of a statement implies that the statement appears to its environment to execute *instantaneously* at some point between the start and the end of the execution. This allows one to use the nondeterministic interleaving model of concurrent execution, in which the simultaneous execution of atomic statements is represented by the set of all possible sequential executions of atomic statements. Figure 3.1 illustrates this for two statements. The interleaving model, which permits the notion of global state, greatly facilitates reasoning about concurrent systems.

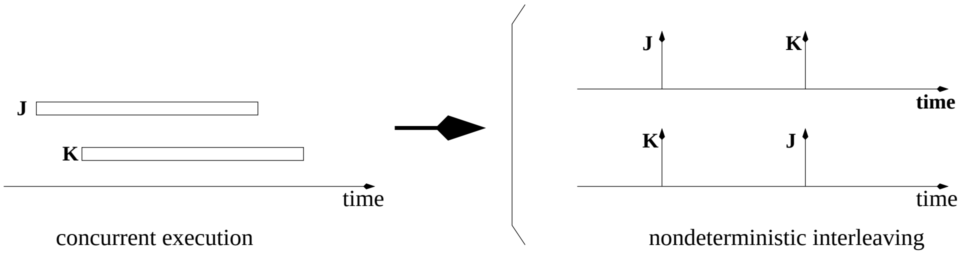


Figure 3.1: Concurrent execution modeled as non-deterministic interleaving

3.2 Systems

In SeSF, a system is a collection of processes that execute **system programs**. A system program can be in any concurrent programming language (e.g., Java, C/PThreads, C++/WinThreads), but it must make explicit the following (in our case, by inserting SeSF tags):

1. Atomically-executed statements.
2. Atomically-executed statements that are callable by the environment.
3. Fairness (or progress) expected from underlying platform.

We refer to atomically-executed statements as **events**. An event can be non-blocking (e.g., $\mathsf{x} = 4$) or blockable (e.g., $\mathsf{P(sem)}$). An event is either externally controlled, denoted

xc event, or locally controlled, denoted **lc event**, depending on whether its execution is initiated by the environment or by the system.

A system program has the form

```
system-program <name>( <parameters> ) {      // header
   <constants, types, variables, functions>    // can include constructor
   <externally-controlled events>              // xc events
   <progress assumptions>
}
```

The header indicates the system program's name and any parameters and their types. Constants, types, variables and functions are as in any procedural language. There can be a "constructor" for initializing variables and starting processes. The externally-controlled events are functions that can be called by the environment. The progress assumptions define the fairness expected of the underlying platform.

Functions can do all the usual things that concurrent programs can do: define variables and functions, update variables, call functions, create processes and start them executing, terminate processes, block on synchronization constructs (e.g., semaphore wait), and so on. They can also call xc events of *other* systems. An event call would be implemented by an interprocess communication facility such as TCP/IP, http exchange, remote procedure call, or a simple function call (if the two systems are threads of the same process).

Atomic statements are indicated by enclosing them in angled brackets or some other convention (e.g., a statement that "every memory read and memory write is atomic"). Every atomic statement corresponds to an **lc event**. It can make at most one event call in any execution. An lc event is said to be **enabled** if a process is at the event and the event, if it has a blocking condition, is not blocked. For example, a semaphore wait statement is enabled if a process is at the statement and the semaphore has a nonzero value.

Externally-controlled events An xc event has the form

```
xc-event <return type> <event name>( <event input parameters> ) { // header
   ec  <enabling condition predicate>      // not checked by system, no side effects
   ac  <action>                            // no event calls, no blocking
}
```

The header indicates the **event's signature**, similar to a function's signature, consisting of return type (which can be void), the event name, and event input parameters (if any) and their types. The **enabling condition** is a predicate in the program variables and parameters. We say an event call is **enabled** in a state if the event's enabling condition holds for the values of the program variables in this state and the parameters (if any) of the call. The **action** is the code that is executed when the event is called. The action has no event calls. It returns a value if <return type> is not void.

We next define the notion of safe event calls and safe event returns. An event call P.e(x) is **safe** if (1) it is **signature-consistent**, that is, system P exists, has e as an xc event, and the instantiated parameters x match the event's signature, and (2) the enabling condition of e(x) holds when the call is made. For a call of an xc event with non-void return, the return is **safe** if the value returned is of the return type.

It is the caller's responsibility, not the callee's, to ensure that the call is safe. The caller must determine this based solely on the event's signature and past interaction with the callee, since nothing else of the callee system is visible. *For a safe event call, the callee's responsibility is to execute the action atomically without blocking and, for an xc-with-return event, to do a safe return.* There is no obligation on the callee if the call is not safe. The callee is not obliged to check that the call is safe, but it can choose to do so in the action. Thus, the enabling condition is needed for analysis and testing only, and not for implementation.

We have imposed the above requirement that a safe event call be nonblocking because it simplifies the theory without any loss of generality. A blockable input operation, e.g., a semaphore wait operation, would be modeled in our formalism by two events, an xc event corresponding to initiating the operation, and an lc event corresponding to the return of the operation. This does not introduce more complexity; it merely makes explicit the inherent complexity of blockable input operations.

Progress assumptions Progress assumptions define the progress properties expected of the underlying platform in scheduling the processes, or equivalently, in executing its lc events. SeSF uses *weak fairness* and *strong fairness* [51].

- **wfair(**e**)** denotes **weak fairness** of event e. This means that if event e is *continuously enabled* beyond a certain point, it will eventually be executed.
- **sfair(**e**)** denotes **strong fairness** of event e. This means that if event e is *enabled infinitely often* beyond a certain point, it will eventually be executed.

Any collection of systems can be grouped to form a **composite** system. In addition to interactions with its environment, a composite system can also have internal interactions, that is, interactions between its components. Naturally, a component system can itself be a composite system.

Explicit and implicit lc events We refer to the lc events defined above as **implicit lc events**. A SeSF program can also have so-called **explicit lc events**, which have the form

```
lc-event <event name>(<event parameters>) { //header
    ec   <enabling condition>      // checked by system, no side effects
    ac   <action>                  // can have event calls
}
```

The header indicates the event name and any parameters and their types. There is no return value. The enabling condition, as in xc events, is a predicate in program variables and parameters, except that here it is checked by the system. Whenever the event is enabled, the action can be executed. The action should execute atomically and without blocking; thus the enabling condition is the only place to block the event. The action can have event calls.

Most conventional programming languages do not have built-in constructs corresponding to explicit lc events. To perform any activity, they have to create processes (or threads). Essentially, an explicit lc-event performs activity without identifying the working process. Explicit lc events are ideal for defining services and during system design, whereas processes would introduce needless structure and complications.

Semantics of systems An **execution** of a system is a sequence of event executions along with the states traversed, starting from an initial state. Each event execution is a transition. There are four kinds of transitions: **internal transitions** represent lc event executions in which no xc event is called, **input transitions** represent xc event executions, **output transitions** represent lc event executions that call xc events, and **fault transitions** represent event executions where an event encounters an undefined operation or an unsafe call to an xc event. A **faulty execution** of a system is an execution that ends in a fault transition. A fault is either a **locally-caused fault**, which happens if the system executes an undefined or non-terminating operation (e.g., division by zero, infinite loops, etc.), or an **externally-caused fault**, which happens if the environment makes an unsafe call of an event of the system. A **fault-free execution** of a system is an execution that contains no fault transitions; it can be finite or infinite. A **complete execution** of a system is a fault-free execution that satisfies the progress assumptions of the system.

3.3 Assertions

Assertions are a way of specifying properties of system executions. Assertions are divided into safety and progress assertions. So far the only assertions we have used are progress assertions, specifically, fairness assertions in the system specifications. Progress assertions are also the only kind of assertions we will use in our service specifications. Safety assertions are needed for reasoning about whether a system satisfies a service.

Predicates A **predicate** is a statement in first-order logic, i.e., involving the operators and ($\wedge$), or ($\vee$), implies ($\Rightarrow$), negation ($\neg$), and the quantifiers forall ($\forall$) and forsome ($\exists$). We are interested in predicates in variables and parameters of the programs about which we want to reason.

Safety assertions SeSF uses two kinds of safety assertions, namely, “invariant assertions” and “unless assertions”. These assertions impose conditions on the inter-event states of executions, not on intra-event states.

An **invariant assertion** has the form $\mathsf{inv}(\mathsf{P})$, where P is a predicate. $\mathsf{inv}(\mathsf{P})$ (read "invariant P") means that P always holds. Formally, $\mathsf{inv}(\mathsf{P})$ holds for an execution iff the execution is fault-free and every inter-event state in the execution satisfies P. $\mathsf{inv}(\mathsf{P})$ holds for a system iff it holds for every fault-free execution of the system and the system has no locally-caused faulty executions.

An **unless assertion** has the form P unless Q, where P and Q are predicates. It means that if P holds at some instant, then it continues to hold until Q holds. Formally, P unless Q holds for an execution iff the execution is fault-free and for every inter-event state in the execution that satisfies $\mathsf{P} \wedge \neg \mathsf{Q}$, either that state is the last state in the execution or the next inter-event state satisfies $\mathsf{P} \vee \mathsf{Q}$. P unless Q holds for a system iff it holds for every fault-free execution of the system and the system has no locally-caused faulty executions.

Progress assertions In addition to the fairness assertions described above, SeSF has two kinds of assertions for expressing progress properties of executions, namely, simple "leads-to" assertions and compound "leads-to" assertions. Like invariant and unless assertions, leads-to assertions do not state conditions on intra-event states.

A **simple leads-to assertion** has the form P $\mathsf{leadsto}$ Q, where P and Q are predicates. P $\mathsf{leadsto}$ Q means that if P holds at some instant, then Q holds at that instant or at some later instant. Formally, P $\mathsf{leadsto}$ Q holds for an execution iff the execution is fault-free and for every inter-event state in the execution that satisfies P, either that state satisfies Q or some later inter-event state satisfies Q. P $\mathsf{leadsto}$ Q holds for a system iff it holds for every *complete* execution of the system (i.e., execution that satisfies the system's fairness assumptions).

A **compound leads-to assertion** is a predicate with its terms replaced by leads-to assertions, for example, $[\forall$ integer n :: (X leadsto Y) $\Rightarrow$ (P leadsto Q)$]$. A compound leads-to assertion R holds for an execution iff the execution is fault-free and R evaluates to true after each simple leads-to assertion S in R is replaced by true or false depending on whether or not the execution satisfies S. R holds for a system iff it holds for every complete execution of the system. [We do not allow the underlying predicate to have ¬'s. This is to avoid assertions like ¬(P leadsto Q), which are really assertions about absence of progress.]

3.4 Services

In SeSF, a service defines the acceptable sequences of interactions between systems in different layers. A service is specified by a **service program**. The purpose of a service program is to:

- Specify the signatures of the system events on each side that are callable from the other side.

- Define the acceptable sequences of these event calls.

- Be directly usable in analysis and testing.

A service program has the form

```
service-program <name>(<parameters>) {
 <constants, types, variables, functions>
 <dnw events>
```

```
  <upw events>
  <progress obligations>
}
```

Events are divided into **downward events** (**dnw**) and **upward events** (**upw**). **Dnw** events **correspond** to xc events of the system below the service callable by the system above the service; xc events of the system below are **mapped** to the dnw events of the service. **Upw** events correspond to xc events of the system above the service callable by the system below the service; xc events of the system above are **mapped** to the upw events of the service. A service event has the form

```
dnw-event|upw-event  <return type><event name>(<event parameters>){ // header
 ec: <enabling condition predicate>
 ac: <action>                              // no event calls, no process creations
}
```

The header indicates the event's signature, consisting of the type (upw or dnw), return type (if any), event name, and parameters (if any) and their types. The event corresponds to an xc event with the same signature.

The progress obligations of a service define the progress that is expected in executing upw events. Service programs should not impose any progress obligations on dnw events. They have the form

```
progress-obligation <name>(<parameters>) {
  <progress assertions>
}
```

It is important to note that service programs have a different purpose than system programs. Service programs are intended not for execution, except in testing, but to provide an easily understandable definition of the service. Service programs can ignore all the constraints of the underlying platform, for example, resorting to system-wide updates and global history variables. Service programs are usually not executable on the underlying distributed platform, but they can be executed on a centralized platform.

Semantics of services The semantics of a service is similar to that of a system. An **execution** of a service is a sequence of event executions along with the states traversed. A service is not supposed to have any faulty executions. A **complete execution** is an execution which satisfies the progress obligations of the service.

3.5 Service satisfaction

In this section, we define what it means for a system to satisfy a service, whether as an offerer or as a user.

Consider a system M that is encapsulated by a service U above and a service V below. That is, every xc event of M visible to its environment corresponds to a dnw event of U or an upw event of V, and every event that M calls in its environment corresponds to an upw event of U or a dnw event of V. The inputs of M are all the possible calls of its xc events. The outputs of M are the possible calls it can make to xc events in its environment.

Definition An execution σ of M is **safe with respect** to U, abbreviated "safe wrt U", if the sequence of inputs and outputs in σ corresponds to that generated by some execution of U.

Definition An execution σ of M is **complete with respect** to U, abbreviated "complete wrt U", if the sequence of inputs and outputs in σ corresponds to that generated by some execution of U that satisfies U's progress obligations.

Execution σ being safe (complete) wrt V is similarly defined.

Definition M **offers** U **uses** V, also said as M **satisfies** U **above** V **below**, if

- **Safety**: For every finite execution x of M such that x is safe wrt U and V:
  - x is fault-free.
  - For every input call e of M: if $\mathsf{x} \circ \langle \mathsf{e} \rangle$ is safe wrt U and V, then e is enabled in the last state of x and its execution is fault-free and nonblocking. If e's execution returns a value, say g, then $\mathsf{x} \circ \langle \mathsf{e}, \mathsf{g} \rangle$ is safe wrt U and V.
  - For every execution y of M such that y is x extended by an internal or output transition: y is fault-free and safe wrt U and V.
- **Progress**: For every execution x of M such that x is safe wrt U and V: if x is complete wrt M and V, then x is complete wrt U.

Program-based formulation The above definition of service satisfaction provides compositionality. However, because it is stated in terms of event traces, it does not lend itself to program verification or testing techniques. We now provide an equivalent program-version of service satisfaction [78].

We first modify M, U and V, so that they interact with each other (rather than M interacting with its environment):

- Define the system M-wrt-$\{\mathsf{U},\mathsf{V}\}$ to be M with every output call $\mathsf{e}(\mathsf{x})$ changed to a call of the corresponding service event in U or V.
- Define system U-wrt-M to be U with the following changes:
  - For every event $\mathsf{e_U}(\mathsf{x})$ that corresponds to an output of M:
    - Change the event type (which would be "upw") to "xc".
  - For every event $\mathsf{e_U}(\mathsf{x})$ that corresponds to an input of M:
    - Change the event type (which would be "dnw") to "lc".
    - If $\mathsf{e_U}(\mathsf{x})$ has no return type, change the action to $\mathsf{e_U}(\mathsf{x}).\mathsf{ac};\mathsf{M}.\mathsf{e}(\mathsf{x})$.
    - If $\mathsf{e_U}(\mathsf{x})$ has a non-void return type, replace each returned value z in the action by code that generates a fault if z is not safe.
  - Set the progress assumptions to null.
- Define V-wrt-M to be the same as U-wrt-M except that U is replaced by V, "dnw" by "upw", and "upw" by "dnw".

Let M^* be the composite system of M-wrt-$\{\mathsf{U},\mathsf{V}\}$, U-wrt-M, and V-wrt-M. We have the following (proof in [78]):

- The safety condition for M offers U uses V holds iff M^* is fault-free.
- The progress condition for M offers U uses V holds iff M^* satisfies the progress assertion $\mathsf{V}.\mathsf{progress} \Rightarrow \mathsf{U}.\mathsf{progress}$.

Chapter 4

SeSFJava by Example

This section introduces SeSFJava with an extended example. The example, called AccountExample, consists of three parts (figure 4.1): a Bank system, one or more Client systems, and an Account service. Bank system offers Account service, while Client systems use this service. Each system (bank or client) is a process that can reside on a separate machine.

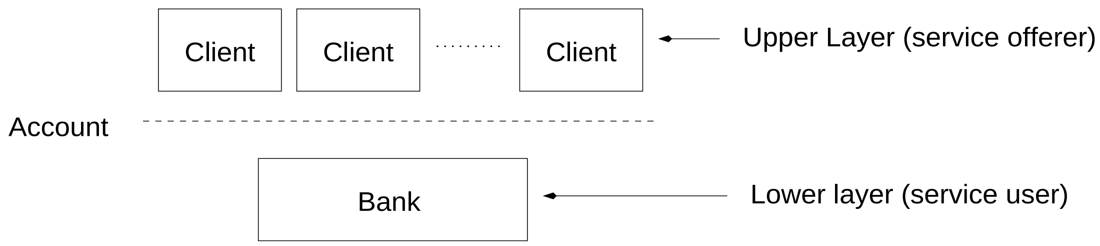


Figure 4.1: AccountExample: systems and services

The example involves three programs (figure 4.2): system program BankSystem, of which Bank system is an instance; system program ClientSystem, of which each Client system is an instance; and service program AccountService, of which Account service is an instance.

Each client is identified by a unique *id*, and resides at a *location* (RMI port name). All clients share an account maintained by the bank. A client can request the bank to update the account balance only if it has no request currently pending. The bank eventu-

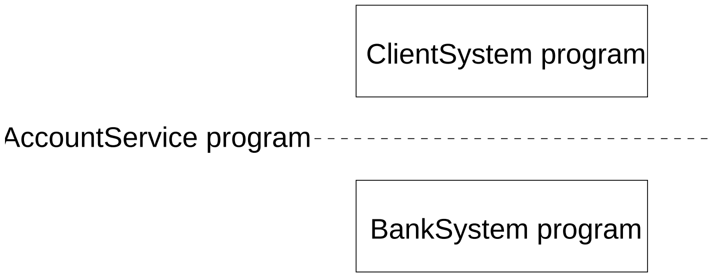


Figure 4.2: AccountExample: system and service programs

ally responds to every request. The response is an ack if the user has a valid id and the account balance is adequate; otherwise, the response is a nack. Account service defines the interactions between the clients (user system) and the bank (offerer system).

Section 4.1 describes the bank and client system programs. Section 4.2 describes the composite system of bank and clients. Section 4.3 describes Account service. Section 4.4 illustrates the event-trace version of the conditions for Bank system to satisfy Account service and for Client system to satisfy Account service. Section 4.5 illustrates the program version of these same conditions.

4.1 Bank and client system programs

A SeSFJava system program is a Java program with a specific structure indicated by SeSFJava tags inserted in the program. SeSFJava tags are special cases of Java comments; specifically, they have the prefix "//#", where the "//" denotes the start of a Java comment. Thus, a SeSFJava program can be treated just like a Java program; it can be compiled and executed by any Java platform without any modifications. In the case of testing, the SeSFJava Harness preprocesses the SeSFJava tags and produces modified Java files.

Consider BankSystem program (figure 4.4 on page 50). It has the following kinds of SeSF tags:

- Tags of the form "//# system_program;" precede and identify the system program, in this case, the system program class BankSystem.

- Tags of the form "//# xc_event;" precede and identify the xc events of the program. There is one xc event, namely update(id, n, location), indicating that the user associated with id and location requests to update the balance by value n.

- Tags of the form "//# ec: <predicate>;" specify the enabling condition of the associated event. For example, xc event update is enabled if $0 \leq \mathsf{id} < \mathsf{N}$ and the user has no pending requests. An enabling condition must always evaluate to true or false; it is not allowed to terminate abruptly, for example, throw an exception.

- Tags involving harness (e.g., "//# harness", "//# breakpoint", etc.) are relevant for testing and will be explained later.

BankInterface (figure 4.6) is a Java interface that indicates the xc event signatures of BankSystem.

When BankSystem is executed, it binds to an RMI port called "Bank" and waits for update requests from client. Every incoming update request starts a new thread in the bank system. Specifically, the statement new UpdateThread(id, n).start() in BankSystem's xc event update creates an instance of UpdateThread and starts executing method run. The JVM should, supposedly, ensure weak fairness for all created threads.

ClientSystem program (figure 4.5 on page 51) is organized in a similar fashion. It has two xc events: ack(id), called by the bank to indicate acceptance of the client's update request; and nack(id), called by the bank to indicate rejection of the client's update request. When executed with parameter id, it first binds to an RMI port called "Client$<$ id $>$". It then repeatedly issues update requests to the bank, specifically, update(id, n, "Client<id>") where n is a random number in the range $[-40, 40]$. To keep the example short, the client's location has the form "Client<id>". An arbitrary location could be chosen but the Bank would have to implement a hash table to map locations to ids.

ClientInterface (figure 4.7 on page 52) is a Java interface that indicates the xc event signatures of ClientSystem.

4.2 Composite system of Bank and Clients

Bank is a process that is created by executing the command-line "java $\cdots$ BankSystem". It binds to a specific port, namely "Bank", using the RMI rebind command. Clients know this port and hence can interact with the bank via RMI methods defined in BankInterface. Each client is a process that is created by executing command-line "java $\cdots$ ClientSystem <id>". A client does a lookup for RMI port "Bank", and binds itself to a port "Client<id>" using RMI rebind command. The client then starts updating the balance account by repeatedly executing update(id, n, "Client<id>"), where n is a random number. Bank uses RMI port "Client<id>" for the callback methods, namely ack and nack. Figure 4.3 illustrates such a composite system.

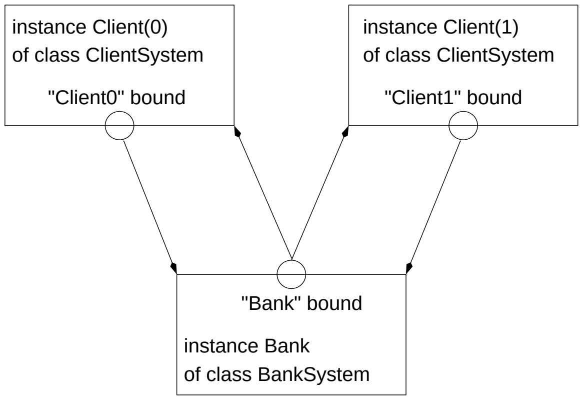


Figure 4.3: Composite system of bank and two clients

4.3 Account service program

The AccountService service program (figure 4.9 on page 53) defines the permissible interactions between client systems (users of the service) and bank system (offerer of the service). Specifically, it defines the signatures of the interactions and the permissible sequences of interactions (i.e., their safety and progress properties)

The program defines three events: update, ack and nack. The signature of each service event is the same as that of the corresponding xc event. Each service event is preceded by a tag indicating the system of the corresponding xc event. So the tag "//# dnw:BankSystem;" preceding event update indicates that dnw event AccountService.update is mapped to xc event BankSystem.update, and that they both have the same signature (for brevity, we refer to the dnw event as AccountService.update rather than the more accurate AccountService.BankSystem.update). Similarly, the tag "//# upw:ClientSystem;" preceding event ack indicates that upw event AccountService.ack is mapped to xc event ClientSystem.ack, and that they both have the same signature. Note that no event creates threads or processes.

Informally, Account service requires the sequence of interactions to satisfy the following properties:

- **Safety:** A client has at most one update request pending. An update request must have a valid id. The bank issues an ack to a client only if the client has a pending update request with a valid id and, in case of negative update, the balance is adequate.
- **Progress:** Every update request is eventually acked or nacked.

Interface AccountInterface (figure 4.8 on page 52) defines the headers of all the methods available in AccountService.

4.4 Service satisfaction conditions: event-trace conditions

We first note that Bank is encapsulated above by Account: that is, the xc event of Bank corresponds to a dnw event in Account, and every output of Bank corresponds to a call of a upw event of Account.

We next give the event-trace version of the conditions for the Bank to satisfy Account as offerer:

- **Safety condition** For every finite execution σ of Bank that is safe wrt Account:
  - If $\sigma \circ \mathsf{Account.update}(\mathsf{id}, \mathsf{n}, \mathsf{loc})$ is safe wrt Account,

    then $\mathsf{Bank.update}(\mathsf{id}, \mathsf{n}, \mathsf{loc})$ is enabled at the end of σ and its execution is well-formed.
  - If a Bank thread is at a statement,

    then the execution of the statement is well-formed.

  - If a Bank thread is at client[id].ack(id) at the end of σ, then $\sigma \circ$ Account.ack(id) is safe wrt Account.
  - If a Bank thread is at client[id].nack(id) at the end of σ, then $\sigma \circ$ Account.nack(id) is safe wrt Account.

- **Progress condition**: For every execution σ of Bank that is safe wrt Account, if σ satisfies Bank progress assumptions (i.e., weak fairness of all Bank threads) then σ is complete wrt Account (i.e., satisfies pending[i] leadsto $\neg$pending[i] for every i).

We next give the event-trace version of the conditions for the Client to satisfy Account as user:

- **Safety condition** For every finite execution σ of Client that is safe wrt Account:
  - If $\sigma \circ$ Account.ack(id) is safe wrt Account, then Client.ack(id) is enabled at the end of σ and its execution is well-formed.
  - If $\sigma \circ$ Account.nack(id) is safe wrt Account, then Client.nack(id) is enabled at the end of σ and its execution is well-formed.
  - If a Client thread is at a statement, then the execution of the statement is well-formed.
  - If a Client thread is at bank.update(id, r.nextInt(80) $-$ 40, "Client" $+$ id) at the end of σ, then $\sigma \circ$ Account.update(id, r.nextInt(80) $-$ 40, "Client" $+$ id) is safe wrt Account.
- **Progress condition**: Null (because Account service does not impose any progress requirements on Client system).

Although we do not do so here, it would be straightforward to prove by operational reasoning that these conditions hold.

4.5 Service satisfaction conditions: program version

As mentioned earlier, the event trace conditions given above cannot be directly tested. We now give the program version of the service satisfaction conditions.

Developing the conditions for Bank to offer Account involve the following steps: (1) constructing Bank-wrt-Account from Bank, (2) constructing Account-wrt-Bank from Account, and (3) constructing the assertions to be satisfied by composite system Bank* consisting of Bank-wrt-Account and Account-wrt-Bank. The above steps are described in sections 4.5.1, 4.5.2 and 4.5.3, respectively.

Developing the conditions for Client to offer Account involve the following steps: (1) constructing Client-wrt-Account from Client, (2) constructing Account-wrt-Client from Account, and (3) constructing the assertions to be satisfied by composite system Client* consisting of Client-wrt-Account and Account-wrt-Client. The above steps are described in sections 4.5.4, 4.5.5 and 4.5.6, respectively.

4.5.1 Constructing Bank-wrt-Account

We construct Bank-wrt-Account from Bank as follows (the complete code is given in appendix A.1):

- System name Bank is changed to Bank-wrt-Account.

- For every xc event e:

  - Change its action to if(!e.ec) then fault; else e.ac;.
  - Change its enabling condition to true.

  In particular, xc event update is changed to:

  ```
  //# xc_event;
  void update(int id, int n, String location) throws RemoteException{
    synchronized(lock){
      //# ec: true;
      if (!(id >= 0 && id < N && client[id] == null ))
        throw new Error("Bank.update enabling failed");
      try {client[id] = (ClientInterface) Naming.lookup(location);}
      catch (Exception e) {e.printStackTrace();}
      new UpdateThread(id, n).start();
    }
  }
  ```

- Every output call in Bank is replaced by a call to the corresponding event of Account. This is done implicitly because Bank determines the location of the callee (which is Account) via parameter *location* of event update.

4.5.2 **Constructing** Account-wrt-Bank

We construct Account-wrt-Bank from Account as follows (the complete code is given in appendix A.2):

- Account service is changed to Account-wrt-Bank system.
- For every upw event:
  - Change the event type to “xc”.
  - Change its action to if(!e.ec) then fault; else e.ac;.
  - Change its enabling condition to true.

In particular, upw events ack and nack are transformed to:

```
//# xc_event;
synchronized public void ack(int id) throws RemoteException {
  //#ec: true;
  if (!(id >= 0 && id < N && pending[id] &&
        (amount[id] >= 0 || balance >= −amount[id])))
    throw new Error();
  pending[id] = false;
  balance += amount[id];
}

//#xc_event;
synchronized public void nack(int id) throws RemoteException {
  //# ec: true;
  if (!(id >= 0 && id < N && pending[id] && balance < −amount[id]))
    throw new Error();
  pending[id] = false;
}
```

- For every dnw service event e:
  - Change the event type to “lc”.
  - Augment its action by a call to the corresponding xc event.

In particular, dnw event update is transformed to

```
lc-event synchronized void update(int id, int n, String location)
    throws RemoteException {
  ec: id >= 0 && id < N && !pending[id];
  ac: amount[id] = n;
      pending[id] = true;
      bank.update(id, n, location);     // corresponding system event.
}
```

Because Java does not have an explicit lc-event construct, this lc event is implemented in Java as follows:

1. Create function update by removing the event’s “lc” construct and its enabling condition. So, lc event update is changed to:

```
synchronized void update(int id, int n, String location) throws RemoteException {
  amount[id]  = n;
  pending[id] = true;
  bank.update(id, n, location);
}
```

2. Create a thread, which we call whirl, that repeatedly checks the enabling condition of this lc event, and executes its action (which is method update) whenever its enabling condition holds. This thread is created manually. For event update, create:

```
class whirl extends Thread {
  int id;
  whirl(int id){
    this.id = id;
  }

   public void run(){
     while(true) {
       synchronized(lock){     // ensures atomicity of this block
         if (id >= 0 && id < N && !pending[id]) {
           update(id, r.nextInt(80) - 40, "Account");
         }
       }
       yield(); //allows other thread to proceed
     }
   }
}
```

4.5.3 Conditions on Bank*

Define Bank* to be the composite system consisting of systems Bank-wrt-Account and Account-wrt-Bank. The safety condition for Bank offers Account is that Bank* is fault-free. Faults in Bank* arise from calling a disabled event or executing an undefined operation (division by zero, signature-inconsistent call, etc.) This reduces to the following conditions:

- Bank* statements do not have undefined values or operations.

- Bank* satisfies inv(Account.update.ec ⇒ Bank.update.ec).

- Bank* satisfies inv(Bank′s thread is at client[id].ack(id) ⇒ Account.ack.ec).

- Bank* satisfies inv(Bank′s thread is at client[id].nack(id) ⇒ Account.nack.ec).

The progress condition holds iff Bank* satisfies assumption pA (figure 4.9) assuming weak fairness of Bank's threads.

Although we do not do so here, it would be straightforward to prove by assertional or operational reasoning that these conditions hold.

4.5.4 Constructing Client-wrt-Account

We construct Client-wrt-Account from Client as follows:

- System name Client is changed to Client-wrt-Account.

- Change the xc events ack and nack to:

```
//# xc_event;
void ack(int id) throws RemoteException{
  //# ec: true;
  synchronized(lock){
    wait = false;
    lock.notify();
  }
}

//# xc_event;
void nack(int id) throws RemoteException{
  //# ec: true;
  synchronized(lock){
    wait = false;
    lock.notify();
  }
}
```

- Every output call in Client is replaced by a call to the corresponding event of Account. This is done implicitly because Client determines the location of the bank via lookup call.

4.5.5 Constructing Account-wrt-Client

We construct Account-wrt-Client from Account as follows:

- Account service is changed to Account-wrt-Client system.

- For every dnw event, change it similar to upw events in constructing Account-wrt-Bank. Dnw event update is transformed to:

```
//# xc_event;
synchronized public void update(int id, int n, String location)
    throws RemoteException {
  //#ec: true;
  if (!(id >= 0 && id < N && !pending[id]))
     throw new Error();
  pending[id] = false;
  balance += amount[id];
}
```

- For every upw service event e, change it similar to dnw events in constructing Account-wrt-Bank.

  1. Create functions ack and nack:

```
synchronized void ack(int id) throws RemoteException {
   pending[id] = false;
   balance += amount[id];
   client.ack(id);
}

synchronized void nack(int id) throws RemoteException {
   pending[id] = false;
   client.nack(id);
}
```

2. Create a thread, which we call whirl, that repeatedly checks the enabling condition of these two events, and executes their action whenever its enabling condition holds. This thread is created manually. For event update, create a thread:

```
class whirl extends Thread {
  int id;
  whirl(int id){
    this.id = id;
  }

  public void run(){
    while(true) {
      synchronized(lock){            // ensures atomicity of this block
        if (id >= 0 && id < N && pending[id] &&
            (amount[id] >= 0 || balance >= -amount[id])) {
          ack(id);
        }

        if (id >= 0 && id < N && pending[id] &&
            balance < -amount[id]) {
          nack(id);
        }
      }
      yield(); //allows other thread to proceed
    }
  }
}
```

4.5.6 Conditions on Client*

Define Client* to be the composite system consisting of systems Client-wrt-Account and Account-wrt-Client. The program-version conditions reduce to the following conditions:

- Client* statements do not have undefined values or operations.
- Client* satisfies inv(Account.ack.ec $\Rightarrow$ Client.ack.ec).
- Client* satisfies inv(Account.nack.ec $\Rightarrow$ Client.nack.ec).

```
import java.rmi.*;
import java.rmi.server.*;

//# system_program;
class BankSystem extends UnicastRemoteObject implements BankInterface {
  //# static HarnessInterface harness;
    static int balance;
    static final int N = 10;                                  // number of clients
    static Object lock = new Object();                        // for atomicity
    static ClientInterface client[] = new ClientInterface[N];  // client[i] is null if it has no pending requests.
  BankSystem() throws RemoteException {}

  public static void main(String argv[]) throws Exception {
    //# harness = (HarnessInterface) Naming.lookup(”AccountHarness”);
    Naming.rebind(”Bank”, new BankSystem());
  }

  //# xc_event;
  public void update(int id, int n, String location) throws RemoteException {
    synchronized(lock){
      //# ec: id >= 0 && id < N && client[id] == null;
      try { client[id] = (ClientInterface) Naming.lookup(location); }
      catch(Exception e){ e.printStackTrace();}
      new UpdateThread(id, n).start();
    }
  }

  class UpdateThread extends Thread {
    int id, n;
    UpdateThread(int id, int n) {
      this.id = id;
      this.n = n;
    }
    public void run(){
      try {
        //# breakpoint(”Bank.bpBegin”, BEGIN);
        synchronized(lock){
          if (n >= 0 || balance >= -n) {
            balance + = n;
            client[id].ack(id);
          } else
            client[id].nack(id);
          client[id] = null;
        }
        //# breakpoint(”Bank.bpEnd”, END);
      } catch (RemoteException re) { re.printStackTrace(); }
    }
  } //End Thread
} //End System
```

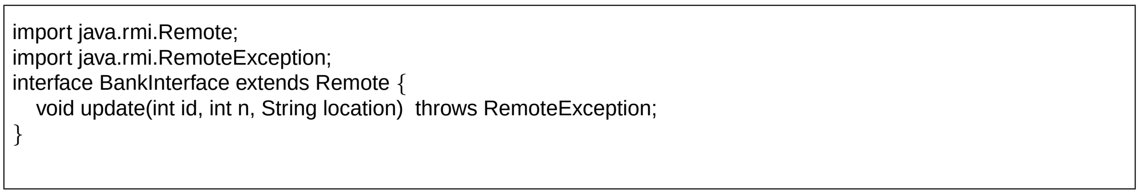
Figure 4.4: BankSystem system program (file BankSystem.java)

- Client* satisfies inv(Client′s thread is at bank.update(id, . . .) $\Rightarrow$ Account.update.ec).

There are no progress conditions.

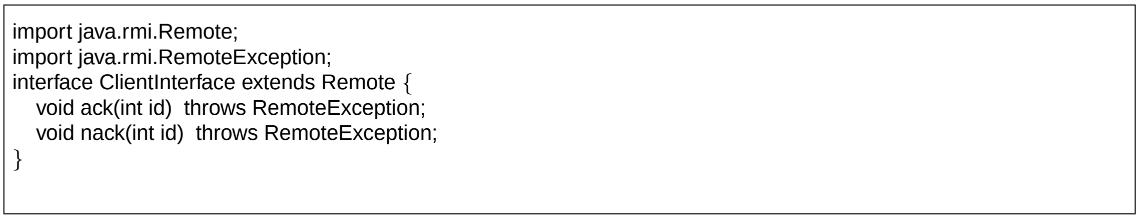
```
import java.rmi.*;
import java.rmi.server.*;

//# system_program;
class ClientSystem extends UnicastRemoteObject implements ClientInterface {
   //# static HarnessInterface harness;
     Object  lock = new Object();            // for atomicity
     Random  r = new Random();               // random number generator
     boolean wait = false;                   // true if it has pending requests, false otherwise
   ClientSystem() throws RemoteException { }
   public static void main(String argv[]) throws Exception {
      if (System.getSecurityManager() == null )
         System.setSecurityManager(new RMISecurityManager());
      //# harness = (HarnessInterface) Naming.lookup("AccountHarness");
      ClientSystem client = new ClientSystem();
      client.execute(Intger.parseInt(argv[0]));
   }

   void execute(int id) throws Exception {
      BankInterface bank = (BankInterface) Naming.lookup("Bank");
      Naming.rebind("Client" + id, this);
      for( int i = 0; i < 50; i++){
        //# breakpoint("Client.bpInc", MANUAL);
        wait = true;
        bank.update(id, r.nextInt(80) - 40, "Client" + id);
        // Wait for ack or nack
        synchronized(lock){
          while (wait){
            //# breakpoint("Client.bpWait", WAIT);
            lock.wait();
          }
        }
      }
      //# breakpoint("Client.bpEnd", END);
   }

   //# xc_event;
   public void ack(int id) throws RemoteException {
      //# ec: true;
      synchronized(lock){
         wait = false;
         lock.notify();
      }
   }

   //# xc_event;
   public void nack(int id) throws RemoteException {
      //# ec: true;
      synchronized(lock){
         wait = false;
         lock.notify();
      }
   }
}
```

Figure 4.5: ClientSystem system program (file ClientSystem.java)



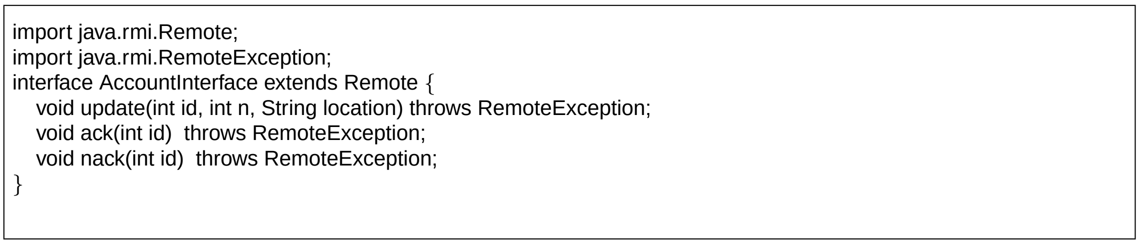
```
import java.rmi.Remote;
import java.rmi.RemoteException;
interface BankInterface extends Remote {
   void update(int id, int n, String location)  throws RemoteException;
}
```

Figure 4.6: BankInterface interface (file BankInterface.java)



```
import java.rmi.Remote;
import java.rmi.RemoteException;
interface ClientInterface extends Remote {
   void ack(int id)  throws RemoteException;
   void nack(int id)  throws RemoteException;
}
```

Figure 4.7: ClientInterface interface (file ClientInterface.java)



```
import java.rmi.Remote;
import java.rmi.RemoteException;
interface AccountInterface extends Remote {
   void update(int id, int n, String location) throws RemoteException;
   void ack(int id)  throws RemoteException;
   void nack(int id)  throws RemoteException;
}
```

Figure 4.8: AccountInterface interface (file AccountInterface.java)

```
import java.rmi.*;
import java.rmi.server.*;

//# service_program;
class AccountService extends UnicastRemoteObject implements AccountInterface {
   //# Harness harness;
     static final int N = 10;                          // number of clients
     int balance;
     boolean  pending[] = new boolean[N];      // pending[i] is false if it has no pending request
     int  amount[] = new int[N];                     // amount[i] is the update value of user i last request
   AccountService() throws RemoteException {
      try {
        Naming.rebind("AccountHarness", this);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

  //# dnw: BankSystem;
  synchronized public void update(int id, int n, String location) throws RemoteException {
     //# ec: id >= 0 && id < N && !pending[id];
        amount[id]  = n;
        pending[id] = true;
     }
  }

  //# upw: ClientSystem;
  synchronized public void ack(int id) throws RemoteException {
     //# ec: id >= 0 && id < N && pending[id] && (amount[id] >= 0 || balance >= -amount[id]);
        pending[id] = false;
        balance  + = amount[id];
  }

   //# upw: ClientSystem;
   synchronized public void nack(int id) throws RemoteException {
      //# ec: id >= 0 && id < N && pending[id] && balance < -amount[id];
         pending[id] = false;
   }

   //# progress_obligation pA {
   //# forall i: 0 − > (N-1)
   //#   beginAssertion
   //#   pending[i] leadsto !pending[i]
   //#   endAssertion
   //# endfor
   //# }
}
```

Figure 4.9: AccountService service program (file AccountService.java)

Chapter 5

SeSFJava Harness by Example

This chapter introduces the SeSFJava Harness by applying it to the Account example. As mentioned earlier, the program-based conditions of service satisfaction give us a way to mechanically test a system against services. To test Bank against Account, we proceed as follows:

1. Create a Harness process to control the execution. The Harness is a process that resides on an arbitrary machine. In our example, the Harness is bound to an RMI port, namely "AccountHarness". The Harness has interface HarnessInterface (figure 5.2).

2. Construct Bank-wrt-Account$'$, a version of Bank-wrt-Account that interacts with the Harness.

3. Construct Account-wrt-Bank$'$, a version of Account-wrt-Bank that interacts with the Harness.

4. Execute composite system Bank$^{*\prime}$ (figure 5.1), consisting of Bank-wrt-Account$'$ and Account-wrt-Bank, along with the Harness, and check whether the generated

execution becomes faulty.

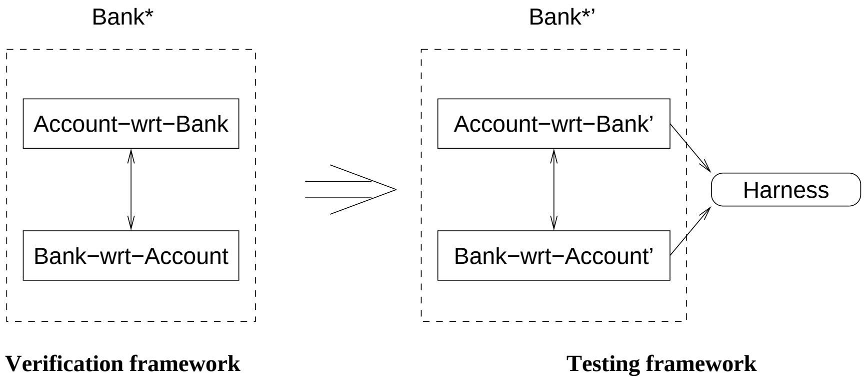


Figure 5.1: Bank* and Bank$^{*\prime}$ composite systems.

Sections 5.1 and 5.2 describe how to obtain Bank-wrt-Account$'$ and Account-wrt-Bank, respectively. Section 5.3 describes how to obtain a testing platform on which Bank$^{*\prime}$ can be executed. Section 5.4 describes how to execute Bank$^{*\prime}$.

5.1 Constructing Bank-wrt-Account$'$

We construct Bank-wrt-Account$'$, referred to as Bank$'$, from Bank-wrt-Account (described in section 4.5.1) as follows:

- Tags

  "//# static HarnessInterface harness;" and

```
import java.rmi.Remote;
import java.rmi.RemoteException;
interface HarnessInterface {
   void printlnLog (String str) throws RemoteException;
   void printLog (String str) throws RemoteException;
   void checkAssertions(boolean debugInfo) throws RemoteException;
   void breakpoint(String name, int mode) throws RemoteException;
}
```

Figure 5.2: HarnessInterface interface (file HarnessInterface.java)

“//# harness = (HarnessInterface) Naming.lookup(”AccountHarness”);”

indicate the location of the Harness.

- For every xc event e:
  - Insert a call to method checkAssertions(), which sends data necessary for assertion checking to Harness module.
  - Log information to the log file.

  So change xc event update to:

```
//# xc_event;
public void update(int id, int n, String location)
    throws RemoteException{
  harness.log.print(...); // log event execution
  checkAssertions();     // check the validity of any assertions.
  synchronized(lock){
    //# ec: true;
    if (!(id >= 0 && id < N && client[id] == null))
      throw new Error(”Bank.update failed”);
    try { client[id] = (ClientInterface) Naming.lookup(location); }
    catch (Exception e){ e.printStackTrace(); }
    new UpdateThread(id, n).start();
  }
}
```

- Breakpoints are called to indicate transition of systems. Insert breakpoints at locations specified by tag //#breakpoint. Breakpoints will be explained later in this section.

5.2 Constructing Account-wrt-Bank′

We construct Account-wrt-Bank′, referred to as Account′, from Account-wrt-Bank (described in section 4.5.2)as follows:

- For every upw event, insert a call to checkAssertions, and log information to log file. Upw events ack and nack are changed to:

```
//# xc_event;
synchronized public void ack(int id) throws RemoteException {
  //# ec: true;
  harness.log.print(...);   // log event execution
  checkAssertions();
  if (!(id >= 0 && id < N && pending[id] &&
      (amount[id] >= 0 || balance >= −amount[id])))
     throw new Error(”ack --- fault”);
   pending[id] = false;
   balance += amount[id];
}

//# xc_event;
synchronized public void nack(int id) throws RemoteException {
  //# ec: true;
  harness.log.print(...);   // writing info to log file
  checkAssertions();
  if (!(id >= 0 && id < N && pending[id] && balance < −amount[id]))
     throw new Error(”nack --- fault”);
  pending[id] = false;
}
```

- Recall that every dnw event should be transformed to an lc event. We handled this situation by constructing a method update and thread whirl. In addition to this, the following modifications have to take place:

  - In method update, insert a call to checkAssertions, and log event execution. So, change method update to:

    ```
    synchronization void update(int id, int n, String location) throws RemoteException {
      harness.log.print(...);   // writing info to log file
      checkAssertions();
      amount[id]  = n;
      pending[id] = true;
      bank.update(id, n, location);
    }
    ```

  - In thread whirl, insert breakpoints at necessary locations. So, thread whirl changes to:

```
class whirl extends Thread {
  int id;
  whirl(int id){
    this.id = id;
  }

  public void run(){
    breakpoint(”whirl.Begin”, BEGIN);
    while(true) {
      synchronized (lock) {
        if (id >= 0 && id < N && !pending[id]) {
          breakpoint(”whirl.implicitLc”, AUTOMATIC);
          update(id, r.nextInt(80) - 40, ‘‘Account’’);
        }
      }
      yield(); //allows other threads to continue
    }
  }
}
```

5.3 Constructing testing platform

Once composite system Bank$^{*\prime}$ consisting of Bank$'$ and Account$'$ is constructed, the next step is to obtain a **testing platform** on which it can be executed. This is not trivial because the atomicity requirements of Bank$^{*\prime}$ are usually much more stringent than those of Bank*.

Let I refer to the platform on which Bank is intended to execute; that is, Bank's program involves I-specific constructs for IO, communication, synchronization, concurrency, and so on. Because Bank$'$ is obtained by a simple redirection of Bank's output calls, Bank$'$ also must be executed on I. However, I invariably cannot ensure atomicity of the interactions between Bank$'$ and other components in the system (e.g., Account$'$). This is because Account, and hence Account$'$, makes use of more powerful atomicity than is intrinsically provided by I. Thus I alone cannot serve as a testing platform.

We need to augment I so that Bank′-Account′ interactions are executed atomically. SAC (Serializer And Checker) module within the Harness is introduced to solve this problem. In order to conform to the interleaving model, SAC ensures that only one thread is proceeding at a time. Every thread within the composite system is associated with a lock. When the lock is released, the thread proceeds. When the lock is revoked, the thread is paused. SeSFJava Harness inserts breakpoints in Bank′ and Account′ such that at any time, at most one thread of Bank*′ runs and every other thread is paused at a breakpoint. SAC module maintains relevant state for every process, such as whether the process is running, paused, blocked, or about to be terminated. Our solution is based on the following steps:

- Whenever a thread is created in Bank*′, it provides its relevant state to the SAC (by calling breakpoint(BEGIN)) and pauses. For example, //# breakpoint(”Bank.bpBegin”, BEGIN); in

  BankSystem.UpdateThread.

- Whenever a thread encounters a breakpoint during its execution, it provides its relevant state to the SAC and pauses.

- Whenever a thread is paused, SAC module chooses one thread from the paused ones to proceed. This thread is selected either automatically or manually by the user. Other operations can take place during the execution, for example, listing unsatisfied assertions so far.

- If the thread is about to execute a blocking statement, it informs the SAC module (by calling breakpoint(WAIT)). When SAC module receives this breakpoint call, it allows the calling thread to proceed to the block procedure, and then it chooses another thread to proceed. When the blocked thread unblocks, it informs the SAC module and goes to pause state. For example, //# breakpoint("Client.bpWait", WAIT); in ClientSystem.execute.

- Whenever a thread is (about to be) terminated, it provides its relevant state to SAC module (by calling breakpoint(END)) and terminates. For example, //# breakpoint("Bank.bpEnd", END); in BankSystem.UpdateThread.

The serializer-based approach is rather conservative (because it prevents parallel execution of processes). However, it is simple and, as we shall see, easily provides the snapshots needed to check assertions.

Assertions are evaluated at **checking locations**, specifically, at the start of every event and at every breakpoint. For example, the scheme to test if Bank satisfies assertion inv(Bank.balance $>=$ 0) is as follows. First, whenever Bank$'$ encounters a checking location, it sends *Bank.balance* to the Harness (via method checkAssertions). Second, whenever the Harness receives this field, it checks whether the predicate Bank.balance $>=$ 0 holds. If the predicate fails once, then the invariant does not hold.

5.4 Testing and GUI

After construction of Bank$^{*\prime}$, it is executed on the same platform as Bank* as follows:

1. SeSFJava Harness is started as a separate process, binds itself to RMI port "AccountHarness".

2. Bank' process is created. It looks up for port “AccountHarness” using RMI lookup command.

3. $\mathsf{Account}'$ process is created, and looks up for harness’ port. So, both systems are hooked up with SeSFJava Harness.

4. The developer can choose to work in batch mode, where he/she leaves the execution to run for a while, and then analyze the log file. Or, he can influence the flow of the execution manually.

Figure 5.3 shows a snapshot of the Harness’ GUI interface during the testing of the AccountExample in the chapter 4. **Thread Panel** displays the current set of threads stopping at breakpoints. The set of breakpoints are displayed in two ways:

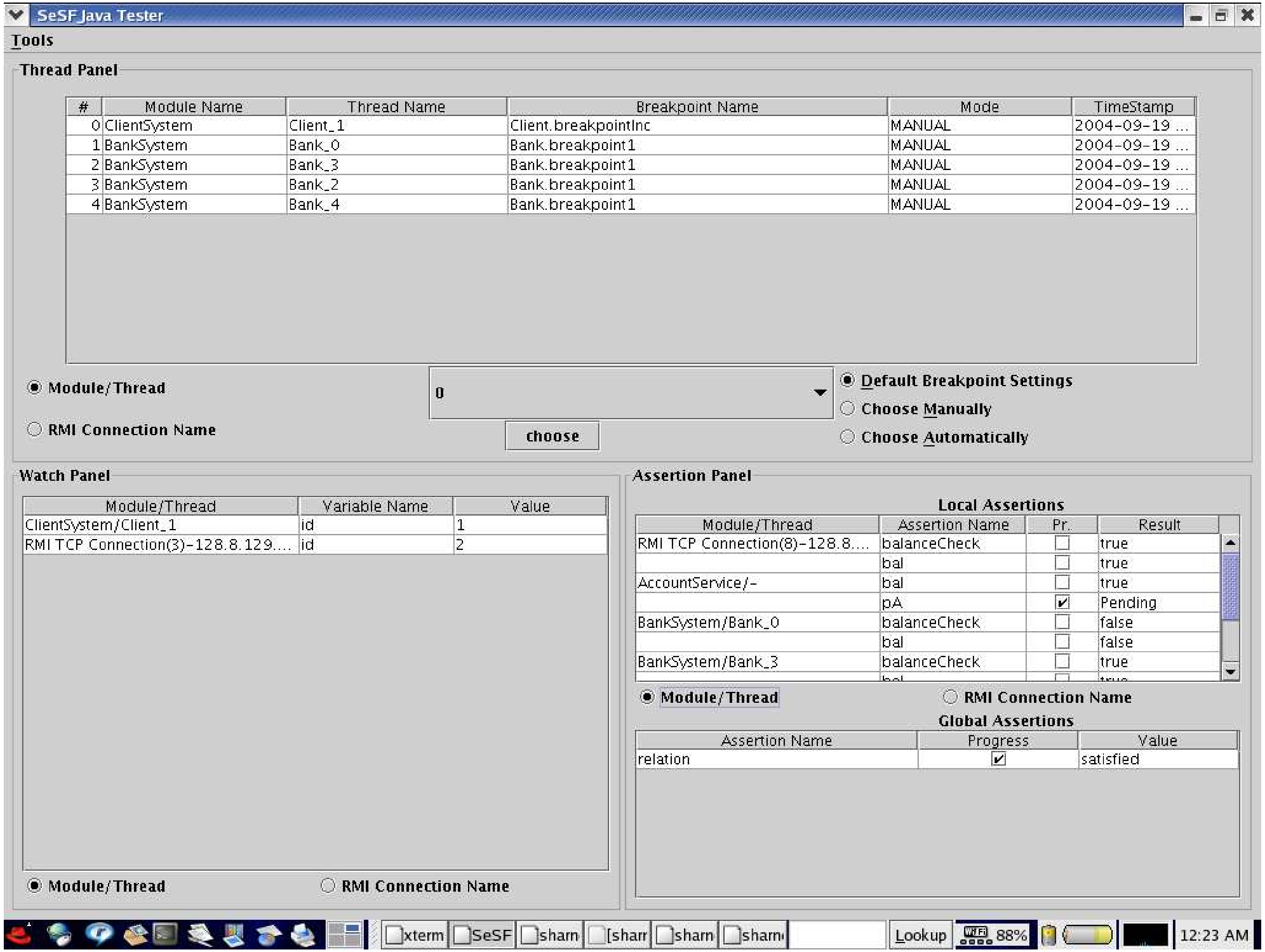
- **Module/Thread:** This displays the breakpoints by their name, mode (e.g., Manual or Automatic), the thread that encountered the breakpoint, and the module that this thread belongs to.

- **RMI Connection Name:** This displays the breakpoints by their name, mode and the name of the RMI connection port that connects the calling module to the Harness.

In Thread Panel (figure 5.3), a user clicks the “Choose Manual” radio button to manually choose the next thread to proceed. Clicking “Choose Automatic” tells the Harness to randomly pick threads to proceed.

Developers can insert tag “//# watch: $< \mathsf{var\_name} >$” inside SeSFJava program. This permits them to monitor these variables throughout execution in the **Watch Panel**.

Assertion Panel displays the assertions and the evolution of their values during the executions. There are two tables, one for local assertions (assertions that involve variables that belong to only one system or service), and another for global assertions (assertions that involve variables that span multiple systems or services). Column “Pr” is checked if the assertion is a progress assertion.



Thread Panel

#	Module Name	Thread Name	Breakpoint Name	Mode	TimeStamp
0	ClientSystem	Client_1	Client.breakpointInc	MANUAL	2004-09-19 ...
1	BankSystem	Bank_0	Bank.breakpoint1	MANUAL	2004-09-19 ...
2	BankSystem	Bank_3	Bank.breakpoint1	MANUAL	2004-09-19 ...
3	BankSystem	Bank_2	Bank.breakpoint1	MANUAL	2004-09-19 ...
4	BankSystem	Bank_4	Bank.breakpoint1	MANUAL	2004-09-19 ...

● Module/Thread
○ RMI Connection Name

0

● Default Breakpoint Settings
○ Choose Manually
○ Choose Automatically

Watch Panel

Module/Thread	Variable Name	Value
ClientSystem/Client_1	id	1
RMI TCP Connection(3)-128.8.129....	id	2

● Module/Thread ○ RMI Connection Name

Assertion Panel

Local Assertions

Module/Thread	Assertion Name	Pr.	Result
RMI TCP Connection(8)-128.8....	balanceCheck	□	true
	bal	□	true
AccountService/-	bal	□	true
	pA	☑	Pending
BankSystem/Bank_0	balanceCheck	□	false
	bal	□	false
BankSystem/Bank_3	balanceCheck	□	true

● Module/Thread ○ RMI Connection Name

Global Assertions

Assertion Name	Progress	Value
relation	☑	satisfied

Figure 5.3: Graphical Interface of the Harness

Chapter 6

SeSFJava Harness Overview

The previous section introduced the SeSFJava Harness by example. This section discusses it in more general terms. Figure 6.1 gives the overall structure and operation of the SeSFJava Harness:

- System and service program files are fed to SeSF Preprocessor. The preprocessor accepts a **configuration file** that contains all the parameters of preprocessing, for example, directories where the program resides, program file names, etc. Figure 6.2 illustrates a sample configuration file.

- The preprocessor generates the following: composite system program, assertion checker that checks assertions of all services and systems, and “Serializer and Checker” (SAC) module.

- The composite system is executed, execution is logged into a log file, system and service properties are checked, and violations are recorded.

- Users can interact with the composite system during its execution to influence the flow of execution and/or to view the results of evaluating assertions.

- The log analyzer is used to analyze the log file to extract event-traces of interest, e.g., those that have led to desired assertions failing. It provides a method to view log files in a readable format.

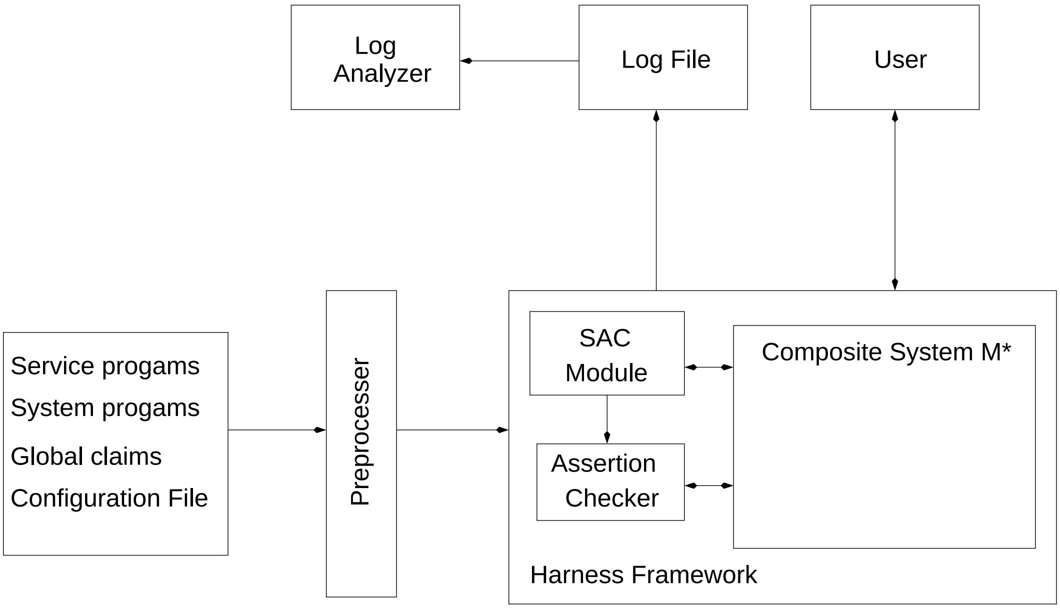


Figure 6.1: SeSFJava Harness: operation overview

Section 6.1 describes the types of systems supported by the Harness. Section 6.2 illustrates the types of assertions supported. Section 6.3 describes where the assertions are checked. Section 6.4 describes how the data necessary for assertion evaluation is collected. Section 6.5 describes how to evaluate assertions. Section 6.6 describes the operation of the breakpoints. Section 6.7 describes the configurations supported.

6.1 Process-based versus thread-based

The SeSFJava Harness can handle both **process-based** composite systems and **thread-based** composite systems. In the process-based case, the component systems of the composite system are all separate processes, perhaps in different machines. Consequently, the

```
OutputToSTDOut: true                                   // Direct comments about the progress to stdout
HarnessDistributed                                     // Process-based system
HarnessMachine: ”leibniz.cs.umd.edu”                   // Machine where the Harness resides
HarnessDirName: ”outApps/outStaticAccountRMI3”         // Directory where the Harness resides

inputDirName: ”../Apps/StaticAccountRMI3”              // Directory that contains input systems and services
outputDirName: ”../outApps/outStaticAccountRMI3”       // Output directory of preprocessed files
service: ”AccountService.java”                         // Input service file
system: ”ClientSystem.java”                            // Input system file
system: ”BankSystem.java”, theoremFiles: ”BankThm.thm” // Input system file
                                                       // Its assertions defined in BankThm.thm
global: ”SAC.java”                                     // File that contains global assertions
```

Figure 6.2: Configuration file of AccountExample (file account.cfg)

composite system being tested is a distributed system potentially spanning multiple machines. The services and the SAC module reside on one (arbitrary) machine. Calls from systems to services are executed using Java RMI (Remote Method Invocation) method calls. Because the SAC module has no access to the data variables inside processes, methods are instrumented into the systems to *marshal* to the SAC module the relevant data needed to calculate global assertions. Using Java eliminates the data encoding problem; for example, SeSFJava Harness does not care whether the underlying platform of a certain system is big endian or little endian. The example described in chapters 4 and 5 is process-based.

In the thread-based case, the component systems are all threads of a single process. Consequently, the composite system being tested resides in one machine. We put the SAC module also on that machine and give it access privilege to all data variables of services and public data variables of systems. For example, one may want to test Bank system against Account service, where Bank, Account and the Harness module are all threads within a single process. In this case, testing Bank system against Account service is the same as executing a composite system Bank^*. Figure 6.3 illustrates the outline of this framework.

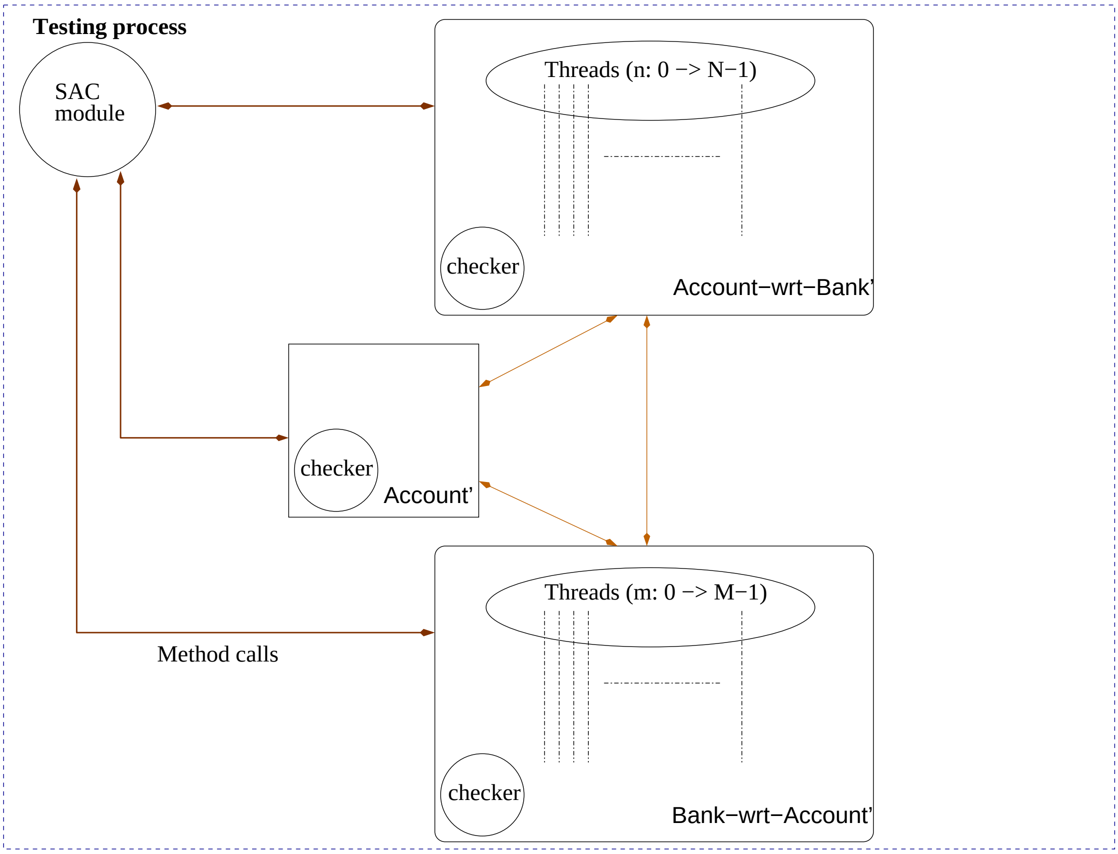


Figure 6.3: Testing framework for thread-based system

6.2 Types of assertions supported

The assertion checker evaluates assertions on the execution of the composite system generated thus far by SeSFJava Harness. The assertions can be progress assertions from the service specifications. They can also be safety and progress assertions specified by the developer, to provide insight and/or aid debugging; such assertions, referred to as **claims**, are not part of the system and service specifications but rather intended to be derivable from them.

SeSFJava supports the same assertions and predicates as SeSF, using a similar syntax. It supports the usual boolean operators: negation (!), equals (==), conjunction (&&), disjunction (||), and implication (⇒). It supports quantified assertions with integer-valued bound variables. The scope of the quantification is denoted by either a forall/endfor pair or a forsome/endfor pair. It supports all the safety and progress temporal operators, i.e., inv, unless, leadsto, wfair, and sfair. Because testing generates only finite executions, wfair and sfair are equivalent for testing purposes.

Fairness assertions require special handling. Consider wfair(X), where X is a thread. A finite execution σ does not satisfy wfair(X) if X is alive and is at a statement that is not blocked. The natural way to check whether this holds is to look into the JVM or operating system, but this is usually not feasible. Alternatively, one can capture this condition using appropriate system predicate. If X is not at a blockable statement, then it suffices to check whether the predicate X.isAlive() holds at the end of σ (where X.isAlive() is a system function that returns true whenever the thread's control pointer is in the thread's run method). If X is at a blocking statement, say m.wait(), where m is a lock object, then X.holdsLock(m) holds at the end of σ and thus wfair(X) succeeds. X.holdsLock(m) returns true if X currently holds the lock of m; note that m.wait() relinquishes the lock during waiting.

6.3 Assertion checking locations

Checking an assertion of the composite system involves three issues: when to check the assertion, how to collect data necessary to evaluate the assertion, and how to evaluate the assertion.

In SeSFJava Harness, assertion are checked whenever control reaches any of the following locations, referred to as **checking locations**:

- **xc** events in a system.
- **breakpoints** in a system.
- **dnw** and **upw** events in a service.

6.4 Collecting data for assertion checking

The assertion checker takes snapshots of the variables used in evaluating the assertions. This process is called **snapshot gathering**. It takes place at the checking locations. There are two kinds of snapshots: local snapshot and global snapshot. Gathering **local** snapshots requires instrumentation of certain method calls to evaluate local assertions. Gathering **global** snapshots requires an external running system (SAC module) that receives snapshots of variables from the systems under execution.

Let X denote the assertions that are to be checked. For any global state, let the X-image of the state denote the part of the state relevant to evaluating X, that is, the values of the variables of X. Note that the X-image may overlap with the states of several processes. For any execution, let the X-image of the execution denote the sequence of X-images of the states of the execution.

To check whether X holds, we need the X-image of the execution generated thus far. This can be collected at SAC if each process, when it reaches a checking location, sends its part of the X-image of its current state to SAC module. By integrating the most recent X-images from all the process, SAC module obtains the X-image of the current global state. By storing past X-images, SAC module obtains the sequence of X-images of the global states encountered at the breakpoints thus far.

SeSF Harness implements the above by inserting the following checkAssertions method in SAC module:

```
void checkAssertions() {
  For every global theorem <t>:
      Evaluate theorem <t>            // using X-image of the execution
      Write value of <t> to log file
}
```

Next, SeSFJava Harness inserts the following checkAssertions method in every system and service:

```
void checkAssertions() {
  For every local theorem <t>:
      Evaluate <t>;
      Write value of <t> to log file
  Inside a system:
      For every global variable <g> relevant to X-image:
         Marshall <g> to Harness module;
      Issue an RMI call to SAC.checkAssertions();
  Inside a service:
      Issue a call to SAC.checkAssertions();
}
```

Whenever the control reaches any checking location, a call is issued to local checkAssertions method which in turn calls SAC.checkAssertions.

6.5 Evaluation of assertions

Assertions can be checked by storing only a small amount of state per assertion, instead of the entire generated sequence. We describe this for each kind of assertion:

- $\mathsf{inv}(\mathsf{P})$

```
Initially: Result[0] = P;
At check i, for i > 0:
        Result[i] = Result[i-1] && P;
```

- $\mathsf{P\ unless\ Q}$

```
Initially: a[0] = P && !Q;
        Result[0] = true ;
At check i, where i > 0:
        a[i] = P && !Q;
        Result[i] = Result[i-1] && (a[i-1] implies (P || Q));
```

- $\mathsf{P\ leadsto\ Q}$

```
Initially: Result[0] =  !P || Q;
At check i, where i > 0:
        Result[i] = (Result[i-1] && !P) || Q;
```

- $\mathsf{wfair}(\mathsf{P}, \mathsf{name})$ or $\mathsf{sfair}(\mathsf{P}, \mathsf{name})$, where P is a predicate, and name is a character string:

```
Initially: Result[0] = !P;
At check i, where i > 0:
        Result[0] = !P;
        Write (<name>, <Result>) into log file.
```

- For an assertion with forall quantifier, the checker dynamically creates n assertions where n is the cardinality of the bound variable of the quantifier. The conjunction of all n assertions forms the result of the forall assertion. For example:

```
forall u: 1 -> N
  beginAssertion
     P(u) unless Q(u)
  endAssertion
```

  results in N assertions, and their conjunction forms one assertion.

- For an assertion with a forsome quantifier, the checker dynamically creates n assertions where n is the cardinality of the bound variable of the quantifier. The disjunction of all n assertions forms the result of the forsome assertion.

6.6 Breakpoints

As previously mentioned, SeSFJava Harness uses breakpoints to produce serialized behaviors. The Harness stops all threads that encounter breakpoints during their executions, and allows only one thread to continue. Consider five threads A, B, C, D and E (figure 6.4) running in the composite system. When the four threads A, B, C and D are dispatched, they stop at their first encountered breakpoints which are a1, b1, c1 and d1 respectively. Then, the harness chooses one thread from these threads (called thread pool) to proceed without interruption till the next breakpoint. For example, it chooses thread B to continue to breakpoint b2. When thread B reaches b2, it stops and the Harness chooses another thread to continue. Since thread E has no breakpoints, SeSFJava Harness cannot stop it during execution.

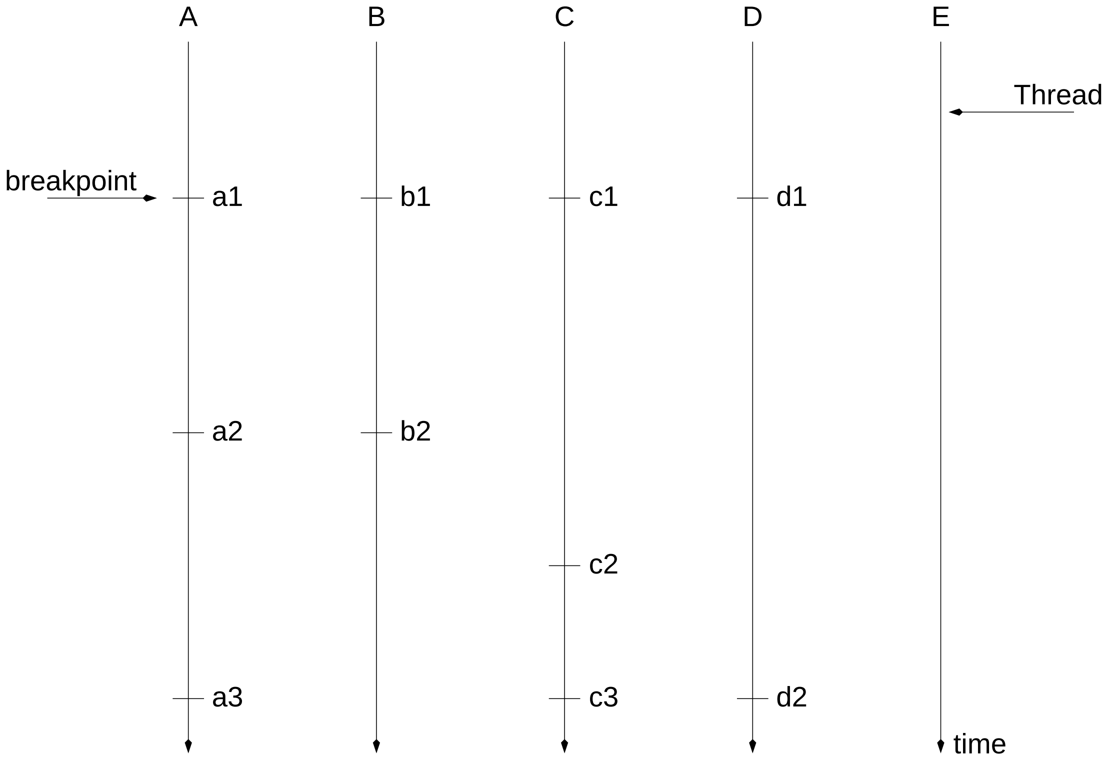


Figure 6.4: List of threads.

Choosing a thread to continue may be done automatically or manually depending on the mode of the breakpoint. The following modes are currently supported. Mode Manual means that the user selects which thread to continue. Mode Automatic tells the Harness to randomly select a thread to proceed. Mode Automatic_And_View instructs the Harness to print the status of the composite system (e.g., values of assertions) before choosing a thread automatically. Mode View prints the status of the composite system before continuing with the same thread. A thread must call a breakpoint with mode End before terminating. Finally, mode Wait means that the thread is going to execute a blocking statement. The different modes supported by SeSFJava Harness are illustrated in figure 6.5.

Setting breakpoints is the mechanism by which the tester achieves serial execu-

```
void breakpoint(name, mode) {
    Variable ThreadPool: all threads that have encountered breakpoints.
    If (mode = MANUAL)
        Add the calling thread to ThreadPool (if the thread is not already present).
        List the available assertions with their values.
        List the available threads inside ThreadPool.
        User chooses the next thread to continue.
        Notify the chosen thread to continue its work.
    If (mode = VIEW_AND_AUTOMATIC)
        Add the calling thread to ThreadPool (if the thread is not already present).
        List the available assertions with their values.
        List the available threads inside ThreadPool.
        Pick randomly the next thread to continue.
        Notify the chosen thread to continue its work.
    If (mode = AUTOMATIC)
        Add the calling thread to ThreadPool (if the thread is not already present).
        Pick randomly the next thread to continue.
        Notify the chosen thread to continue its work.
    If (mode = VIEW)
        List the available assertions with their values.
        List the available threads inside ThreadPool.
        Allow the calling thread to continue work
    If (mode = END)
        Remove the calling thread from ThreadPool.
        Allow the calling thread to continue its work.
        Pick randomly the next thread to continue.
        Notify the chosen thread to continue its work.
    If (mode = WAIT)
        Allow the calling thread to wait.
        Pick randomly the next thread to continue.
        Notify the chosen thread to continue its work.
}
```

Figure 6.5: method breakpoint of Tester.java

tions. Therefore, the user should insert breakpoints at appropriate locations. A misplaced breakpoint may lead to a violation of a valid assertion. For example,

```
Thread {
  ...
  x = 0;
  //# breakpoint("1", AUTOMATIC)
  ....
  y = 4;
  ...
  y = 3;
  //# breakpoint ("2", AUTOMATIC)
  ....
}
```

Assertion “(x==0) leadsto (y==4)” fails upon testing the previous program, although the behavior satisfies the assertion. The program has to be modified to:

```
Thread {
  ...
  x = 0;
  //# breakpoint(“1”, AUTOMATIC)
  ....
  y = 4;
  //# breakpoint (“2”, AUTOMATIC)
  ...
  y = 3;
  ....
}
```

A misplaced breakpoint may lead to a deadlock. For example,

```
Thread X {
  ...
  x = 0;
  //# breakpoint(“1”, AUTOMATIC)
  ....
  y = 4;
}
```

The thread ends while holding the lock from breakpoint “1”. Consequently, all other threads in the system will be blocked at their respective breakpoints waiting for thread X to relinquish control of the lock, which will not happen as the thread X is no longer active. The program has to be modified to:

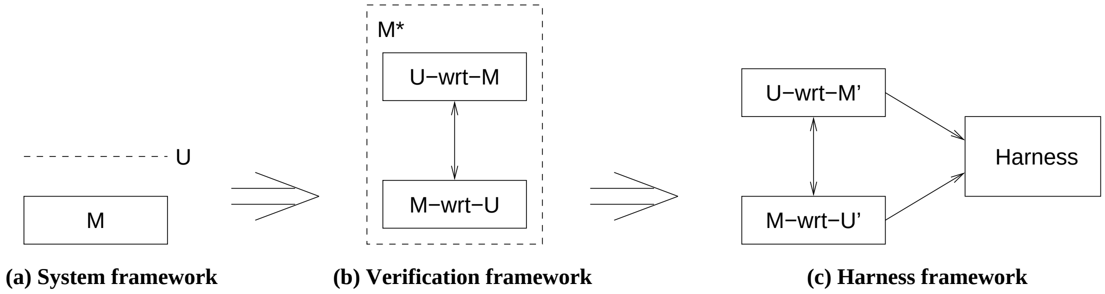
```
Thread {
  ...
  x = 0;
  //# breakpoint(“1”, AUTOMATIC)
  ....
  y = 4;
  //# breakpoint(“2”, END)
}
```

It is important to call breakpoint(<name>, END) before ending the thread (the call can be placed in a finally clause). Missing a call to breakpoint(<name>, END) results in the violation of the fairness assumptions (other threads are continuously enabled, but never executed).

6.7 Configurations

SeSFJava Harness can test various configurations of systems and services. In chapter 4, in order to verify that system M satisfies service U (figure 6.6(a)), we constructed composite system M^* of M-wrt-U and U-wrt-M as in figure 6.6(b) and proved that its execution results in no faulty transition. But in order to test M against U, we had to construct composite system $M^{*\prime}$ of U-wrt-M' and M-wrt-U' (Figure 6.6(c)). After construction of $M^{*\prime}$, it is executed to test the service satisfaction. We will present two more configurations.

Convention An *input* composite system is the system composed of systems and services that are stated in a *configuration* file.



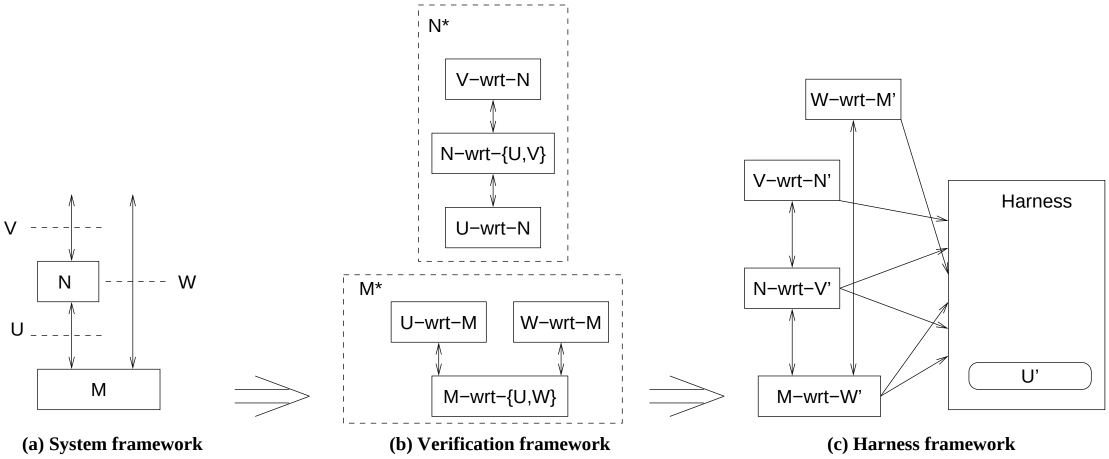
(a) System framework **(b) Verification framework** **(c) Harness framework**

Figure 6.6: Component system phases

6.7.1 Example 1

Consider a component system that is itself a composite system. Figure 6.7(a) illustrates systems M and N, and services U, V and W. In order to verify that M and N satisfies V and W, we construct composite systems M^* and N^* as in figure 6.7(b). M^* is composed of M-wrt-$\{\mathsf{U},\mathsf{W}\}$, U-wrt-M and W-wrt-M. N^* is composed of N-wrt-$\{\mathsf{U},\mathsf{V}\}$, U-wrt-N and V-wrt-N. If M^* and N^* are correct (each satisfy its services and assertions), then $\{\mathsf{M},\mathsf{N}\}$ satisfies $\{\mathsf{V},\mathsf{W}\}$.

In order to test $\{\mathsf{M},\mathsf{N}\}$ against $\{\mathsf{V},\mathsf{W}\}$, we have two options. The first is to test each component alone, that is, to test that M satisfies U and W, and to test that N uses U satisfies V. The second option is to construct composite system $\mathsf{MN}^{*\prime}$ of M-wrt-W′, U′, N-wrt-V′, V′, V-wrt-N′, W′ and W-wrt-M′ (figure 6.7(c)). After construction of $\mathsf{MN}^{*\prime}$, it is executed to test the service satisfaction.



(a) System framework (b) Verification framework (c) Harness framework

Figure 6.7: Component system phases of example 1

6.7.2 Example 2

Consider a closed composite system of systems M, N and O, and services U, V and W (figure 6.8(a)). For verification, we construct M^*, N^* and O^* (figure 6.8(b)). and verify each system independently. If each is correct (each satisfy its services and assertions), then the composite system MNO is correct. For testing of the composite system MNO, we construct the components shown in figure 6.8(c), and execute the entire composite system, and check the validity of the assertions.

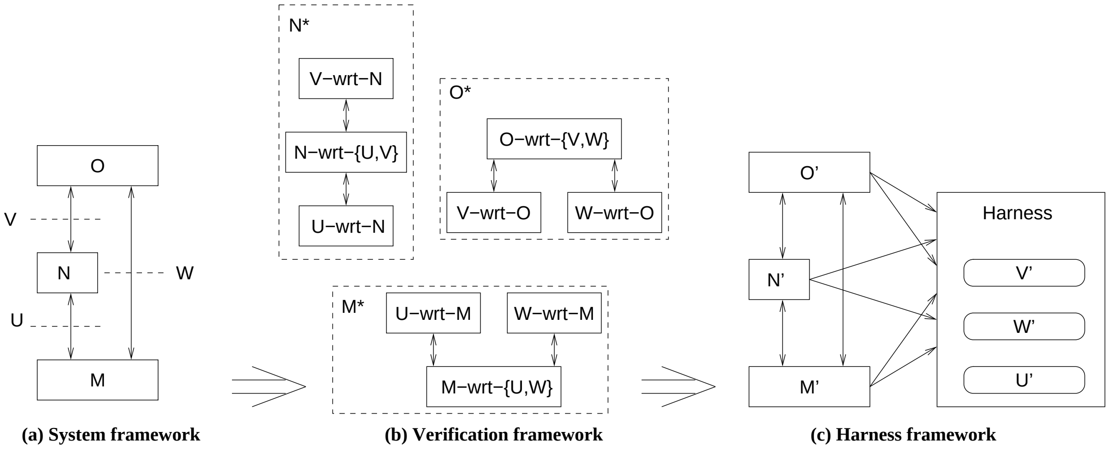


Figure 6.8: Composite system phases of example 2

The possible executions of M^* and N^* in figure 6.8(c) is typically a subset of the the possible executions of M^* and N^* in figure 6.7(c) because the system O' has a constraining effect (i.e., because it does not supply all the possible inputs that V and W can accept).

Part II

Applications

Chapter 7

Data Transfer Protocol

In this chapter, we apply SeSFJava to the data transfer part of a transport protocol, specifically, a sliding window protocol that provides reliable flow-controlled data transfer from a source to a sink over unreliable channels that can lose, reorder and duplicate messages in transit subject to a maximum message lifetime.

Fig. 7.1 illustrates the data transfer layers. SW_SourceUser passes data to SW_Source. SW_Source buffers the data (in a send window) and transfers it to SW_Sink, resending until it is acknowledged by SW_Sink. SW_Sink buffers data received out of sequence (in a receive window) and delivers data in sequence to SW_SinkUser. The sliding window protocol is significantly more complex than stop-and-wait or go-back-N protocols [42]. We assume fixed size messages for readability reasons.

SW_Source, SW_Sink, and the unreliable channels make up the SW_Sys composite system. SW_SourceUser and SW_SinkUser make up the composite system using the service. DT denotes the data transfer service, that is, the signature of the interactions between the systems on either side, as well as the permissible sequences of these interactions.

This chapter is organized as follows. Section 7.1 describes the SW_Source and

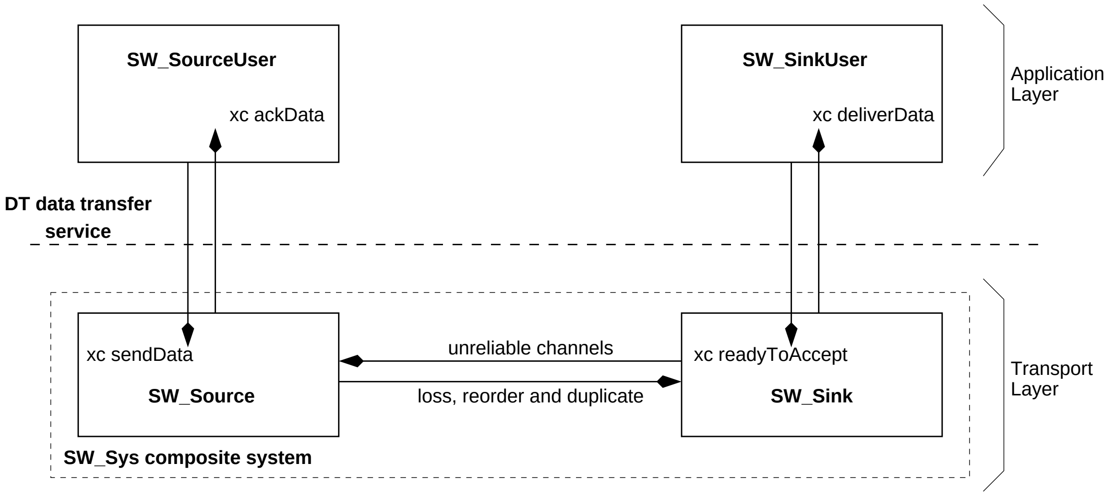


Figure 7.1: Data transfer service and protocol system

SW_Sink systems. Section 7.2 describes the DT service. Section 7.3 illustrates the program-version conditions for SW_Sys system to satisfy DT service. Section 7.4 demonstrates how to test SW_Sys system against DT service.

7.1 Systems

Figures 7.2 and 7.3 show the system programs in (high-level) SeSF for SW_Source and SW_Sink systems, respectively. SW_SourceUser creates SW_Source process and sets SW_Source.sourceuser to refer to itself (for callback methods). Similarly, SW_SinkUser creates SW_Sink process and sets SW_Sink.sinkuser to refer to itself (for callback methods). SW_SourceUser sends an array of bytes via xc-event SW_Source.sendData. SW_Source divides the received array into data blocks, and sends those data blocks to SW_Sink. When SW_Sink receives a data block, it replies with an ACK message. If SW_SinkUser has enough space (var SW_Sink.allowedBytes $>$ 0), SW_Sink delivers the data block to its user via xc-event SW_SinkUser.deliverData; otherwise SW_Sink waits (busy waiting) for

SW_SinkUser to call xc-event SW_Sink.readyToAccept before delivering more data to the user. Whenever SW_Source receives a new ACK (not a duplicate), it calls xc-event SW_SourceUser.ackData to inform the user that it has more empty space in the buffer.

Inside SW_Source system, thread DataSender sends data packets whenever data packets are ready in the buffer within the boundaries of the send window. Thread Retransmission retransmits un-acked packets whenever the timeout fires. Thread SourceReceiver receives ACK messages and modifies the variables accordingly.

Thread DataDelivery delivers received data to SW_SinkUser if the user has enough buffer space. Thread SinkReceiver receives data packets and store them in the sink's buffer.

Atomically-executed code is indicated by enclosing it in angled brackets (e.g., see DataSender thread in fig. 7.2; we use large-scale atomicity to keep the example small).

Both systems have the standard progress assumptions, that is, weak fairness of all threads.

Figures 7.4 and 7.5 outline the SeSFJava programs of the SW_Source and SW_Sink, respectively (for the complete SeSFJava code, see appendices C.1 and C.2). As usual, statements preceded by "//#" are SeSFJava constructs that are used only for testing.

7.2 Service

The service program DT in SeSF is given in figure 7.6. Dnw event DT.sendData corresponds to SW_SourceUser passing data to SW_Source (for brevity, we refer to the dnw event as DT.sendData rather than the more accurate DT.SW_Source.sendData). The

Description of sliding window protocol (source side):
At any time at the source, let $\mathsf{sendBuf}[0, 1, \ldots, (\mathsf{ng} - \mathsf{na} - 1)]$ denotes the sequence of data blocks generated by the source. Of these, $\mathsf{sendBuf}[0, 1, \ldots, (\mathsf{ns} - \mathsf{na} - 1)]$ have been sent but not yet acknowledged, and $\mathsf{na} \leq \mathsf{ns} \leq \mathsf{ng}$ holds. The variable sw is the source's estimate of the current *receive window* size of the sink, where $\mathsf{sw} \leq \mathtt{constant}\ \mathsf{SW}$. $[\mathsf{na}..(\mathsf{na} + \mathsf{sw} - 1)]$ constitutes the *send window*.

```
system-program SW_Source {               // system header
  constant int bufSize := 32 * 1024;     // buffer size is constant (equals SW ∗ message size)
  constant int msgSize := 128;           // message size is constant
  constant int SW := bufSize/msgSize,    // maximum send window size
  int ng := 0,                           // number of data blocks generated by local user, initially 0
     ns := 0,                            // number of data blocks sent at least once, initially 0
     na := 0,                            // number of data blocks acknowledged, initially 0
     bufUsed := 0;                       // occupied portion of buffer in bytes, initially 0
     sw := SW,                           // send window size, initially SW.
  Buffer sendBuf;                        // send buffer of SW equal-sized data blocks;
                                         // no need to store acked data blocks [0,1,. . .,(na-1)]
  Timer rTimer;                          // retransmission timer, fires after timeout elapses
  boolean rTimerFired                    // it is true whenever rTimer fires
  SW_SourceUser sourceuser;              // reference to the user application for callback methods

 // data.length is the number of bytes in array data
 xc-event void sendData(byte[] data)  {                          //xc-event header
    ec:   bufUsed + data.length ≤ bufSize ∧ data.length = 0 ∧ data.length % msgSize = 0;
    ac:   Divide data array into data blocks;
          tmp := number of constructed data blocks;
          Store tmp data blocks in sendBuf;                       // sendBuf[ng..(ng+tmp-1)] := data[...]
          ng := ng + tmp;
          bufUsed := bufUsed + data.length;
 }

  Thread DataSender (){
   // Busy waiting is used to keep the example simple
     while ⟨                                                  // '⟨': begin atomic section
          (1 ≤ ns − na  < min (ng, na + sw) − na) {
            Send data block with sequence number (ns);        // via unreliable channel
            Reset rTimer of data block ns;
            ns := ns + 1;
     } ⟩                                                     // '⟩': end atomic section
  }

  lc-event Retransmission (int seqNo) {
     ec: na ≤ seqNo < ns ∧ rTimerFired;
     ac: Send data block (seqNo);              // via unreliable channel
          Reset rTimer of data block seqNo;
  }

  Thread SourceReceiver {
     while(true) {
          Receive ACK(seqNo, w);        // blocks till an ACK message is received with sequence number seqNo
                                        // and window size w
          ⟨                             // begin atomic section
           int tmp := seqNo − na;       // number of newly acked messages
          if (1 ≤ tmp ≤ (ns − na)) {
             sourceuser.ackData(tmp ∗ message size);
             na := na + tmp;              // remove first tmp data blocks from sendBuf;
             sw := w;
             bufUsed := bufUsed − tmp ∗ data block size;
           } else if (tmp = 0)
             sw := max(sw, w);
          ⟩                                 // end atomic section
     }
  }

 progress-assumption default {
     wfair(DataSender, Retransmission, SourceReceiver);
 }
}
```

Figure 7.2: SW_Source system program in SeSF

Description of sliding window protocol (sink side):
At any time at the sink, $\mathsf{recvBuf}[0]$ has not yet been delivered to the user. $\mathsf{recvBuf}[0, 1, \ldots, (\mathsf{RW} - 1)]$ may have been received out-of-sequence, in which case, they are temporarily buffered, but are not passed to the user. $[\mathsf{nr}..\mathsf{nr} + \mathsf{RW} - 1]$ (which is $\mathsf{recvBuf}[0..\mathsf{RW} - 1]$) constitutes the receive window.

```
system-program SW_Sink {        // system header
  int allowedBytes := 0,        // number of the bytes that SW_Sink is able to foist on user's buffer, initially 0.
     nr := 0;                   // number of data blocks delivered to the local user, initially 0.
  Buffer recvBuf;               // buffer of RW equally-sized data blocks
  SW_SinkUser sinkuser;         // reference to the user application for callback methods

  xc-event void readyToAccept(int n)  {    // xc-event header
      ec: true;                            // not checked by system, no side effects
      ac: allowedBytes := n;               // no event calls, no blocking
 }

 Thread DataDelivery () {
    // Busy waiting is used to keep the example simple
      while ⟨ (recvBuf[nr] $\neq$ null $\wedge$ allowedBytes $>$ 0) {       // '⟨': begin atomic section
            allowedBytes := allowedBytes $-$ recvBuf[nr].length;
            sinkuser.deliverData(recvBuf[nr]);
            remove recvBuf[nr];                                         // no need to store recvBuf[nr]
            nr := nr + 1;
      }   ⟩                                                             // '⟩': end atomic section
 }

 Thread SinkReceiver {
     while (true) {
        Receive data block (cj, data);     // blocks until a data block with sequence number (cj) and contents (data)
          ⟨ if (0 $\leq$ cj $-$ nr $<$ RW)          // begin atomic section
               recvBuf[cj $-$ nr] := data;
           Send ACK message ACK(nr, RW);
          ⟩                                 // end atomic section
     }
 }

 progress-assumption default {
     wfair(ModifyWindow, DataDelivery, SinkReceiver);
 }
}
```

Figure 7.3: SW_Sink system program in SeSF

```
//# system_program;
class  SW_Source{
  //# HarnessInterface harness = ...;
  SW_SourceUser sourceuser; // ref. for callback methods
  Socket nSocket;
  Vector sendBuf = new Vector ();
  final static int msgSize = 128;
  final static int bufSize = 32*1024;
  final static int SW = bufSize / msgSize;
  int bufUsed, ns, na, ng, sw = SW;
  Object lock = new Object();  // lock object
  . . .

  //# xc_event;
  public void sendData(byte[] data)  {
    //# ec: data.length !=0 &&
    //#     bufUsed + data.length <= bufSize &&
    //#     data.length % msgSize == 0;
    //# breakpoint(...);
    synchronized(lock){
      . . .
      bufUsed += data.length;
    }
  }

  // Thread is a class that continuously
  // executes method run
  class DataSender extends Thread {
    . . .
    public void run() {
      while (true){
        //# breakpoint(...);
        synchronized(lock){
          . . .
          sendDataBlock(ns);  // Send data block ns
        }
        . . .
      }
    }
  }
```

```
  // TimerTask is class that executes method run
  // whenever its timer fires
  class Retransmission extends TimerTask {
    . . .
    public void run() {
      //# breakpoint(...);
      sendDataBlock(j);  // retransmit block j
                  // when timer fires and it is not acked
      //# breakpoint(...);
    }
  }

  class SourceReceiver extends Thread {
    . . . ;
    public void run(){
      while (true){
        //# breakpoint(...);
        // get ACK message with (seqNo, w)
        . . .
        synchronized(lock){
          int tmp = seqNo -na;
          if (tmp >= 1 &&
              tmp <= ns - na){
              . . .
              sourceuser.ackData(ackedBytes);
          } else if (tmp == 0)
              sw = sw > w ? sw : w;
        }
        //# breakpoint(...);
        . . . ;
      }
    }
  }

  //# progress_assumption default {
  //#   beginAssertion {
  //#     wfair(!DataSender.isAlive()) &&
  //#     wfair(!SourceReceiver.isAlive()))
  //#   }
  //# }
}
```

Figure 7.4: Outline of SeSFJava SW_Source system program (file SW_Source.java) (see appendix C.1 for complete program)

```
//# system_program;
class  SW_Sink {
  //# HarnessInterface harness = ...;
  . . .
  SW_SinkUser sinkuser;
  Socket nSocket;
  Vector recvBuf = new Vector();
  final static int bufSize = 32 * 1024;
  final int msgSize = 128;
  final int RW = bufSize / msgSize;
  int nr, allowedBytes = bufSize;
  Object lock = new Object();
  . . .

  //# xc_event;
  public void readyToAccept(long n)  {
    //# ec: true;
    allowedBytes = n;
  }

  class SinkReceiver extends Thread {
    . . .
    public void run() {
      while (true) {
        // receive data block with (seqNo, data)
        . . .
        //# breakpoint(...);
        synchronized(lock){
          int tmp = seqNo − nr − 1;
          if ((seqNo − nr − 1 >= 0)
              && tmp < RW && data.length ! = 0 &&
              recvBuf.elementAt(tmp) == null) {
             recvBuf.set(tmp, data); // recvBuf[tmp] = data
             // send ACK
             . . .
          }
          . . .
        }
      }
    }
  }

  class DataDelivery extends Thread {
    . . .
    public void run() {
      while (true) {
        //# breakpoint(...);
        synchronized(lock){
          if (recvBuf.elementAt(0) ! = null &&
             allowedBytes > 0) {
             . . .
             dtsink.deliverData(delData); // delData denotes
                                          // deleted data
          }
        }
      }
    }
  }

  //# progress_assumption default {
  //#   beginAssertion {
  //#     wfair(!DataDelivery.isAlive()) &&
  //#     wfair(!SW_SinkReceiver.isAlive())
  //#   }
  //# }
}
```

Figure 7.5: Outline of SeSFJava SW_Sink system program (file SW_Sink.java) (see appendix C.2 for complete program)

event appends the data to a stream (infinite array), and is enabled if the data fits the available space (as advertised by prior calls of upw event SW_SourceUser.ackData). Upw event DT.deliverData corresponds to SW_Sink passing data to SW_SinkUser. It is enabled if the data to be delivered is in sequence (with respect to the data sequence passed down by SW_SourceUser), and the SW_SinkUser buffer has enough space. SW_SinkUser can advertise its window at any time (via dnw event DT.readyToAccept). Upw DT.ackData informs the source user how much data has been delivered to the sink user.

Service DT has two progress obligations: allDataAcked which requires that all sent data are eventually acked, and dataDelivered which requires that all sent data are eventually delivered to the sink user.

Figure 7.7 outlines the SeSFJava service program of the DT. Notice that there is a a difference between assertion allDataAcked in SeSF (fig. 7.6) and the assertion allDataAcked in SeSFJava (fig. 7.7). We cannot apply SeSF.allDataAcked to SeSFJava systems, because we have to check for every integer value of n, which is infeasible. So, we have to use an assertion that models the same constraint. Because the execution is finite, SeSFJava.allDataAcked can be used instead of SeSF.allDataAcked.

7.3 DT satisfaction conditions

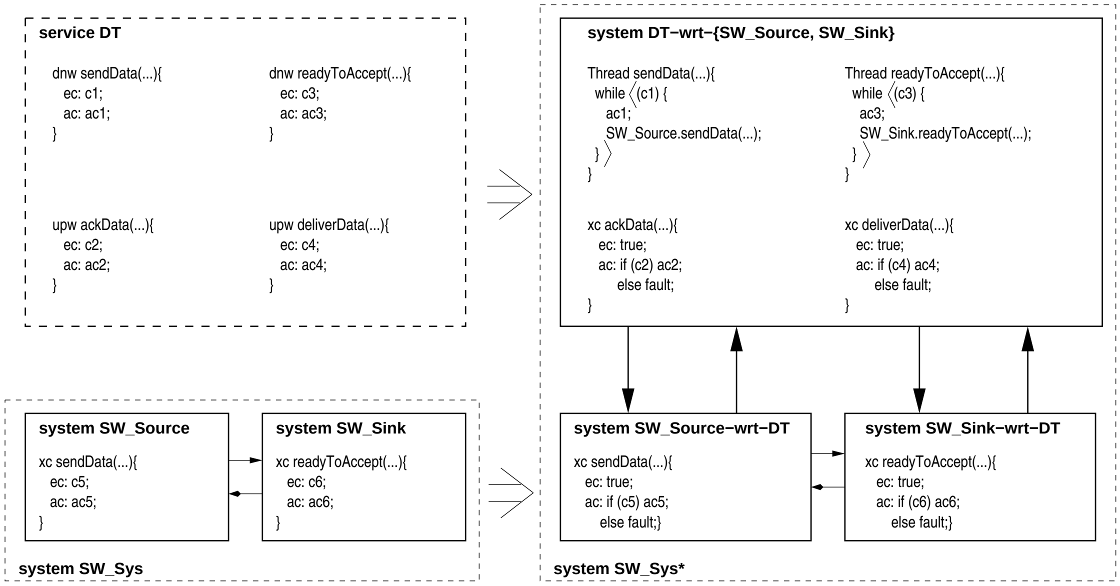
Fig. 7.8 illustrates the construction of SW_Sys* from SW_Sys and DT. SW_Sys* consists of SW_Source-wrt-DT, SW_Sink-wrt-DT, the channels between them, and DT-wrt-{SW_Source, SW_Sink}. In particular, every output call in SW_Source and SW_Sink is replaced by a call to the corresponding event of DT by appropriately modifying variables

```
service-program DT {            // service program's header
   // Declarations
   int msgSize;                 // message size (constant)
   // Source side variables.
   Stream srcHist;               // source entity history in bytes
   int srcBufSize,               // equals SW ∗ message size
      srcBufUsed;                // occupied portion of source buffer in bytes, always srcBufUsed ≤ srcBufSize
   int srcNumSent,               // number of bytes accepted from source's local user, initially 0
      srcNumAcked;               // number of acked bytes (at source entity), initially 0

   // Sink side variables.
   int sinkNumDelivered,        // number of bytes delivered to sink user, initially 0
      sinkBufAvail;             // number of bytes that sink user can accept, initially (RW ∗ message size)
  // Events of source side:

  // sends data from local user to source entity to be delivered to remote user
  dnw-event void SW_Source.sendData(byte []data) {        // dnw event header
       ec:    srcBufUsed + data.length ≤ srcBufSize ∧ data.length > 0 ∧ data.length % msgSize = 0;
       ac:    // data.length is number of bytes in data array
              srcHist[srcNumSent .. srcNumSent + data.length - 1] := data[0..data.length-1];
              srcNumSent := srcNumSent + data.length;
              srcBufUsed := srcBufUsed + data.length;
  }

  // notifies the entity user that n bytes have been acked by remote user
  upw_event void SW_SourceUser.ackData(int n) {        // upw event header
       ec:    srcNumAcked + n  ≤ srcNumSent;
       ac:    srcBufUsed  := srcBufUsed − n;
              srcNumAcked := srcNumAcked + n;
  }

  // Events of sink side

  // informs sink entity that its user can accept cumulative amount of data (in bytes) equals to n
  dnw_event void SW_Sink.readyToAccept(long n) {
       ec:    true;
       ac:    sinkBufAvail := n;
  }

  // delivers data to local user, such that, data is delivered in sequence without loss or duplication
  upw_event void SW_SinkUser.deliverData(byte []data) {
       ec:    sinkNumDelivered + data.length ≤ srcNumSent ∧
              data.length ≤ sinkBufAvail ∧ data.length > 0 ∧
              srcHist[sinkNumDelivered .. sinkNumDelivered + data.length] = data[0..data.length];
       ac:    sinkNumDelivered := sinkNumDelivered + data.length;
              sinkBufAvail     := sinkBufAvail − data.length;
  }

  progress-obligation allDataAcked {
         ((srcNumAcked = n) ∧ (sinkNumDelivered > n) leadsto  (srcNumAcked = n))
  }

  progress-obligation dataDelivered {
         ((sinkNumDelivered = n) ∧ (srcNumSent > n) ∧ (sinkBufAvail > 0)) leadsto (sinkNumDelivered > n)
  }
}
```

Figure 7.6: SeSF DT: data transfer service program

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

//# service_program;
class DT  extends UnicastRemoteObject implements ... {
   final static int msgSize = 128;

   // Source side variables.
   ByteArrayOutputStream srcHist = new ByteArrayOutputStream ();
   int srcBufSize     = 32 *1024,
      srcBufUsed;
   long srcNumSent, srcNumAcked;    // = 0

   // Sink side variables.
   long sinkNumDelivered, // = 0
      sinkBufAvail = 32 * 1024 ;

   DT() throws RemoteException {
      try {
         Naming.rebind("DT", this);
      } catch (Exception e) { throw new RemoteException(); }
   }

   // Events of source side
   //# dnw_event: SW_Source;
   public synchronized void sendData(byte []data)  throws RemoteException {
      //# ec: srcBufUsed + data.length <= srcBufSize && data.length > 0 && data.length % msgSize == 0;
      srcHist.write(data, 0, data.length);
      srcNumSent + = data.length;
      srcBufUsed + = data.length;
   }

   //# upw_event: SW_SourceUser;
   public synchronized void ackData(int n)  throws RemoteException  {
      //# ec: srcNumAcked + n  <= srcNumSent;
      srcBufUsed  = srcBufUsed  − n;
      srcNumAcked = srcNumAcked + n;
   }

   // Events of sink side
   ⋮
   //# progress_obligation allDataAcked {
   //#   beginAssertion {
   //#     (srcNumAcked < sinkNumDelivered)  leadsto
   //#          (srcNumAcked == sinkNumDelivered)
   //#   }
   //# }
   ⋮
}
```

Figure 7.7: Outline of data transfer service program (file DT.java) (see appendix C.3 for complete program)



```
dnw sendData(...){
  ec: c1;
  ac: ac1;
}

dnw readyToAccept(...){
  ec: c3;
  ac: ac3;
}

upw ackData(...){
  ec: c2;
  ac: ac2;
}

upw deliverData(...){
  ec: c4;
  ac: ac4;
}
```

```
Thread sendData(...){
 while 〈(c1) {
   ac1;
   SW_Source.sendData(...);
 } 〉
}

Thread readyToAccept(...){
 while 〈(c3) {
   ac3;
   SW_Sink.readyToAccept(...);
 } 〉
}

xc ackData(...){
  ec: true;
  ac: if (c2) ac2;
      else fault;
}

xc deliverData(...){
  ec: true;
  ac: if (c4) ac4;
      else fault;
}
```

```
xc sendData(...){
  ec: c5;
  ac: ac5;
}

xc readyToAccept(...){
  ec: c6;
  ac: ac6;
}
```

```
xc sendData(...){
  ec: true;
  ac: if (c5) ac5;
     else fault;}

xc readyToAccept(...){
  ec: true;
  ac: if (c6) ac6;
     else fault;}
```

Figure 7.8: Service satisfaction transformations

sourceuser and sinkuser.

The safety condition for SW_Sys offers DT reduces to the following:

1. SW_Sys* does not have undefined values or operations (division by zero, signature-inconsistent call, type mismatch, etc.).

2. SW_Sys* does not call a disabled event, which reduces to the following predicates being invariant:

   - DT.sendData.ec ⇒ SW_Source.sendData.ec

     (This formalizes the constraint that SW_Source.sendData should be enabled whenever its user calls DT.sendData. The predicates below are similarly obtained.)

   - DT.readyToAccept.ec ⇒ SW_Sink.readyToAccept.ec

   - SW_Source at sourceuser.ackData$(\cdots)$ ⇒ DT.ackData.ec

- SW_Sink at sinkuser.deliverData$(\cdots)$ $\Rightarrow$ DT.deliverData.ec

The progress condition holds iff SW_Sys* satisfies progress obligations allDataAcked and dataDelivered assuming weak fairness of SW_Sys's threads.

Although we do not do so here, it would be straightforward to prove by assertional reasoning that these conditions hold (e.g., as in [76]).

7.4 Testing and assertion checking harness

To test SW_Sys against DT, we do the following:

1. Create a Harness process to control the test execution. The Harness process is bound to RMI (Remote Method Invocation in Java) port "DTHarness".

2. Construct from SW_Sys* a composite system SW_Sys$^{*\prime}$ which interacts with the harness. SW_Sys$^{*\prime}$ consists of SW_Source-wrt-DT$'$ (a version of SW_Source-wrt-DT that interacts with the harness), SW_Sink-wrt-DT$'$ (a version of SW_Sink-wrt-DT that interacts with the harness), and DT-wrt-$\{$SW_Source, SW_Sink$\}'$ (a version of DT-wrt-$\{$SW_Source, SW_Sink$\}$ that interacts with the harness).

3. Execute SW_Sys$^{*\prime}$ along with (and under the control of) the harness process.

4. Check whether the generated execution becomes faulty.

Section 7.4.1 describes how to obtain SW_Sys$^{*\prime}$. Section 7.4.2 describes how to execute SW_Sys$^{*\prime}$ under the control of the harness process.

7.4.1 Constructing SW_Sys$^{*\prime}$

The first step is to construct composite system SW_Sys$^{*\prime}$ (figure 7.9). Section 7.3 described how to get SW_Source-wrt-DT, SW_Sink-wrt-DT and DT-wrt-{SW_Source, SW_Sink}. In addition to those modification, we need these components to connect to the harness. This leads to the following modifications.

First, we construct SW_Source-wrt-DT′, referred to as SW_Source′, from SW_Source-wrt-DT as follows:

- Tag //#HarnessInterface harness = . . . ; indicates the location of the harness, i.e., its RMI port.

- For every xc event, (1) insert a call to method checkAssertions which sends data necessary for assertion checking to SAC module, and (2) log information to the log file.

- Insert breakpoints at locations specified by tag //#breakpoint.

Second, we construct SW_Sink-wrt-DT′, referred to as SW_Sink′, from SW_Sink-wrt-DT.

Third, we construct DT-wrt-{SW_Source, SW_Sink}′, referred to as DT′, from DT-wrt-{SW_Source, SW_Sink}′ as follows. For every upw/dnw event, insert a call to method checkAssertions, and log information to log file.

Fourth, we construct SW_Sys$^{*\prime}$ consisting of SW_Source′, SW_Sink, the channels between them and DT′.

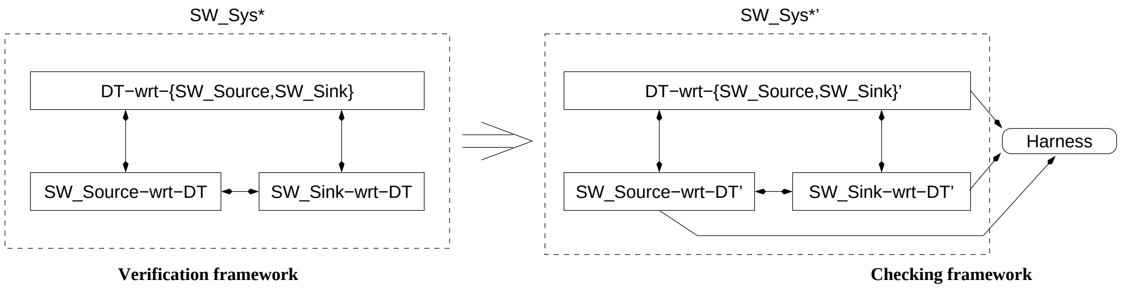


Figure 7.9: $\mathsf{SW\_Sys}^*$ and $\mathsf{SW\_Sys}^{*\prime}$ composite systems

7.4.2 Executing $\mathsf{SW\_Sys}^{*\prime}$

Once $\mathsf{SW\_Sys}^{*\prime}$ is constructed, the next step is to obtain the testing platform on which it can be executed. SAC (Serializer And Checker) module, within the harness, ensures that $\mathsf{SW\_Sys}'$-DT' interactions are executed atomically, and that only one thread is proceeding at a time. SeSFJava harness inserts breakpoints in $\mathsf{SW\_Sys}'$ and DT' such that at any time, at most one thread of $\mathsf{SW\_Sys}^{*\prime}$ runs and every other thread is paused at a breakpoint. SAC module maintains relevant state for every process, such as whether the process is running, paused, blocked, or about to be terminated. Each thread sends its state to the SAC module. Breakpoints are inserted manually to indicate where the thread transitions take place.

Assertions are evaluated at checking locations, specifically, at the start of every event and at every breakpoint as mentioned in section 6.3. For example, the scheme to test if $\mathsf{SW\_Source}$ satisfies assertion $\mathsf{inv(SW\_Source.sw} >= 0)$ is as follows. First, whenever $\mathsf{SW\_Sys}'$ encounters a checking location, it sends $\mathsf{SW\_Source.sw}$ to the Harness (via method $\mathsf{checkAssertions}$). Second, whenever the harness receives this field, it checks whether the predicate $\mathsf{SW\_Source.sw} >= 0$ holds. If the predicate fails once, then the invariant does not hold.

After $\mathsf{SW\_Sys}^{*\prime}$ is constructed, it is executed on the same platform as $\mathsf{SW\_Sys}^{*}$ as follows:

1. SeSFJava harness starts as a separate process, and binds itself to RMI port "DTHarness".

2. DT' process starts, and looks up for the harness' port "DTHarness".

3. $\mathsf{SW\_Sys}'$ process is created. It looks up for port "DTHarness" using RMI lookup command. So, both system (source and sink) are hooked up with the harness.

4. The developer can use the harness either in batch mode, letting the harness run for a while and then analyzing the log file, or in interactive mode, influencing the flow of the execution manually.

Chapter 8

Connection Management Protocol

In this chapter, we apply SeSFJava to the connection management part of a transport protocol. Here, a client connects and terminates connections to a server using messages sent over unreliable channels that can lose, reorder and duplicate messages in transit subject to a maximum message lifetime. The protocol has been taken from [59, 77].

Figure 8.1 illustrates the components of connection management protocol. When CM_ClientUser wants to establish a connection to CM_ServerUser, it passes its request to CM_Client. CM_Client, in turn, establishes a connection with CM_Server via a three-way handshake. If the handshake is successful, CM_Server and CM_Client notify CM_ServerUser and CM_ClientUser, respectively, of the connection establishment; otherwise they notify the users of the cancellation. After establishing the connection, the client and the server may exchange data using a data transfer protocol (e.g., the one described in chapter 7). When CM_ClientUser wants to terminate an open connection with CM_ServerUser, it passes its request to CM_Client, which, in turn, terminates the connection with CM_Server via a two-way handshake. Both CM_Client and CM_Server notify their respective user applications of the termination. CM_Client, CM_Server, and the unreliable channels make

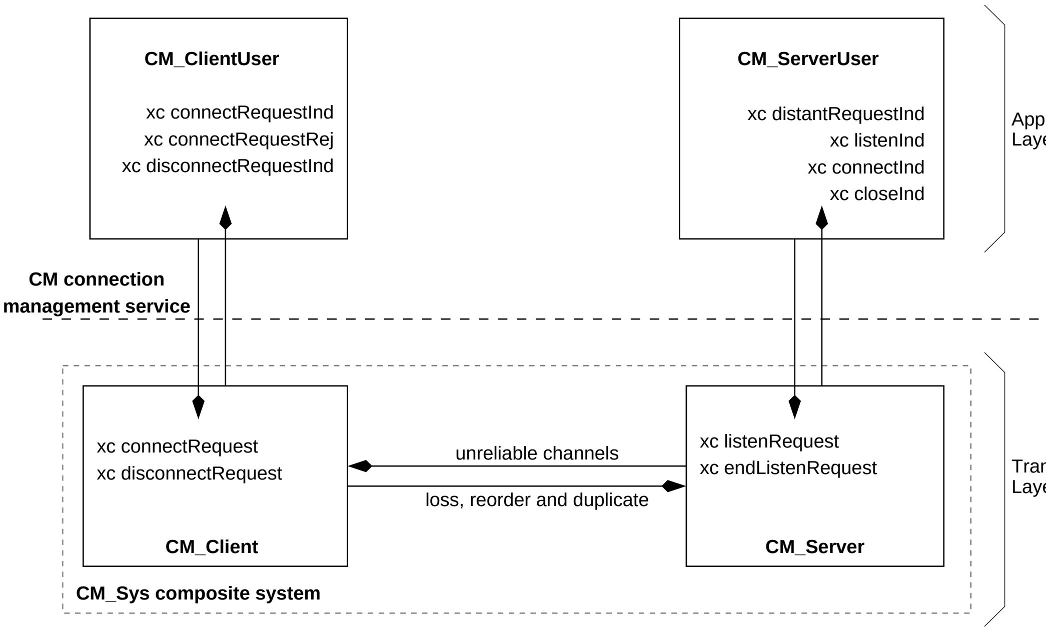


Figure 8.1: Connection management service and protocol system

up the CM_Sys composite system. CM_ClientUser and CM_ServerUser make up the composite system of the application. CM denotes the connection management service, that is, the signature of the interactions between the systems on either side, as well as the permissible sequences of these interactions.

This chapter is organized as follows. Section 8.1 describes the CM_Client and CM_Server systems. Section 8.2 describes the CM service. Section 8.3 illustrates the conditions necessary for system CM_Sys to satisfy CM service. Section 8.4 demonstrates how to test CM_Sys system against CM service.

8.1 Systems

Figures 8.2 and 8.3 illustrate the system programs in SeSF for CM_Client and CM_Server systems, respectively. Before explaining these programs, we first define the notion of incarnations. Each connection between the client and the server is an association between two incarnations: one at the client and another at the server. A new incarnation at the CM_Client is created whenever its user requests a new connection establishment. A new incarnation at the CM_Server is created whenever it becomes willing to accept a remote connection request. Every incarnation is assigned an incarnation number when it starts; the incarnation is uniquely distinguished by its incarnation number and user id. Each of the client and the server has at most one incarnation at any time.

CM_Client and CM_Server exchange messages of the form $\mathsf{M}(\mathsf{sin}, \mathsf{rin})$, where M is the type of the message, sin is the sender's incarnation number, and rin is the intended receiver's incarnation number. In some messages, sin or rin may be absent, denoted by "-". The CM_Client, sends the following messages: $\mathsf{CR}(\mathsf{sin}, -)$, which indicates connection request; $\mathsf{CRRACK}(\mathsf{sin}, \mathsf{rin})$, which indicates connection request reply ack; $\mathsf{DR}(\mathsf{sin}, \mathsf{rin})$, which indicates disconnect request; and $\mathsf{REJ}(-, \mathsf{rin})$, which indicates reject. The CM_Server sends the following messages: $\mathsf{CRRACK}(\mathsf{sin}, \mathsf{rin})$, which indicates connection request reply; $\mathsf{DRACK}(\mathsf{sin}, \mathsf{rin})$, which indicates disconnect request ack; and $\mathsf{REJ}(-, \mathsf{rin})$, which indicates reject.

Each message is either a primary or a secondary message. A **primary** message is sent repeatedly until a response is received. Client's CR and DR, and Server's CRR are primary messages. A **secondary** message is sent only in response to the reception of a primary message, and does not wait for response. Client's CRRACK and REJ, and Server's DRACK and REJ are secondary messages.

CM_ClientUser creates CM_Client process and sets CM_Client.clientuser to refer to itself (for callback methods). Similarly, CM_ServerUser creates CM_Server process and

Description of connection management protocol (client side):
Var status: {CLOSED, OPENING, OPEN, CLOSING}; initially CLOSED. Status of client's relationship with the server. CLOSED iff client has no incarnation involved with the server. OPENING means client has an incarnation requesting a connection with the server. OPEN means client has an incarnation open to the server. CLOSING means client has an incarnation closing a connection with the server.
Var lin: {nil, 0, 1, ...}; initially nil. Local incarnation number. nil if status = CLOSED. Otherwise identifies client incarnation involved with the server.
Var din: {nil, 0, 1, ...}; initially nil. Distant incarnation number. nil if status equals CLOSED or OPENING. Otherwise identifies the incarnation of the server with which the client incarnation is involved.

```
system-program CM_Client {
  //# static HarnessInterface harness := (HarnessInterface) Naming.lookup("CMHarness");
    int lin := nil;                 // local incarnation number
    int din := nil;                 // distant incarnation number
    int linGen := 0;                // incarnation number generator
    Timer rTimer;                   // retransmission timer, fires after timeout elapses
    boolean rTimerFired             // this boolean is true whenever rTimer fires
    CM_ClientUser clientuser;       // Reference to the user application for callback methods

    xc-event void connectRequest () {             xc-event void disconnectRequest () {
       ec: status = CLOSED;                          ec: status = OPEN
       ac: status := OPENING;                        ac: status := CLOSING;
            lin := linGen++;                              Send msg DR(lin, din);
            Send msg CR(lin, -);                          Reset rTimer of msg DR;
            Reset rTimer of msg CR;                  }
    }

    lc-event CR_Retransmission () {                      lc-event DR_Retransmission () {
       ec: (status = OPENING ∧ rTimerFired);                 ec: status = CLOSING ∧ rTimerFired;
       ac: Send msg CR(lin,-);                               ac: Send msg DR(lin,din);
            Reset rTimer of msg CR;                               Reset rTimer of msg DR;
    }                                                      }

  Thread CM_Receiver {
    Receive msg {
       //# breakpoint("CM_Client.msgRcvd", AUTOMATIC);
               case CRR(sin, rin):          ⟨ if (status = OPENING ∧ rin = lin){
                                                  status := OPEN;
                                                  din := sin;
                                                  Send msg CRRACK(lin, din);
                                                  clientuser.connectRequestInd();
                                            } else if (status = OPEN ∧ sin = din ∧ rin = lin) {
                                                  Send msg CRRACK(lin, din);
                                            } else if (status = CLOSED || status = CLOSING)
                                               Send msg REJ(-,sin); ⟩
               case REJ(-,rin):             ⟨ if (status = OPENING  ∧ rin = lin){
                                                  status := CLOSED;
                                                  din := nil;  lin := nil;
                                                  clientuser.connectRequestRej();
                                            } else if (status = CLOSING ∧ rin = lin){
                                                  status := CLOSED;
                                                  din := nil;  lin := nil;
                                                  clientuser.disconnectRequestInd();
                                            } ⟩
                case DRACK (sin, rin):      ⟨  if (status = CLOSING ∧ rin = lin ∧ sin = din){
                                                  status := CLOSED;
                                                  din := nil;  lin := nil;
                                                  clientuser.disconnectRequestInd();
                                            } ⟩
       }
       //# breakpoint("CM_Client.CM_Receiver", END);
  } //End Thread CM_Receiver.
}
```

Figure 8.2: CM_Client system program in SeSF

Description of connection management protocol (server side):
Var status: $\{\text{CLOSED}, \text{OPENING}, \text{OPEN}\}$; initially CLOSED. Status of server's relationship with the client. CLOSED iff server has no incarnation involved with the client. OPENING means server has an incarnation accepting a connection request from the client. OPEN means server has an incarnation open to the client.
Var lin: $\{\text{nil}, 0, 1, ...\}$; initially nil. Local incarnation number. nil if $\text{status} = \text{CLOSED}$. Otherwise identifies server incarnation involved with the client.
Var din: $\{\text{nil}, 0, 1, ...\}$; initially nil. Distant incarnation number. nil if $\text{status} = \text{CLOSED}$. Otherwise identifies the incarnation of the client with which the server incarnation is involved.

```
system-program lass CM_Server{
  //# static HarnessInterface harness := (HarnessInterface) Naming.lookup("CMHarness");
   int status := CLOSED;              // status of server's relationship with the client
   int listening := false;            // equals true if the server is accepting incoming connections
   int lin := nil;                    // local incarnation number
   int din := nil;                    // distant incarnation number
   int linGen := 0;                   // incarnation number generator
   Timer rTimer;                      // retransmission timer, fires after timeout elapses
   boolean rTimerFired                // it is true whenever rTimer fires
   CM_ServerUser serveruser;          // reference to the user application for callback methods

    xc-event void listenRequest(){      xc-event void endListenRequest (){    lc-event CRR_Retransmission () {
        ec: true;                          ec:  true;                           ec: status = OPENING ∧ rTimerFired;
        ac: listening := true;             ac:  listening := false;             ac: Send msg CRR(lin,din);
    }                                   }                                            Reset rTimer of msg CRR;
                                                                                }
  Thread CM_Receiver {
    Receive msg {
    //# breakpoint("CM_Server.msgRcvd", AUTOMATIC);
      case CR(sin,-):             ⟨ if (status = CLOSED ∧ !listening){
                                      Send msg REJ(-,sin); // Not in accept mode
                                  } else if (status = CLOSED ∧ listening){
                                      lin := linGen++; // Attempted connection
                                      din := sin;
                                      status := OPENING;
                                      Send msg CRR(lin,din);
                                      Reset rTimer of msg CRR
                                      serveruser.distantRequestInd(sin);
                                  } else if (status = OPENING ∧ sin > din){ // new remote incarnation
                                      din := sin;
                                      Send msg CRR(lin,din);
                                      serveruser.distantRequestInd(sin);
                                  } ⟩
      case CRRACK(sin,rin):        ⟨ if (status = OPENING ∧ sin = din ∧ rin = lin){
                                      status := OPEN;
                                      serveruser.connectInd();
                                  } ⟩
      case DR(sin,rin):           ⟨ if (status = OPEN ∧ sin = din ∧       rin = lin){
                                      Send msg DRACK(lin,din);
                                      status := CLOSED;
                                      lin := nil; din := nil;
                                      serveruser.closeInd();
                                  } else if (status = CLOSED)
                                      Send msg DRACK(rin,sin);
      case REJ(-,rin):            ⟨ if (status = OPENING ∧ rin = lin){
                                      status := CLOSED;
                                      lin :=  nil; din := nil;
                                      serveruser.listenInd();
                                  } ⟩
    }
    //# breakpoint("CM_Server.CM_Receiver", END);
  } //End Class CM_Receiver.
} //End Class CM_Server
```

Figure 8.3: CM_Server system program in SeSF

sets CM_Client.sinkuser to refer to itself (for callback methods). The handshake sequences of connection establishment operate as follows:

- CM_ServerUser instructs CM_Server to accept incoming connection requests via xc-event CM_Server.listenRequest(), and to reject incoming connection requests via CM_Server.endListenRequest().
- CM_ClientUser requests connection establishment via CM_Client.connectRequest(). CM_Client creates a new incarnation x0, and sends CR(x0, −) to CM_Server.
- When CM_Server receives CR(x0, −):
    - If it is accepting incoming connections, it informs CM_ServerUser of the arrival of the connection request via CM_ServerUser.distantRequestInd(x0), creates a new incarnation y0, and replies with CRR(y0, x0).
    - If it is not accepting incoming connections, it replies with REJ(−, x0).
- If CM_Client receives CRR(y0, x0), it informs the user of the connection establishment via CM_ClientUser.connectRequestInd(), and replies with CRRACK(x0, y0).
- If CM_Client receives REJ(−, x0) or its timeout fires before it receives a response, it calls CM_ClientUser.connectRequestRej() to inform the user of the failure to establish a connection.
- When CM_Server receives CRRACK(x0, y0), it informs the user of the connection establishment via CM_ServerUser.connectInd(). If CM_Server receives REJ(−, y0), it informs the user of the connection cancellation via CM_ServerUser.listenInd().

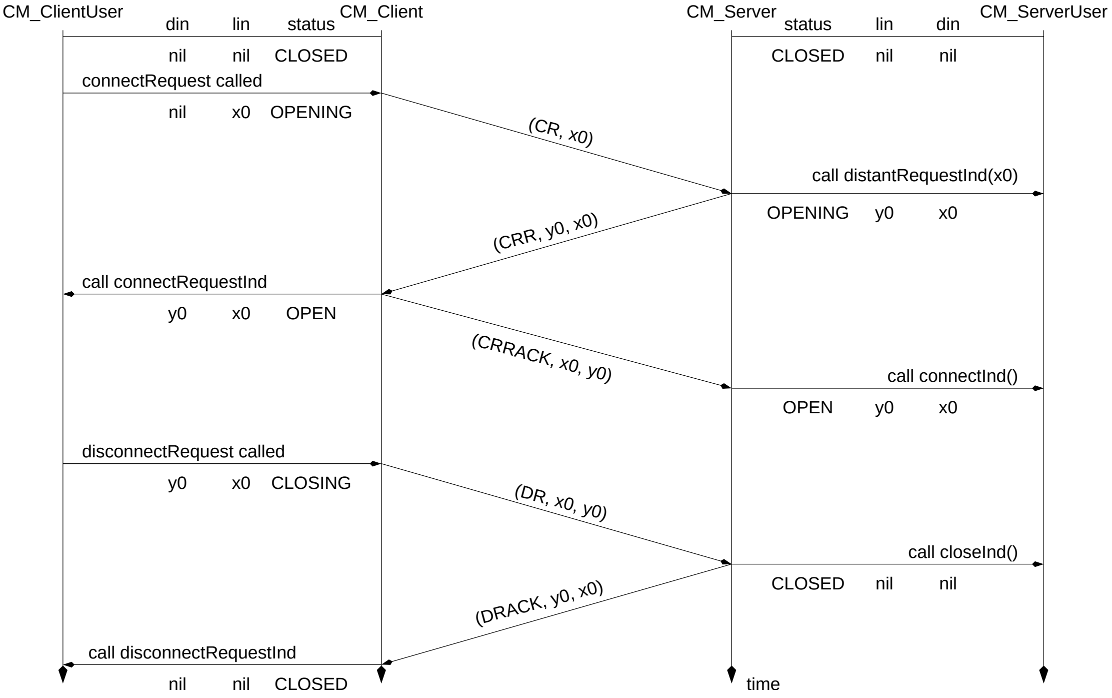


Figure 8.4: Successful connection and disconnection scenario

The handshake sequences of connection termination operate as follows:

- CM_ClientUser requests connection termination via CM_Client.disconnectRequest(). CM_Client sends DR(x0, y0) to CM_Server.
- When CM_Server receives DR(x0, y0), it informs CM_ServerUser of the connection termination via CM_ServerUser.closeInd(), and replies with DRACK(y0, x0).
- When CM_Client receives DRACK(y0, x0), it informs the user of the connection termination via CM_ClientUser.disconnectRequestInd.

Both systems assume the standard progress assumptions, that is, weak fairness of all threads. Figure 8.4 illustrates a successful scenario of connection establishment and termination.

8.2 Service

Figures 8.5 and 8.6 illustrate the CM service program in SeSF. The service defines the following variables:

- $\mathsf{cStatus}$: $\{\mathsf{CLOSED}, \mathsf{OPENING}, \mathsf{OPEN}, \mathsf{CLOSING}\}$; initially CLOSED. Status of client's relationship with the server. CLOSED iff client has no incarnation involved with the server. $\mathsf{OPENING}$ means client has an incarnation requesting a connection with the server. OPEN means client has an incarnation open to the server. $\mathsf{CLOSING}$ means client has an incarnation closing a connection with the server.

- clin: $\{-1, 0, 1, \ldots\}$; initially -1. Number of client's local incarnations minus 1, that is, the number of times that the client has requested a connection establishment minus 1. The “-1” indicates the nil.

- cdin: $\{-1, 0, 1, \ldots\}$; initially -1. Equals the value of the server's local incarnation during the client most recent transition to state OPEN, that is, the last time a connection was established (at client side). The “-1” indicates the nil.

- $\mathsf{sStatus}$: $\{\mathsf{CLOSED}, \mathsf{OPENING}, \mathsf{OPEN}\}$; initially CLOSED. Status of server's relationship with the client. CLOSED iff server has no incarnation involved with the client. $\mathsf{OPENING}$ means server has an incarnation accepting a connection request from the client. OPEN means server has an incarnation open to the client.

- $\mathsf{sAccepting}$: $\{\mathsf{REJECT}, \mathsf{ACCEPT}\}$; initially REJECT. Current status of the server, that is, whether it can accept connections or not.

- slin: $\{-1, 0, 1, \ldots\}$; initially -1. Number of server's local incarnations, that is, the number of times that the server has entered state OPENING. The "-1" indicates the nil.

- sdin: $\{-1, 0, 1, \ldots\}$; initially -1. Equals the value of the client's local incarnation when the last connection request was received by the server. The "-1" indicates the nil.

Table 8.1 illustrates the events used in the CM component. Client-side events are the interactions between CM_Client and CM_ClientUser. Server-side events are the interactions between CM_Server and CM_ServerUser. Figure 8.7 indicates the effect of client-side events on $<\mathsf{cStatus}>$. Figure 8.8 indicates the effect of server-side events on $<\mathsf{sStatus}, \mathsf{sAccepting}>$.

Service CM defines the following progress obligations:

P1 If client has requested a connection establishment and server is accepting connections, then eventually (1) server is state OPENING, (2) client's status is CLOSED, or (3) server rejects connections.

P2 If the client is OPENING and server is OPENING, then eventually (1) the client is OPEN, or (2) one or both entities' status are CLOSED.

P3 If the client is Open and the server is OPENING, then eventually (1) the client and the server are OPEN, or (2) one or both entities close the connection.

P4 If connectRequest() occurs, then either connectRequestInd() or connectRequestRej() will eventually be executed. The client cannot stay in state Opening forever.

```
service-program CM {
  // Client entity variables.
    int cStatus := CLOSED;      // Status of client's relationship with the server
    int clin := -1;             // Client's local incarnation number
    int cdin := -1;             // Client's distant incarnation number
  // Server entity variables.
    int sStatus := CLOSED;        // Status of server's relationship with the client
    int sAccepting := REJECT;     // Reflect whether server is accepting or rejecting incoming connections
    int slin := -1;               // Server's local incarnation number
    int sdin := -1;               // Server's distant incarnation number

 // client requests to connect to server
  dnw-event void CM_Client.connectRequest() {
     ec: cStatus = CLOSED;
     ac: cStatus :=OPENING;
         clin++;
  }

 // client user requests to disconnect.
  dnw-event CM_Client.disconnectRequest() {
     ec: cStatus = OPEN;
     ac: cStatus := CLOSING;
  }

 // client learns that its connection request to server is accepted;
 // client becomes open to server.
  upw-event void CM_ClientUser.connectRequestInd() {
     ec: cStatus = OPENING ∧ clin = sdin;
     ac: cStatus := OPEN;
         cdin := slin;
  }

 // client learns that its connection request to server is rejected.
  upw-event void CM_ClientUser.connectRequestRej() {
     ec: cStatus = OPENING;
     ac: cStatus := CLOSED;
  }

 //  client's request to disconnect is fulfilled.
  upw-event void CM_ClientUser.disconnectRequestInd() {
     ec: cStatus = CLOSING;
     ac: cStatus := CLOSED;
  }

 // server will accept incoming connections.
  dnw-event void CM_Server.listenRequest() {
     ec: sAccepting = REJECT;
     ac: sAccepting := ACCEPT;
  }

 //  server will not accept incoming connections.
  dnw-event void CM_Server.endListenRequest() {
     ec: sStatus = CLOSED ∧ sAccepting = ACCEPT;
     ac: sAccepting := REJECT;
  }

 //  server receives a connection request from client.
  upw-event void CM_ServerUser.distantRequestInd(int sin) {
     ec: sAccepting = ACCEPT ∧ sin ≤ clin;
     ac: if (sStatus = CLOSED) {
            slin++; // Attempt connection
            sdin := sin;
            sStatus := OPENING;
         } else if (sStatus = OPENING ∧ sin > sdin) {
            sdin := sin;
            sStatus := OPENING;
         }
  }
```

Figure 8.5: SeSF CM: connection management service program (Part 1)

```
  // server learns that the client's connection request has succeeded;
  // server becomes open.
   upw-event void CM_ServerUser.connectInd() {
      ec: sStatus = OPENING ∧ slin = cdin;
      ac: sStatus := OPEN;
   }

  // server learns that its connection with the client is closed.
   upw-event void CM_ServerUser.closeInd() {
      ec: sStatus = OPEN ∧ cStatus ! = OPEN;
      ac: sStatus := CLOSED;
   }

  // server learns of the rejection to the connection request, and goes to listening.
   upw-event void CM_ServerUser.listenInd() {
      ec: sStatus = OPENING;
      ac: sStatus := CLOSED;
   }

  // If client has requested a connection establishment and server is accepting connections,
  // then eventually (1) server is state OPENING, (2) client's status is CLOSED, or
  // (3) server rejects connections.
   progress-obligation P1 {
     (cStatus = OPENING ∧ sAccepting = ACCEPT)  leadsto
       (cStatus = OPENING ∧ sStatus = OPENING) ∨ cStatus = CLOSED ∨ sAccepting = REJECT;
   }

  // If the client is OPENING and server is OPENING, then eventually (1) the client is OPEN,
  // or (2) one or both entities' status are CLOSED.
   progress-obligation P2 {
     (cStatus = OPENING ∧ sStatus = OPENING)  leadsto
       (cStatus = OPEN ∧ sStatus = OPENING) ∨ cStatus = CLOSED ∨ sStatus = CLOSED;
   }

  // If the client is Open and the server is OPENING, then eventually (1) the client and
  // the server are OPEN, or (2) one or both entities close the connection.
   progress-obligation P3 {
     (cStatus = OPEN ∧ sStatus = OPENING)  leadsto
       (cStatus = OPEN ∧ sStatus = OPEN) ∨ cStatus ≠ OPEN ∨ sStatus = CLOSED;
   }

  // If connectRequest occurs, then either connectRequestInd or connectRequestRej
  // will eventually be executed. The client cannot stay in state Opening forever.
   progress-obligation P4 {
      cStatus = OPENING  leadsto cStatus = OPEN ∨ cStatus = CLOSED;
   }

  // If distantRequestInd occurs, then connectInd() or listenInd is eventually executed,
  // or  the server closes the connection. The server cannot stay in state OPENING forever.
   progress-obligation P5 {
      sStatus = OPENING  leadsto sStatus = OPEN ∨ sStatus = CLOSED;
   }

  // If the client is in state CLOSING, then the connection is eventually closed.
   progress-obligation P6 {
      cStatus = CLOSING  leadsto cStatus = CLOSED;
   }
}
```

Figure 8.6: SeSF CM: connection management service program (Part 2)

Client-side Events		
upw/dnw	Event	Indication
dnw	connectRequest	client requests to connect to server.
upw	connectRequestRej()	client learns that its connection request to server is rejected.
upw	connectRequestInd()	client learns that its connection request to server is accepted; client becomes open to server.
dnw	disconnectRequest()	client user requests to disconnect.
upw	disconnectRequestInd()	client's request to disconnect is fulfilled.
Server-side Events		
upw/dnw	Event	Indication
dnw	listenRequest	server will accept incoming connections.
dnw	endListenRequest	server will not accept incoming connections.
upw	distantRequestInd($\cdots$)	server receives a connection request from client.
upw	listenInd()	server learns of the rejection to the connection request, and goes to listening;
upw	connectInd()	server learns that the client's connection request has succeeded; server becomes open.
upw	closeInd()	server learns that its connection with the client is closed.

Table 8.1: Events of service CM

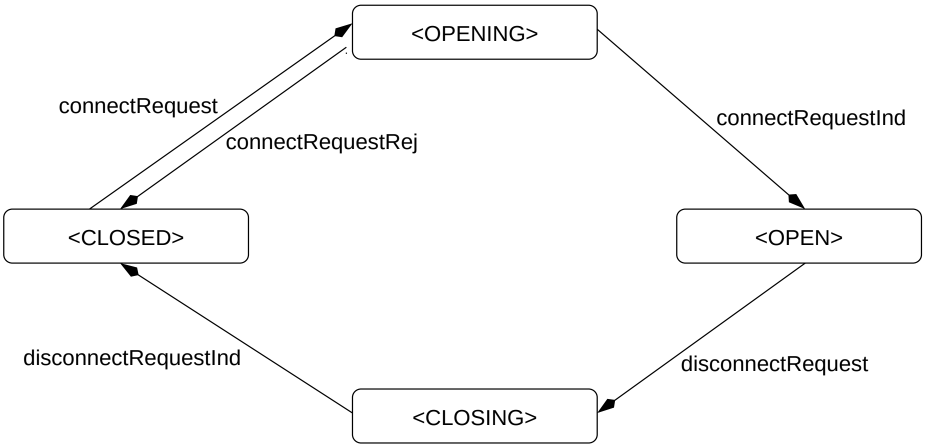


Figure 8.7: Effect of client-side events fo service CM on $<\mathsf{cStatus}>$

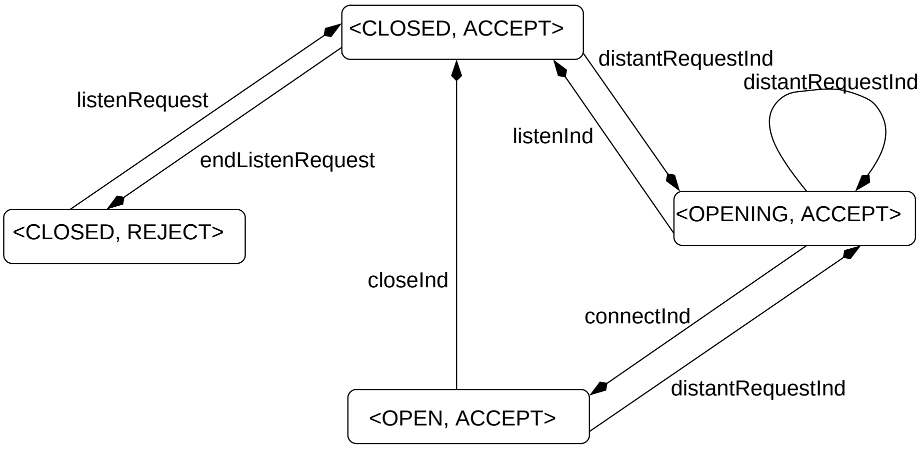
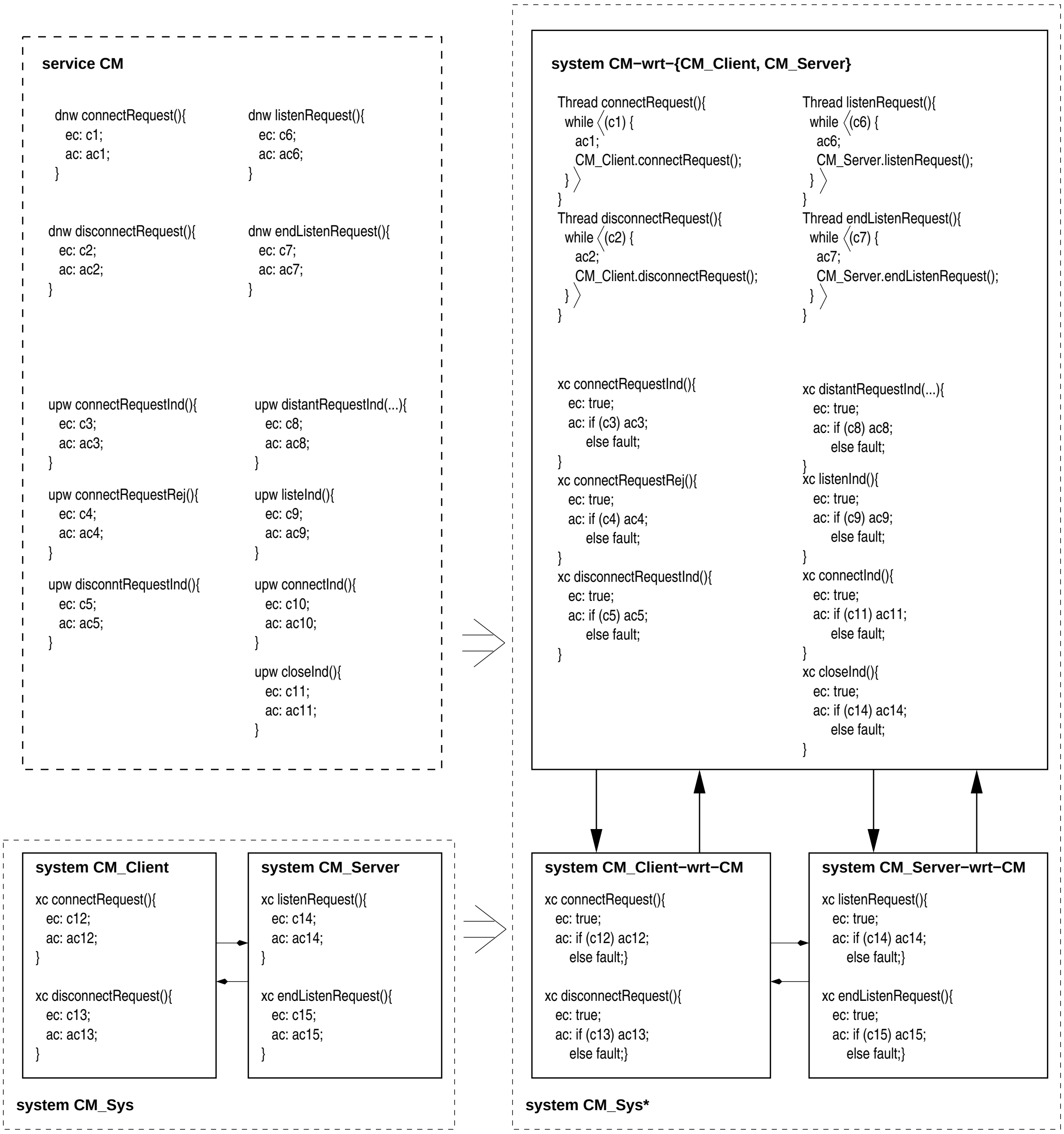


Figure 8.8: Effect of server-side events of service CM on $<\mathsf{sStatus}, \mathsf{sAccepting}>$

P5 If $\mathsf{distantRequestInd}(\cdots)$ occurs, then $\mathsf{connectInd}()$ or $\mathsf{listenInd}()$ is eventually executed, or the server closes the connection. The server cannot stay in state $\mathsf{OPENING}$ forever.

P6 If the client is in state $\mathsf{CLOSING}$, then the connection is eventually closed.



```
service CM

dnw connectRequest(){
   ec: c1;
   ac: ac1;
}

dnw listenRequest(){
   ec: c6;
   ac: ac6;
}

dnw disconnectRequest(){
   ec: c2;
   ac: ac2;
}

dnw endListenRequest(){
   ec: c7;
   ac: ac7;
}

upw connectRequestInd(){
   ec: c3;
   ac: ac3;
}

upw distantRequestInd(...){
   ec: c8;
   ac: ac8;
}

upw connectRequestRej(){
   ec: c4;
   ac: ac4;
}

upw listeInd(){
   ec: c9;
   ac: ac9;
}

upw disconntRequestInd(){
   ec: c5;
   ac: ac5;
}

upw connectInd(){
   ec: c10;
   ac: ac10;
}

upw closeInd(){
   ec: c11;
   ac: ac11;
}
```

```
system CM−wrt−{CM_Client, CM_Server}

Thread connectRequest(){
  while 〈(c1) {
     ac1;
     CM_Client.connectRequest();
  } 〉
}
Thread disconnectRequest(){
  while 〈(c2) {
     ac2;
     CM_Client.disconnectRequest();
  } 〉
}

Thread listenRequest(){
  while 〈(c6) {
     ac6;
     CM_Server.listenRequest();
  } 〉
}
Thread endListenRequest(){
  while 〈(c7) {
     ac7;
     CM_Server.endListenRequest();
  } 〉
}

xc connectRequestInd(){
   ec: true;
   ac: if (c3) ac3;
        else fault;
}
xc connectRequestRej(){
   ec: true;
   ac: if (c4) ac4;
        else fault;
}
xc disconnectRequestInd(){
   ec: true;
   ac: if (c5) ac5;
        else fault;
}

xc distantRequestInd(...){
   ec: true;
   ac: if (c8) ac8;
        else fault;
}
xc listenInd(){
   ec: true;
   ac: if (c9) ac9;
        else fault;
}
xc connectInd(){
   ec: true;
   ac: if (c11) ac11;
        else fault;
}
xc closeInd(){
   ec: true;
   ac: if (c14) ac14;
        else fault;
}
```

```
system CM_Client

xc connectRequest(){
   ec: c12;
   ac: ac12;
}

xc disconnectRequest(){
   ec: c13;
   ac: ac13;
}
```

```
system CM_Server

xc listenRequest(){
   ec: c14;
   ac: ac14;
}

xc endListenRequest(){
   ec: c15;
   ac: ac15;
}
```

```
system CM_Client−wrt−CM

xc connectRequest(){
   ec: true;
   ac: if (c12) ac12;
      else fault;}

xc disconnectRequest(){
   ec: true;
   ac: if (c13) ac13;
      else fault;}
```

```
system CM_Server−wrt−CM

xc listenRequest(){
   ec: true;
   ac: if (c14) ac14;
      else fault;}

xc endListenRequest(){
   ec: true;
   ac: if (c15) ac15;
      else fault;}
```

Figure 8.9: Service satisfaction transformations

8.3 CM satisfaction conditions

Fig. 8.9 illustrates the construction of CM_Sys* from CM_Sys and CM. CM_Sys* consists of of CM_Client-wrt-CM, CM_Server-wrt-CM and CM-wrt-{CM_Client, CM_Server} In particular, every output call in CM_Client and CM_Server is replaced by a call to the corresponding event of CM by appropriately modifying variables clientuser and serveruser. The safety condition for CM_Sys offers CM reduces to the following:

1. CM_Sys* does not have undefined values or operations (division by zero, signature-inconsistent call, type mismatch, etc.).

2. CM_Sys* does not call a disabled event, which reduces to the following predicates being invariant:

   - CM.connectRequest.ec ⇒ CM_Client.connectRequest.ec

     (This formalizes the constraint that CM_Client.connectRequest should be enabled whenever its user calls CM.connectRequest. The predicates below are similarly obtained.)

   - CM.disconnectRequest.ec ⇒ CM_Client.disconnectRequest.ec
   - CM.listenRequest.ec ⇒ CM_Server.listenRequest.ec
   - CM.endListenRequest.ec ⇒ CM_Server.endListenRequest.ec
   - CM_Client at clientuser.connectRequestInd() ⇒ CM.connectRequestInd.ec
   - CM_Client at clientuser.connectRequestRej() ⇒ CM.connectRequestRej.ec
   - CM_Client at clientuser.disconnectRequestInd() ⇒ CM.disconnectRequestInd.ec

- $\mathsf{CM\_Server}$ at $\mathsf{serveruser.distantRequestInd(\cdots) \Rightarrow CM.distantRequestInd.ec}$
- $\mathsf{CM\_Server}$ at $\mathsf{serveruser.listenInd() \Rightarrow CM.listenInd.ec}$
- $\mathsf{CM\_Server}$ at $\mathsf{serveruser.connectInd() \Rightarrow CM.connectInd.ec}$
- $\mathsf{CM\_Server}$ at $\mathsf{serveruser.closeInd() \Rightarrow CM.closeInd.ec}$

The progress condition holds iff $\mathsf{CM\_Sys}^*$ satisfies progress obligations $\mathsf{P1} \sim \mathsf{P6}$ assuming weak fairness of CM_Sys's threads.

Although we do not do so here, it would be straightforward to prove by assertional reasoning that these conditions hold (e.g., as in [76]).

8.4 Testing and assertion checking harness

To test CM_Sys against CM, we do the following:

1. Create a Harness process to control the test execution. The Harness process is bound to RMI (Remote Method Invocation in Java) port "CMHarness".

2. Construct from $\mathsf{CM\_Sys}^*$ a composite system $\mathsf{CM\_Sys}^{*\prime}$ which is to interact with the harness. $\mathsf{CM\_Sys}^{*\prime}$ consists of $\mathsf{CM\_Source\text{-}wrt\text{-}CM}'$ (a version of CM_Client-wrt-CM that interacts with the harness), $\mathsf{CM\_Server\text{-}wrt\text{-}CM}'$ (a version of CM_Server-wrt-CM that interacts with the harness), the channels between them, and CM-wrt-$\{\mathsf{CM\_Client}, \mathsf{CM\_Server}\}'$ (a version of CM-wrt-$\{\mathsf{CM\_Client}, \mathsf{CM\_Server}\}$ that interacts with the harness).

3. Execute $\mathsf{CM\_Sys}^{*\prime}$ along with (and under the control of) the harness process.

4. Check whether the generated execution becomes faulty.

Section 8.4.1 describes how to obtain $\mathsf{CM\_Sys}^{*\prime}$. Section 8.4.2 describes how to execute $\mathsf{CM\_Sys}^{*\prime}$ under the control of the harness process.

8.4.1 Constructing $\mathsf{CM\_Sys}^{*\prime}$

The first step is to construct composite system $\mathsf{CM\_Sys}^{*\prime}$ (figure 8.10). Section 8.3 described how to get CM_Client-wrt-CM, CM_Server-wrt-CM and CM-wrt-{CM_Client, CM_Server}. In addition to those modification, we need these components to connect to the harness. This leads to the following modifications:

- Construct CM_Client-wrt-CM′, referred to as CM_Client′, from CM_Client-wrt-CM as follows:
  - Tag //#HarnessInterface harness = . . . ; indicate the location of the harness, i.e., its RMI port.
  - For every xc event, (1) insert a call to method checkAssertions which sends data necessary for assertion checking to SAC module, and (2) log information to the log file.
  - Insert breakpoints at locations specified by tag //#breakpoint.
- Similarly, construct CM_Server-wrt-CM′, referred to as CM_Server′, from CM_Server-wrt-CM.
- Construct CM-wrt-{CM_Client, CM_Server}′, referred to as CM′, from CM-wrt-{CM_Client, CM_Server}′ as follows. For every upw/dnw event, insert a call to

method checkAssertions, and log information to log file.

- Construct CM_Sys*′ of CM_Client′, CM_Server and CM′.

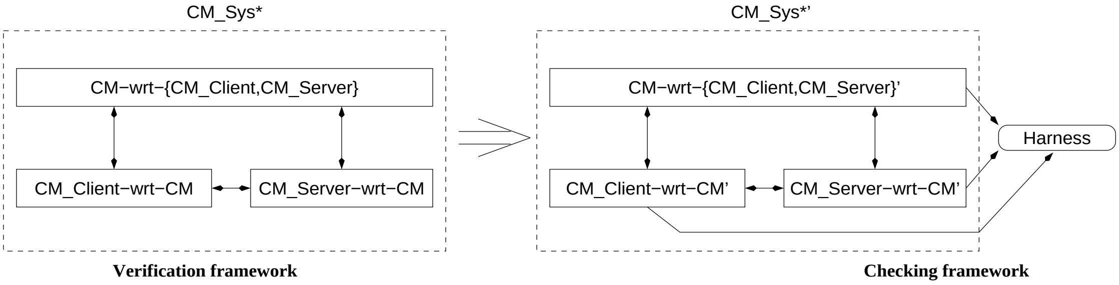


Figure 8.10: CM_Sys* and CM_Sys*′ composite systems

8.4.2 Executing CM_Sys*′

Once CM_Sys*′ is constructed, the next step is to obtain the testing platform on which it can be executed. SAC (Serializer And Checker) module, within the harness, ensures that CM_Sys′-CM′ interactions are executed atomically, and that only one thread is proceeding at a time. SeSFJava harness inserts breakpoints in CM_Sys′ and CM′ such that at any time, at most one thread of CM_Sys*′ runs and every other thread is paused at a breakpoint. SAC module maintains relevant state for every process, such as whether the process is running, paused, blocked, or about to be terminated. Each thread sends its state to the SAC module. Breakpoints are inserted manually to indicate where the thread transitions take place.

Assertions are evaluated at checking locations, specifically, at the start of every event and at every breakpoint as mentioned in section 6.3.

After CM_Sys*′ is constructed, it is executed on the same platform as CM_Sys* as follows:

1. SeSFJava harness starts as a separate process, and binds itself to RMI port “CMHarness”.

2. CM' process starts, and looks up for the harness’s port “CMHarness”.

3. $\mathsf{CM\_Sys}'$ process is created, and it looks up for port “CMHarness” using RMI lookup command. So, both systems (source and sink) are hooked up with the harness.

4. The developer can use the harness either in batch mode, letting the harness run for a while and then analyzing the log file, or in interactive mode, influencing the flow of the execution manually.

Chapter 9

Educational Use of SeSFJava

SeSFJava has been used in teaching an introductory senior-level network course (CMSC417) at University of Maryland. The goal of the programming assignments in this course is to teach the students the following:

- The role of network protocols.
- The different roles of the layers of the network and how they stack above each other.
- How to enhance the performance of the network in the face of changing network conditions.
- How to implement a distributed multi-threading applications, for example, client-server or peer-to-peer applications.

In fall 1999, we introduced a three-phase project that takes the above goals into account. The project was to implement a transport protocol providing client and server TCP sockets. Phase I implements a data transfer protocol. Phase II implements congestion control in order to enhance the performance of the data transfer protocol. Phase III implements the connection management and the two-way data transfer protocols of TCP/IP.

All project specifications were described informally, and test cases were provided.

During the course, a number of problems emerged. Some students misunderstood the specification or oversimplified it to just fit the test cases provided with the project assignment. Other students did not test their projects thoroughly with various inputs. Others did not finish the project because they did not budget enough time, especially in phase III which involved much more work than the other two phases. The teaching assistants (TAs) spent excessive time in testing and grading the student projects.

These problems prompted us to integrate SeSFJava into the networks course. SeSFJava (and formal methods in general), in theory, removes all misunderstandings about the project specifications. The harness provides technique to test the projects extensively on the actual platform, which helps the students to discover more bugs.

Here, the students do not have to learn a new formal language, as the specifications are written in Java which is familiar to the students. The Harness depends on runtime monitoring, a concept understandable by most students (as opposed to model checking for example). SeSFJava and the Harness can be learned under the tight time constraints of the semester.

The transport-protocol project is divided into four phases. Each phase is independently tested for correctness. Section 9.1 describes phase I, which is the data transfer protocol. Section 9.2 describes phase II, which focuses on the performance of the data transfer protocol. Section 9.3 describes phase III, which is the connection management protocol. Section 9.4 describes phase IV, which puts it all together (connection management plus two-way data transfer). Section 9.5 describes our experience with the students.

9.1 Phase I: Data transfer protocol (correctness)

In this phase, the student implements a protocol that achieves reliable data transfer over unreliable network channels. Specifically, the project consists of two interacting programs, a Source and a Sink, as shown in fig. 9.1. The Source consists of three components: SW_SourceUser, SW_Source and NetworkSocket. The Sink consists of three components: SW_SinkUser, SW_Sink and NetworkSocket.

The students are provided with:

- The applications, SW_SourceUser and SW_SinkUser, which transfer a file from the source to the sink.
- The NetworkSocket entity which provides the unreliable channels to be used by the transport entities. NetworkSocket entity is a wrapper to the standard sockets. It is used instead of the usual UDP sockets, because in a LAN environment, the standard sockets display hardly any loss, reordering or duplication. The students can change the probabilities of loss, reordering and duplication on the fly, which is important for testing.
- The SeSFJava Harness module and the data transfer service specification.

The students are to implement SW_Source and SW_Sink so that they conform to the provided data transfer service. The students are free to choose the particulars of the design, including message types and formats, sequence number space, data block size, retransmission policy, acknowledgment (cumulative and/or selective) policy, round-trip time estimator, etc.

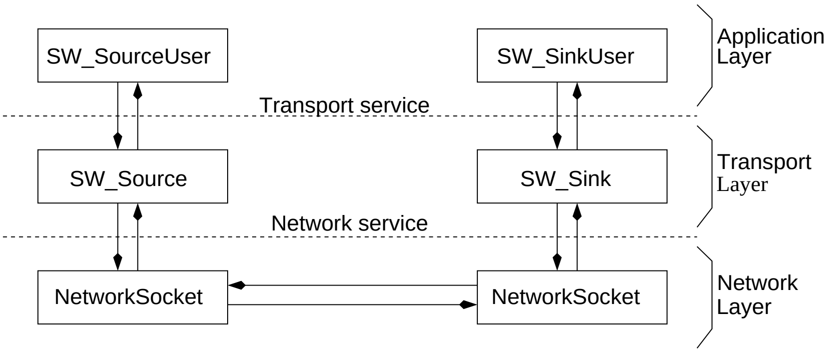


Figure 9.1: Phase I overview

9.1.1 Testing phase I

To participate in the testing, the system and service programs need to be instrumented using SeSFJava Preprocessor in order connect services and systems to the Harness. This includes issues like connecting to the Harness, checking event enabling conditions, inserting breakpoints, etc. Instead of letting the students use the preprocessor to generate the Harness, services and systems, we gave the students the preprocessed code, thereby relieving them of the preprocessing hassle. The preprocessed code include the following:

- A simpler version of SeSFJava Harness which is encapsulated in a single class that contains the following:
  - A constructor that binds the Harness to Remote Method Invocation (RMI) port “Harness”.
  - Lock and unlock methods for the Harness main lock, for synchronizing the programs and threads of the project. When a thread acquires the main lock, no other thread in the network system can proceed, until the lock is released.
  - Methods that represent the interactions between the transport layer and the application layer (as described in chapter 7).

  - Invariants of the data transfer protocol, for example, the number of bytes delivered to Sink's user cannot exceed the number of bytes sent by Source's user.

- The application systems (SourceUser and SinkUser) where the xc-events are already modified to check the enabling conditions. Statements that connect the application to the Harness are already instrumented.

- Templates of the transport layer systems (SW_Source and SW_Sink) which include statements that lookup for the Harness RMI port, and the structure of the xc-event methods. For example, SW_Source.sendData method appears in the template as follows:

```
// Inside SW_Source.java
void sendData (byte []data) throws Exception {
     harness.lock();               // obtain Harness main lock
     harness.sendData(data);       // RMI call of Harness method with same parameters
     . . .                         // Student inserts sendData method body here
     harness.unlock();             // release Harness main lock
}
```

Consequently, a student can determine the correctness of both source and sink sides by checking that no errors were thrown during the execution of Harness. (To detect deadlocks, we add an extra condition: a file sent by the source has to be received.)

The program is executed as follows: (1) Execute the Harness module, so it can bind to port "Harness", (2) Execute the sink side so it can hook to the Harness class, (3) Execute the source side to start sending the file. A log file is recorded for every execution.

9.1.2 Grading phase I

The TAs grade the data in a semi-mechanical way. They run scripts to execute the projects with different input files and different network conditions. Each execution is stored in a log file, which is checked for thrown errors. If there is an error, the TA checks the log file to print out the trace that has generated this error, and determines the grade accordingly. The student can resort to a very simple solution, say a send window size of 1, but they will then suffer in Phase II.

9.2 Phase II: Data transfer protocol (performance)

This phase emphasizes the protocol's performance; that is, the grade is primarily based on the throughput achieved under varying network conditions, which in turn depends on how well the protocol adapts to congestion, the overhead of the congestion control mechanism, etc.

The students strip the RMI calls inserted in Phase I, and enhance their code to perform better. Enhancements are of two kinds: (1) network optimizations, for example, adding Tahoe congestion control, and (2) code optimizations, for example, reducing the thread-switching in their code. In this phase, the NetworkSocket has the ability to play scenarios that emulate real-life network traffic. Thus, the students can view how their code performs under various conditions.

The TAs grade this project by running scripts that execute the students projects a number of times for every test scenario, and record the throughput for each run. The average throughput is computed and the students are classified according to the performance into four groups, from fast to slow, and the grade is determined accordingly.

9.3 Phase III: Connection management protocol

In this phase, the students build a connection management protocol over unreliable network channels. The grade in this phase is primarily based on the protocol's correctness (as described in chapter 8). Specifically, the project consists of two interacting programs, a Client and a Server, as shown in fig. 9.2. Client consists of three components: ClientUser, CM_Client and NetworkSocket. Server consists of three components: ServerUser, CM_Server and NetworkSocket.

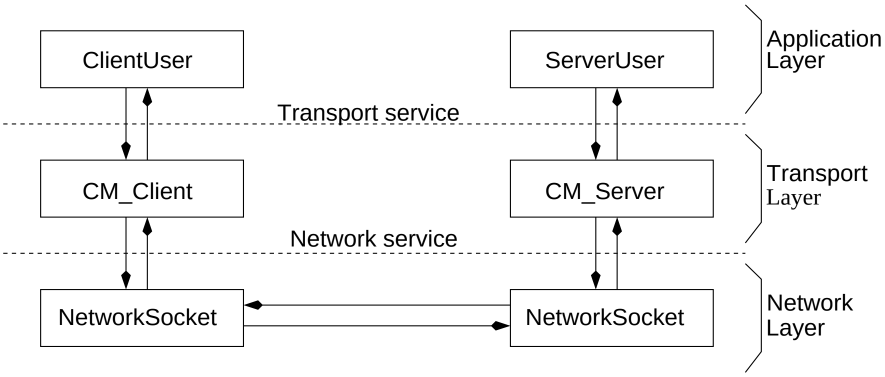


Figure 9.2: Phase III overview

The students are to implement CM_Client and CM_Server which are the transport entities at the two ends. They are provided with the other entities. ClientUser and ServerUser are the users of the transport entities. These applications open and close hundreds of connections under different circumstances. The pair of NetworkSockets are as in phases I and II. The specifications formally describe the three-way handshaking connection establishment, and the two-way handshaking of the disconnection procedure. Similar to that of phase I, the service specifications, the Harness, the application level systems, and the templates of the transport layer systems are provided in Harness file. The testing and grading are carried out similarly to that of phase I.

9.4 Phase IV: Putting it all together

In this phase, the students build a full-fledged transport service over unreliable network channels, specifically, combining phases II and III (after stripping the RMI calls). The grade of this project is based on the correctness and the performance of the students' implementations.

9.5 Experience with the students

We have been using SeSFJava in the senior-level undergraduate computer networks course for the past three years. The projects are mandatory: no student can pass the course without passing the projects. The average number of students per class is 50. Most students have not been exposed to formal methods before taking this course.

Using SeSFJava significantly improves the performance of the students. Table 9.1 compares the use of detailed informal description of the projects (without SeSF) against the use of SeSF in specifying these projects. The number of students who completed all the phases of their projects almost doubled. Their questions about the specifications decreased by 40%. The student drop rate decreased by almost half.

From the TA perspective, using SeSF reduces the grading time per student, because considerable amount of the grading is carried mechanically. The number of regrading

	Without SeSF	With SeSF	Improv.
% of students who completed their projects	45%	88%	95%
# of email queries per students	16	10	60%
% of students dropping the class	27%	14%	93%

Table 9.1: Improvement using SeSFJava

requests fell by 60%. We think this is because a student can test his/her implementation against the project specification, and because the TA provides the student with the trace demonstrating any errors (and thus grade penalties).

Chapter 10

Peer-to-Peer Network: Gnutella

In the past few years, many peer-to-peer network specifications have been introduced, for example, Gnutella [19, 40], Napster [38], Kazaa [36], Chord [82], NICE [8, 47] and Freenet [16]. For each of these specifications, many implementations become available, for example, Gnutella implementations include Limewire [37], Furi [87], and JTella [53]. Because these specifications are informal, developers interpret “holes” and ambiguities in different ways, resulting in different interpretations of the specifications.

In this chapter, our goal is to (1) formally define the services of a peer-to-peer network protocol in SeSFJava; and (2) apply the testing harness to an open-source implementation of the peer-to-peer protocol to test whether it conforms to the defined services.

We focus on the Gnutella protocol [19, 40]. We chose Gnutella because it is the most prevalent peer-to-peer system in the world (with 25 million users), and many open-source implementations are available. Gnutella is a decentralized peer-to-peer file sharing protocol. Gnutella uses the TCP service below, and provides join/depart, query, node discovery and upload/download services to the application level above. Figure 10.1 illustrates the Gnutella protocol stack. Each Gnutella node is referred to as a *servent*.

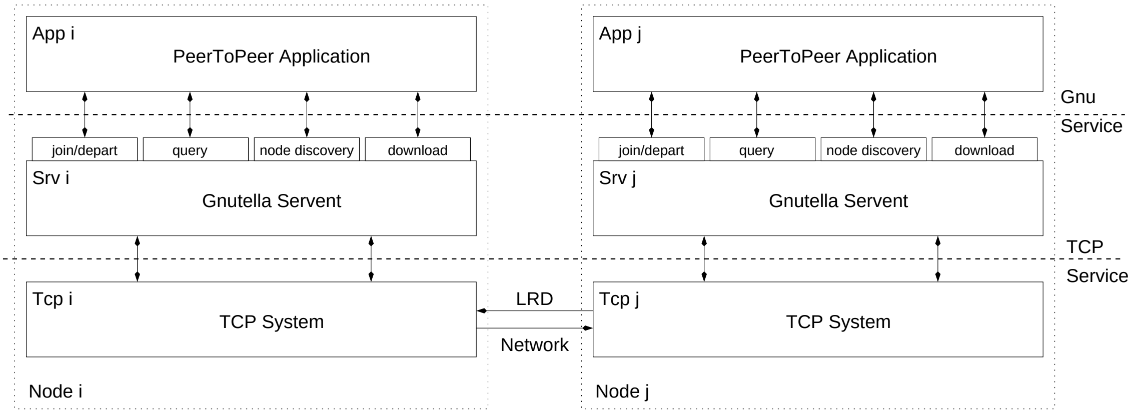


Figure 10.1: Gnutella protocol stack

In this chapter, we define the service provided by Gnutella, referred to as Gnu service. We also consider a particular Gnutella implementation, namely Furi, and apply the Harness to test whether Furi offers Gnu.

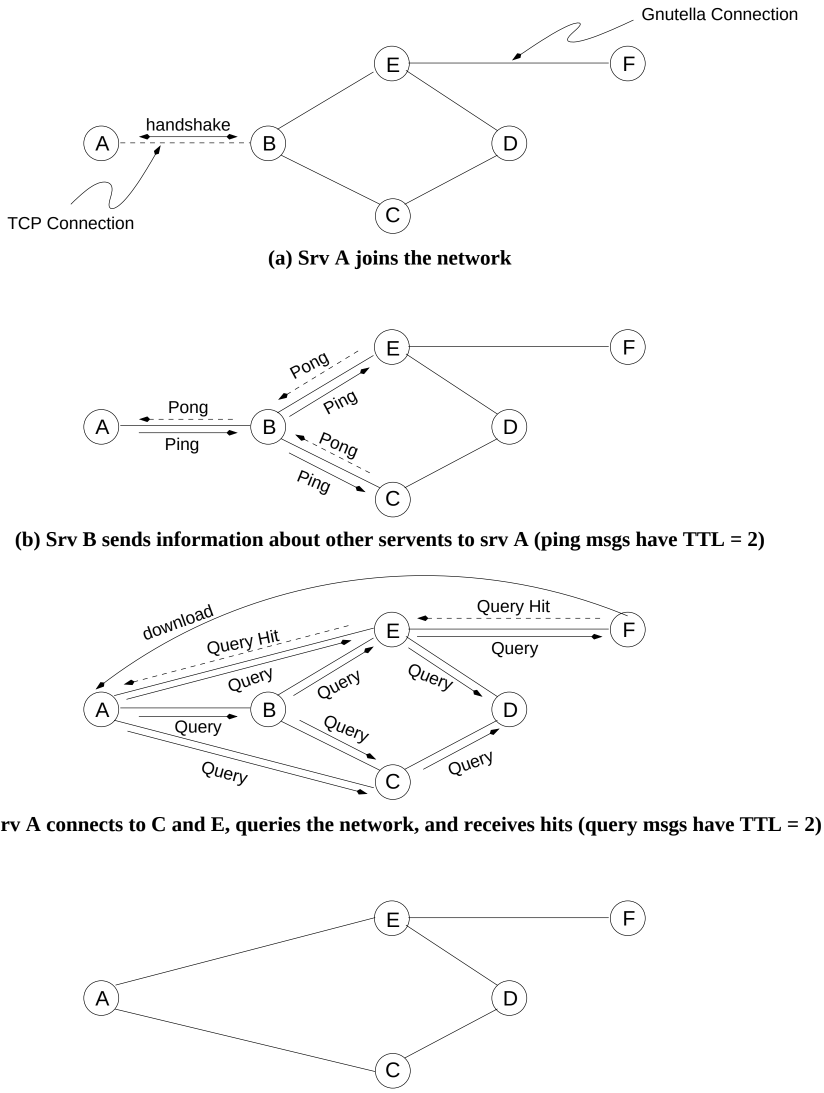
We could also test whether Furi correctly uses TCP service, which has been defined in earlier chapters. But this is rather trivial and not interesting. Instead, we define the special case of the TCP service as it is used by Gnutella, that is, the messages and control exchanged between the Gnutella layer and the TCP layer, and the permissible sequences of these exchanges. We refer to this as the *internal* service Gnu_TCP. We apply the Harness to test whether Furi satisfies this Gnu_TCP internal service.

The remainder of the chapter is organized as follows. Section 10.1 gives an overview of the operation of a Gnutella network. Section 10.2 defines the Gnu service. Section 10.3 defines the Gnu_TCP internal service. Section 10.4 explains how SeSFJava Harness can test an open-source Gnutella system against the services.

10.1 Gnutella overview

A Gnutella network [19, 40] is a dynamic overlay on the top of TCP network. Each node, or *servent*, can have connections to a number of servents. A servent sends a message by flooding it to its neighbors (ones with direct connection to it) and those neighbors flood to their neighbors until the message's TTL (time to live) ends. A servent is identified by an address, which consists of the host machine id (IP address or domain name) and the port that it uses for listening to incoming connections.

Figure 10.2 illustrates an example of how a Gnutella network works. Srv A wants to join the Gnutella network. It knows the address of srv B, one of the servents of the network, from a prior connection. Srv A establishes a Gnutella connection with srv B via a handshake sequence exchanged through a TCP channel (figure 10.2(a)). After establishing the Gnutella connection, srv A pings the network for information about more servents via ping messages. Srv B sends information about the connected servents (C and E) back to srv A (figure 10.2(b)) using pong messages. From this information, srv A knows about servents C and E, and connects to both of them as in figure 10.2(c). When srv A wants to locate file sesf.pdf, it sends a query message, which includes string "sesf.pdf", to its neighbors B, C and E. Each of these neighbors forwards the message to its neighbors (figure 10.2(c)). Srv F is the only servent that has hits to this file. So when srv F receives the query, it replies with the query hit message. When srv A receives the query hit message, it downloads the file using http protocol. When srv B decides to leave, it closes all its Gnutella connections, leaving the network as shown in figure 10.2(d).



(a) Srv A joins the network

(b) Srv B sends information about other servents to srv A (ping msgs have TTL = 2)

(c) Srv A connects to C and E, queries the network, and receives hits (query msgs have TTL = 2)

(d) Srv B departs the network

Figure 10.2: Example of a Gnutella network

10.1.1 Joining the Gnutella network

Initially, a servent (client servent in this case) gets the address of a servent on the network (server servent in this case) by searching public databases or by extracting addresses from recent connections. Next, it (client servent) connects to this server servent via a handshake sequence, afterwhich it is connected to the Gnutella network. The handshake sequence operates as follows:

1. The client establishes a TCP/IP connection to the server. If this step fails, the client considers it as a rejection to the connection attempt.

2. The client sends an ASCII string, "GNUTELLA CONNECT/<protocol version> \n\n", followed by its capability headers. A capability header, which is an ASCII string terminated by a new line, indicates a feature supported by the sender.

3. The server replies by sending to the client the ASCII string, "GNUTELLA/<protocol version> 200 OK\n\n" followed by its capability headers. Any other reply by the server indicates the rejection of the connection and the attempt ends here.

4. If the client is satisfied with the server's capability headers, it sends the ASCII string, "GNUTELLA 200 OK\n\n", back to the server. Any other reply indicates that the client is rejecting the connection and the attempt ends here.

Figure 10.3 illustrates an example of a successful connection.

After the handshake sequence is successfully completed, the servent exchanges binary messages with the rest of the network for various purposes: discovering new servents, querying the network for certain criteria, and sending files for servents behind

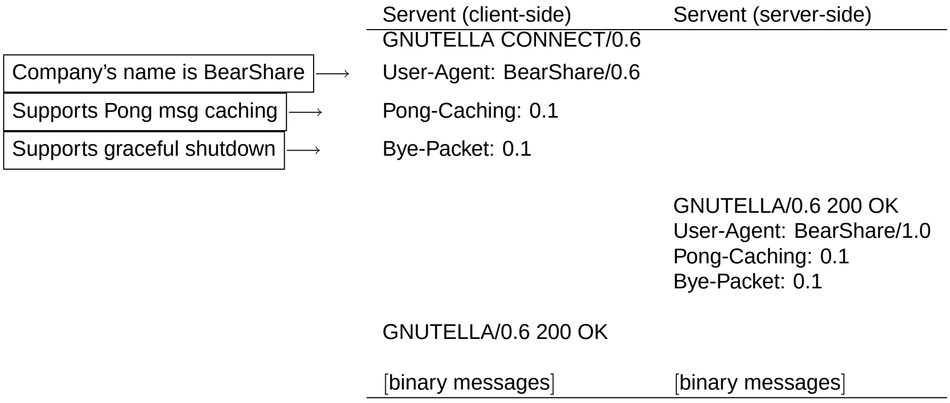


Figure 10.3: Example of a successful connection scenario in Gnutella 0.6

proxies (seldom used).

A Gnutella servent can exit either abruptly by shutting down, or gracefully by sending a Bye message and then shutting down after a specified timeout.

10.1.2 Gnutella binary messages

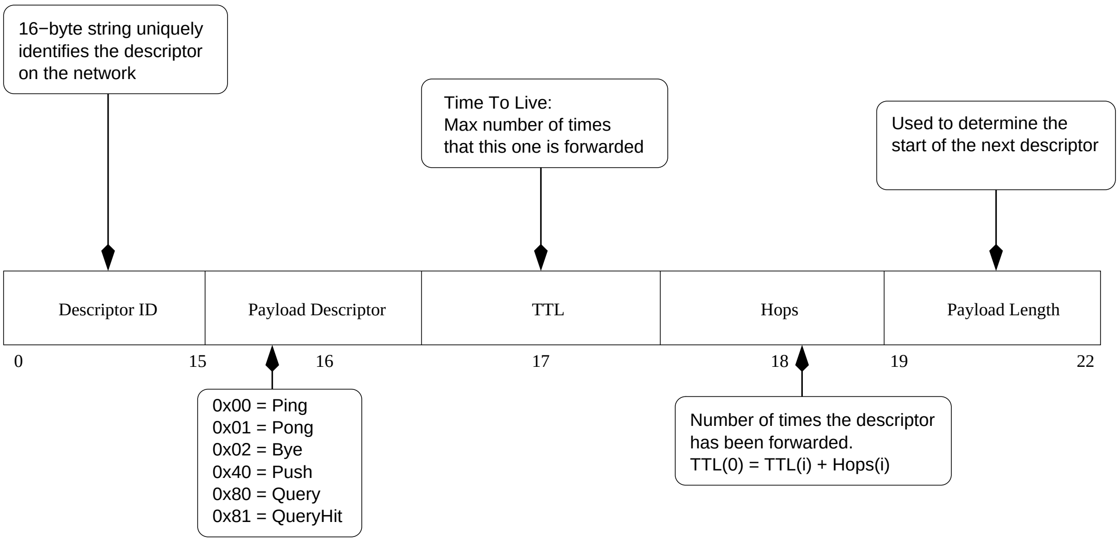

16−byte string uniquely identifies the descriptor on the network
Time To Live: Max number of times that this one is forwarded
Used to determine the start of the next descriptor
Descriptor ID
Payload Descriptor
TTL
Hops
Payload Length
0
15
16
17
18
19
22
0x00 = Ping
0x01 = Pong
0x02 = Bye
0x40 = Push
0x80 = Query
0x81 = QueryHit
Number of times the descriptor has been forwarded.
TTL(0) = TTL(i) + Hops(i)

Figure 10.4: Gnutella message structure

Figure 10.4 shows the structure of the binary messages. Each message begins with a 16-byte descriptor ID that uniquely identifies this message, and is generated based on the host IP and its port. This is followed, in Gnutella (ver. 0.6), by a payload descriptor which specifies the following messages:

- Ping/Pong *Messages*: These are used for servent discovery. A servent sends Ping messages periodically in order to probe the network for other servents. Whenever a servent receives a Ping message, it responds with a Pong message which includes its IP address and listening port. The Pong message's descriptor ID must equal that of the corresponding Ping message.

- Bye *Message:* The Bye message can be used only if the handshake capability header includes string "Bye-Packet: 0.1". A Bye message must be sent with TTL = 1. The receiver of the Bye closes the connection immediately. The sender must wait for few seconds before closing down the connection. All incoming messages during this period must be discarded.

- Query/QueryHit *Messages*: These are used for locating files. A servent queries by flooding the network with Query message that includes a criteria string (a string terminated by char 0x00). When a node receives a query, if it has hits (file indexes that meet the search criteria of the Query message), it replies by sending a QueryHit message, which includes those hits, through the network. If it has no hits for the query, it discards the request. When the sender receives a QueryHit for a query it has sent, it passes that hit to the servent application. The descriptor ID of a QueryHit must equal that of the corresponding Query message.

  The query requester will not receive a response if either the files requested are not available or the paths between the requesting server and the potential responders are broken.

- *Push Message*: A servent sends a Push message to trigger the download process for users behind a proxy. This message is seldom used.

The routing protocol of Gnutella network is simple. Whenever a servent wants to send a message, it sends the message to all the servents connected to it. Whenever a servent receives a message and the message's TTL (Time-To-Live) is greater than zero, it decrements the TTL, and sends the message to all its connected servents, except the one it received the message from. The servent does not forward any message with TTL equal to zero.

10.2 The Gnu service

In this section, we define service Gnu. Section 10.2.1 describes the join/depart component, while section 10.2.2 describes the query component. Throughout the remainder of this chapter, we ignore node discovery and upload/download services for simplicity. We identify a node by the IP address of its host machine and the port it uses for listening to incoming connections. The domain of the class Node (defined below) is all the possible values of its attributes and the value null.

```
class Node {
    IPAddress    ipAddr;
    int          port;  // Port used for listening to incoming connections
}
```

10.2.1 Join/depart component

Figures 10.18, 10.19 and 10.20 specify the join/depart component of Gnu service program in SeSF. The rest of this section explores informally parts of the specifications. The service defines the following variables for each node i (i, j $\in$ Node $-$ $\{\mathsf{null}\}$ for the following definitions):

- $\mathsf{state[i]} = \{\mathsf{Inactive}, \mathsf{Active}, \mathsf{Departing}\}$. Initially Inactive.

  This variable indicates the state of node i with respect to the peer-to-peer network.

  Inactive means i is not connected to the peer-to-peer network and is not attempting to connect to it. Active means i can issue connection requests to, or receive connection requests from other nodes. Departing means i has requested to depart the network and is waiting for the connections (if any) to close.

- $\mathsf{outStatus[i,j]} = \{\mathsf{Closed}, \mathsf{Connecting}, \mathsf{Connected}, \mathsf{Closing}\}$. Initially Closed.

  This variable reflects the status of node i's outgoing connection to node j. An *outgoing* connection is a connection initiated by node i.

  Closed means i does not have an outgoing connection to node j. Connecting means i has requested a connection to node j. Connected means i has established a connection to node j. Closing means i has requested termination of the connection to node j.

- $\mathsf{inStatus[i, j]} = \{\mathsf{Closed, Connecting, Connected, Closing}\}$. Initially Closed.

  This variable reflects the status of node i's incoming connection from node j. An *incoming* connection is a connection initiated by remote node j.

  Closed means i does not have an incoming connection from node j. Connecting means i has indicated a connection establishment attempt by node j. Connected means i has established a connection to node j (node j initiated the attempt). Closing means i has requested termination of the connection to node j.

- $\mathsf{JR[i, j]} = \{\mathsf{true, false}\}$. Initialized to false.

  This variable is used to prevent node j from accepting a nonexistent connection request from node i. It is true iff node i has requested to join node j (issuing $\mathsf{joinReq(i, j)}$ as explained later), node j has not received the request yet, and node j has not yet departed the network.

- $\mathsf{JRR[i, j]} = \{\mathsf{true, false}\}$. Initialized to false.

  This variable indicates that node i has received a connection request from node j. It is used to prevent node j from acknowledging a connection request that hasn't reached node i. It is true iff node i has received a join request from node j, node j hasn't acknowledged or rejected this join reply yet, and node j has not yet departed the network.

- $\mathsf{JRRAck[i, j]} = \{\mathsf{true, false}\}$. Initialized to false.

  This variable serves to prevent node j from accepting a nonexistent connection request ack from node i. It is true iff node i has sent the last ack in the handshake sequence to node j, j hasn't received it yet, and node j has not departed the network

upw/dnw	Event	Indication
dnw	activate(i)	node i becomes active, and can receive incoming connection requests from any node in the network
dnw	joinReq(i,j)	i requests to connect to j
upw	joinRej(i,j)	i learns that its connection request to j is rejected
upw	joinInd(i,j)	i learns that its connection request to j is accepted; i becomes connected to j
upw	joinReqInd(i,j)	i receives a connection request from j
upw	joinReqRej(i,j)	i learns that j's connection request to i has been rejected.
upw	joinReqAck(i,j)	i learns that j's connection request to i has succeeded and i becomes connected to j
upw	departInd(i,j)	i learns that its connection with j is terminated
dnw	departReq(i)	i requests to depart the network
upw	departAck(i)	i's request to depart the network is fulfilled
dnw	abort(i)	i terminates abruptly

Table 10.1: Events of join/depart component of service Gnu. The first parameter indicates the node where the event occurs.

yet.

We need both outStatus[i, j] and inStatus[i, j], because if both node i and node j issue join requests to each other simultaneously, two separate connections will eventually be established (if they can reach each other and neither departs).

Table 10.1 lists the events used in the join/depart component.

Figures 10.5 through 10.10 show various scenarios for connection establishment and termination.

Figure 10.5 shows a typical successful connection establishment and termination between two nodes i and j. Here, app i and app j are applications, srv i and srv j are the corresponding servents, and nodes i and j are active.

- App i calls joinReq(i, j) of srv i.

- Srv i establishes a TCP connection with srv j, and then sends “Connect Request” string to srv j, followed by its capability headers.

- Upon establishing the TCP connection, srv j informs app j that srv i is requesting a connection. Afterwards, srv j receives the capability headers. Upon finding them acceptable, srv j sends “OK” string followed by its own capability headers.

- Upon receiving the “OK” string, and accepting the capability headers, srv i calls $\mathsf{joinInd}(\mathsf{i},\mathsf{j})$ to inform app i of the success of the connection establishment, and then sends “OK” string to srv j.

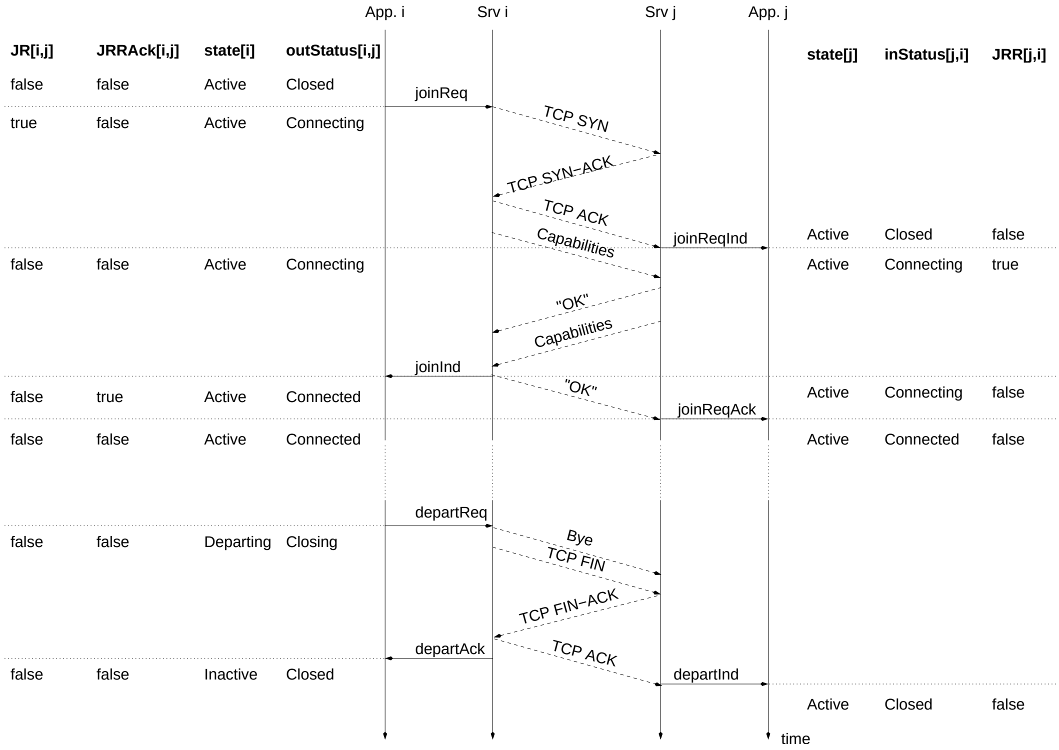
- Srv j receives the “OK” string and calls $\mathsf{joinReqAck}(\mathsf{j},\mathsf{i})$ to inform app j of the connection establishment with srv i.

- Later, app i decides to depart the network. It issues $\mathsf{departReq}(\mathsf{i})$. Consequently, srv i sends “Bye” string to all its connections (including j’s), and then terminates all the outgoing paths of its connections.

- Upon receiving “Bye”, srv j terminates its connection to srv i, and informs app j of the departure of node i.

- After a certain timeout, srv i aborts all active connections (if any) and informs app i of the termination.

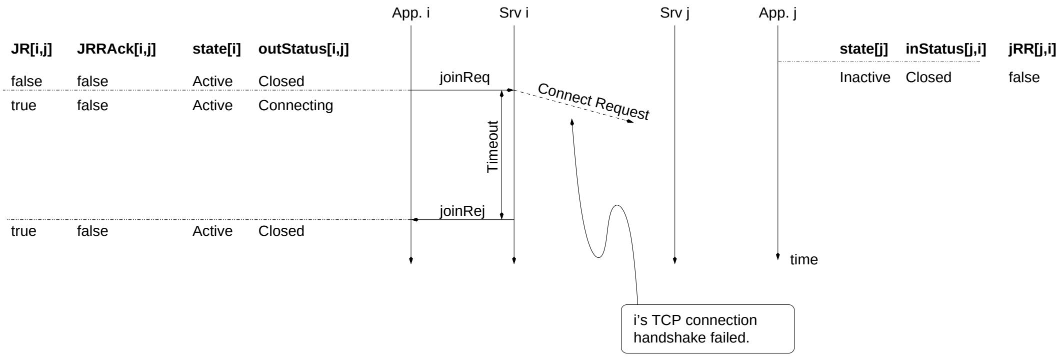
Figure 10.6 shows a variation of the scenario mentioned in Figure 10.5, where TCP handshake sequence fails, because, for example, node j is not reachable or node j is reachable but not listening. Suppose that one of these conditions occurs, then srv i waits for a specified timeout, and then informs app i of the failure of the connection request.



JR[i,j]	JRRAck[i,j]	state[i]	outStatus[i,j]
false	false	Active	Closed
true	false	Active	Connecting
false	false	Active	Connecting
false	true	Active	Connected
false	false	Active	Connected
false	false	Departing	Closing
false	false	Inactive	Closed

state[j]	inStatus[j,i]	JRR[j,i]
Active	Closed	false
Active	Connecting	true
Active	Connecting	false
Active	Connected	false
Active	Closed	false

Figure 10.5: Gnu join/depart scenario 1



JR[i,j]	JRRAck[i,j]	state[i]	outStatus[i,j]
false	false	Active	Closed
true	false	Active	Connecting
true	false	Active	Closed

state[j]	inStatus[j,i]	jRR[j,i]
Inactive	Closed	false

Figure 10.6: Gnu join/depart scenario 2

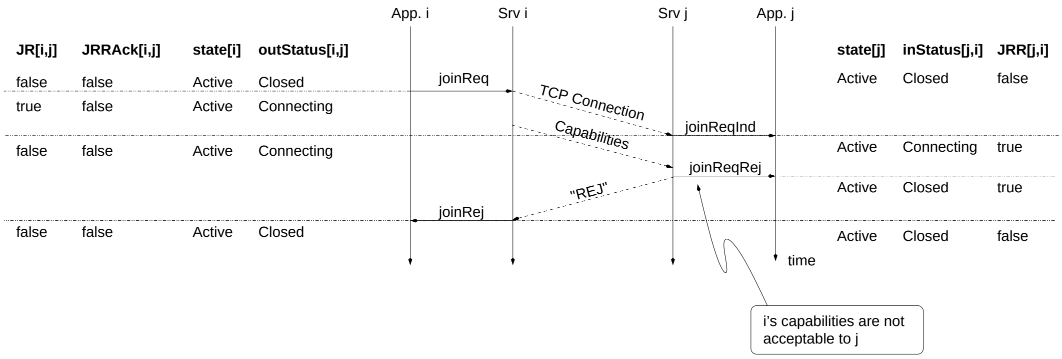

App. i
Srv i
Srv j
App. j
JR[i,j] JRRAck[i,j] state[i] outStatus[i,j]
false false Active Closed
true false Active Connecting
false false Active Connecting
false false Active Closed
joinReq
TCP Connection
Capabilities
joinReqInd
joinReqRej
"REJ"
joinRej
state[j] inStatus[j,i] JRR[j,i]
Active Closed false
Active Connecting true
Active Closed true
Active Closed false
time
i's capabilities are not acceptable to j

Figure 10.7: Gnu join/depart scenario 3

Figure 10.7 shows a variation of the scenario mentioned in Figure 10.5. After informing app j of the existence of an attempt, j checks the capability headers and finds them unacceptable. Then, it informs app j of the rejection of the connection, and sends "REJ" string to node i, which in turn informs app i of the failure of the connection establishment.

Figure 10.8 shows a variation of the scenario mentioned in Figure 10.7. Suppose that srv j accepts srv i's capability headers, and consequently, it sends "OK" string to srv i followed by its own capability headers. After receiving the "OK" string, srv i finds srv j's capability headers unacceptable. Thus, it informs app i of the failure of the connection establishment, and sends "REJ" string to srv j, which in turn informs app j of the rejection of the connection establishment.

Figure 10.9 shows a scenario when app i requests connection to app j, but *aborts* (closes abruptly) before connection is established. Srv j waits for an arbitrary timeout before informing app j of the failure of the connection attempt. In figure 10.10, app i recovers, activates srv i and requests the establishment of the connection before j's timeout fires. In this case, srv j informs app j of the existence of a new connection request.

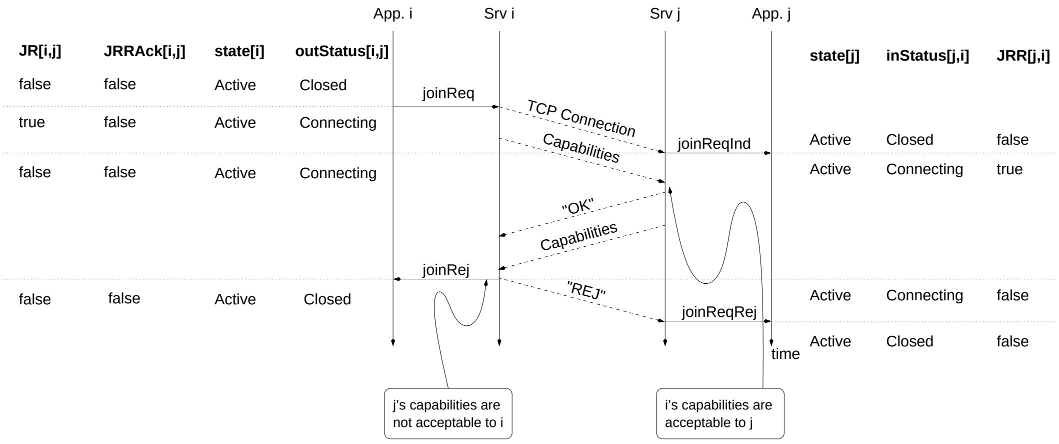


Figure 10.8: Gnu join/depart scenario 4

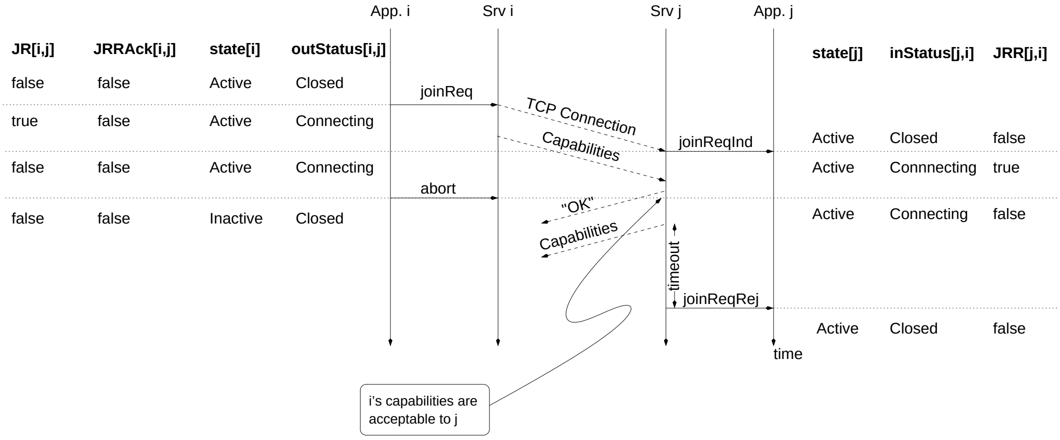


Figure 10.9: Gnu join/depart scenario 5

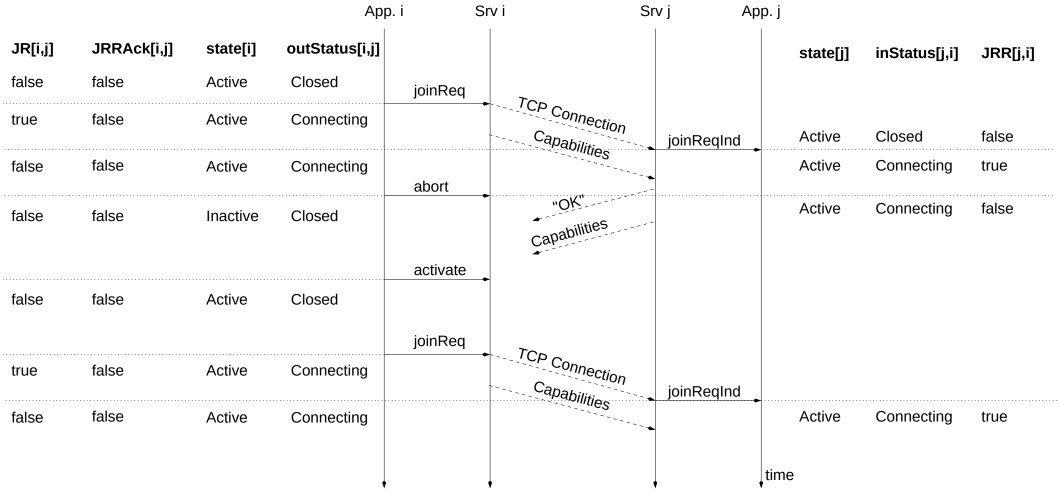


Figure 10.10: Gnu join/depart scenario 6

The join/depart component of service Gnu defines the following progress obligations for every pair i and j:

P1 If i's outStatus is Connecting and j's inStatus is Connecting, then eventually (1) i's outStatus is Connected, (2) both i's outStatus and j's inStatus are Closed, or (3) one or both nodes depart.

P2 If i's outStatus is Connected and j's inStatus is Connecting, then eventually (1) i's outStatus and j's inStatus are Connected, or (2) one or both nodes depart.

P3 If $\mathsf{joinReq}(\mathsf{i},\mathsf{j})$ occurs, then either $\mathsf{joinInd}(\mathsf{i},\mathsf{j})$ or $\mathsf{joinRej}(\mathsf{i},\mathsf{j})$ will eventually be executed, or node i departs the network. Node i's outStatus cannot stay in state Connecting forever.

P4 If $\mathsf{joinReqInd}(\mathsf{i},\mathsf{j})$ is occurred, then either $\mathsf{joinReqAck}(\mathsf{i},\mathsf{j})$ or $\mathsf{joinReqRej}(\mathsf{i},\mathsf{j})$ is eventually executed, or i departs the network. Node i's inStatus cannot stay in state Connecting forever.

P5 If $\mathsf{departReq}(\mathsf{i})$ occurs, then either $\mathsf{departAck}(\mathsf{i})$ or $\mathsf{abort}(\mathsf{i})$ is eventually executed. Node i cannot stay in state Departing forever.

P6 If i is inactive, then all connections to i are eventually closed.

10.2.2 Query component

The query component of the Gnu service, given in figure 10.21, is extremely simple. It sends a query string, and waits for a reply set (empty set means no hits). This component at a node i is active only if srv i's state (defined in the join/depart component) is Active. Gnu service (Query component) defines the following variable:

- $\mathsf{querying[i]} = \{\mathsf{true}, \mathsf{false}\}$. Initialized to false. We assume that i $\in$ Node $-$ $\{\mathsf{null}\}$. Variable querying[i] is true if node i has requested a query, but haven't received a reply yet.

10.3 Internal service Gnu_TCP

In this section, we define service Gnu_TCP. Recall that the internal service is just the special case of the TCP service as it is used by Gnutella. Section 10.3.1 describes the join/depart component, while section 10.3.2 describes the query component. Again, for simplicity, we ignore node discovery and upload/download services. Since service Gnu_TCP is internal, it can impose progress obligations on dnw events.

10.3.1 Join/depart component

Figures 10.22 through 10.26 illustrate the join/depart component of Gnu_TCP service program. The service defines the following variables for each node i (we assume that i, j $\in$ Node $-$ $\{\mathsf{null}\}$ for the following definitions):

- $\mathsf{outStatus[i,j]} = \{\mathsf{Closed}, \mathsf{Connecting}, \mathsf{Handshaking}, \mathsf{Connected}, \mathsf{Closing}\}$. Initially Closed.

  This variable reflects the status of node i's outgoing connection to node j.

  Closed means that node i does not have an outgoing connection to node j. Connecting means that node i has requested a connection to node j, and the TCP handshake is underway. $\mathsf{Handshaking}$ means that node i has established a TCP connection to node j and has initiated the Gnutella handshake sequence. $\mathsf{Connected}$ means that i has established a Gnutella connection to node j (Gnutella handshake is successful). $\mathsf{Closing}$ means that node i has requested termination of the connection to node j.

- $\mathsf{inStatus}[\mathsf{i},\mathsf{j}] = \{\mathsf{Closed}, \mathsf{Waiting}, \mathsf{Handshaking}, \mathsf{Connected}, \mathsf{Closing}\}$. Initially Closed. This variable reflects the status of node i's incoming connection from node j.

  Closed means that node i does not have an incoming connection from node j. $\mathsf{Waiting}$ means node i has established a TCP connection to node j (node j initiated the attempt), and is waiting for j to start the Gnutella handshake sequence. $\mathsf{Handshaking}$ means i has received j's capabilities (first leg of the handshake) and has accepted these capabilities. $\mathsf{Connected}$ means i has established a Gnutella connection to node j (node j initiated the attempt). $\mathsf{Closing}$ means i has requested termination of the connection to node j.

- $\mathsf{JR}[\mathsf{i},\mathsf{j}] = \{\mathsf{true, false}\}$. Initialized to false.

  Similar to Gnu, this variable is used to prevent node j from accepting a nonexistent connection request from node i.

- $\mathsf{handshake}[\mathsf{i},\mathsf{j}] = \{\mathsf{None}, \mathsf{cCpbs}, \mathsf{sCpbs}, \mathsf{Ack}, \mathsf{Bye}\}$. Initialized to None.

  This variable reflects the status of the Gnutella handshake sequence between node i and node j, where node i initiated the sequence. Specifically, $\mathsf{handshake}[\mathsf{i},\mathsf{j}]$ indicates the last message in the last handshake sequence.

None means that node i hasn't initiated a handshake sequence or the sequence is terminated either by an acceptance or a rejection. cCpbs (Client Capabilities) means that node i has sent its capabilities to node j. sCpbs (Server Capabilities) means that node j has accepted i's capabilities and has sent its own capabilities. Ack means that node i has accepted j's capabilities, and has sent the ack. Bye means that node i has sent message Bye.

- $\mathsf{cRP}[\mathsf{i},\mathsf{j}] = \{\mathsf{None}, \mathsf{Cpbs}, \mathsf{sCpbsReply}, \mathsf{Close}\}$. Initially None.

  This variable indicates which reply is pending during the communication between client i and server j.

  None means that there is no pending reply. Cpbs means i has established a TCP connection with j, and has to send its capabilities. sCpbsReply means i has received j's server capabilities, and has to send its reply (either acceptance or rejection of these capabilities). Close means i has sent a Bye message, and has to close its TCP connection with j.

- $\mathsf{sRP}[\mathsf{i},\mathsf{j}] = \{\mathsf{None}, \mathsf{cCpbsReply}, \mathsf{Close}\}$. Initially None.

  This variable indicates which reply is pending during the communication between server i and client j.

  None means that nothing is pending. cCpbsReply means i has received j's client capabilities, and has to send its reply (either acceptance or rejection of these capabilities). Close means i has received the Bye message, and has to close its TCP connection with j.

Table 10.2 illustrates the events used in the join/depart component.

upw/dnw	Event	Indication
dnw	joinReq(i,j)	i requests to connect to j
upw	peerReached(i,j)	i learns that it has established a TCP connection with j
dnw	cTx(i,j,s)	client i sends message s to j
upw	cRx(i,j,s)	client i receives message s from j
upw	joinAborted(i,j)	i learns that its connection request to j is aborted
upw	joinReqInd(i,j)	i learns that it has established a TCP connection with j (initiated by j)
dnw	sTx(i,j,s)	server i sends message s to j
upw	sRx(i,j,s)	server i receives message s from j
upw	joinReqAbort(i,j)	i learns of the abortion to j's connection request
dnw	cDepartReq(i,j)	client i requests to close its TCP connection with j
upw	cDepartAck(i,j)	client i's TCP connection with j is closed
dnw	sDepartReq(i,j)	server i requests to close its TCP connection with j
upw	sDepartAck(i,j)	server i's TCP connection with j is closed
dnw	abort(i,j)	i terminates its connection to j abruptly

Table 10.2: Events of join/depart component of service Gnu_TCP. The first parameter indicates the servent where the event occurs.

Figures 10.11 through 10.16 show various scenarios for connection establishment.

Figure 10.11 shows a typical successful connection establishment and termination between two nodes i and j. Here, srv i and srv j are servents, tcp i and tcp j are the corresponding network systems, and nodes i and j are active.

- Srv i calls $\mathsf{joinReq}(\mathsf{i}, \mathsf{j})$ of tcp i, which starts establishing a TCP connection with tcp j.

- Upon establishing the TCP connection, tcp i informs srv i of its successful TCP connection establishment by calling $\mathsf{peerReached}(\mathsf{i}, \mathsf{j})$, and tcp j informs srv j that a TCP connection with tcp i is established by calling $\mathsf{joinReqInd}(\mathsf{j}, \mathsf{i})$. Then, Srv i passes "Connect Request" string followed by its capability headers to tcp i via $\mathsf{cTx}(\mathsf{i}, \mathsf{j}, \mathsf{cpbs}[\mathsf{i}])$.

- Upon receiving the "Connect Request" string and finding the capability headers acceptable, srv j passes an "OK" string (indicating its acceptance of the capability headers) followed by its own capability headers to tcp j via $\mathsf{sTx}(\mathsf{j}, \mathsf{i}, \mathsf{cpbs}[\mathsf{j}])$, which sends then to tcp i.

- Upon receiving the "OK" string and the capability headers, and finding them acceptable, srv i sends "OK" string to tcp j as an indication of the success of the Gnutella connection establishment.

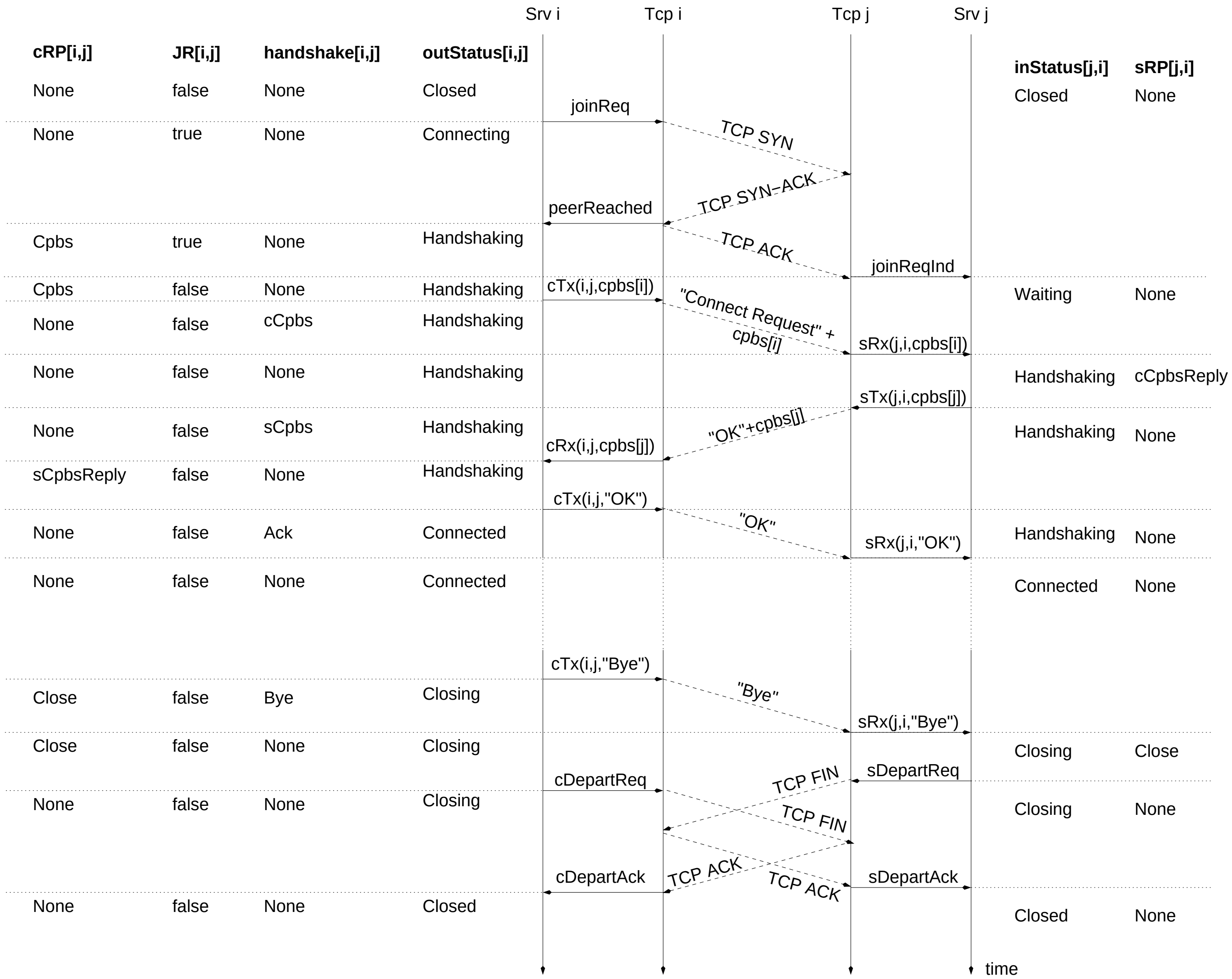
- Tcp j receives the "OK" string and passes it srv j to indicate the connection establishment with tcp i.

- Later, srv i decides to close the connection. It sends Bye, and then terminates the TCP connection with tcp j.

- Upon receiving "Bye", srv j terminates its connection to tcp i.

Figure 10.12 shows a variation of the scenario mentioned in Figure 10.11, where TCP handshake sequence fails, because, for example, node j is not reachable or node j is reachable but not listening. Suppose that one of these conditions occurs, then tcp i waits for a specified timeout, and then informs srv i of the failure of the connection request.

Figure 10.13 shows a variation of the scenario mentioned in Figure 10.11. After being informed of the existence of an attempt, srv j checks the capability headers and finds them unacceptable. Then, it informs tcp j of the rejection of the connection by sending "REJ" string to tcp i. Tcp i informs srv i of the failure of the connection establishment.



cRP[i,j]	**JR[i,j]**	**handshake[i,j]**	**outStatus[i,j]**
None	false	None	Closed
None	true	None	Connecting
Cpbs	true	None	Handshaking
Cpbs	false	None	Handshaking
None	false	cCpbs	Handshaking
None	false	None	Handshaking
None	false	sCpbs	Handshaking
sCpbsReply	false	None	Handshaking
None	false	Ack	Connected
None	false	None	Connected
Close	false	Bye	Closing
Close	false	None	Closing
None	false	None	Closing
None	false	None	Closed

inStatus[j,i]	**sRP[j,i]**
Closed	None
Waiting	None
Handshaking	cCpbsReply
Handshaking	None
Handshaking	None
Connected	None
Closing	Close
Closing	None
Closed	None

Figure 10.11: Gnu_TCP join/depart scenario 1

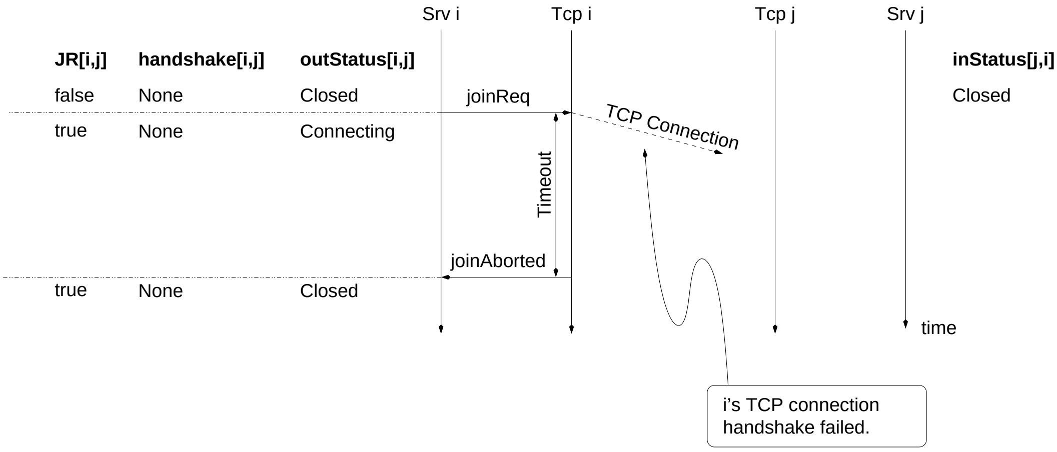


Figure 10.12: Gnu_TCP join/depart scenario 2

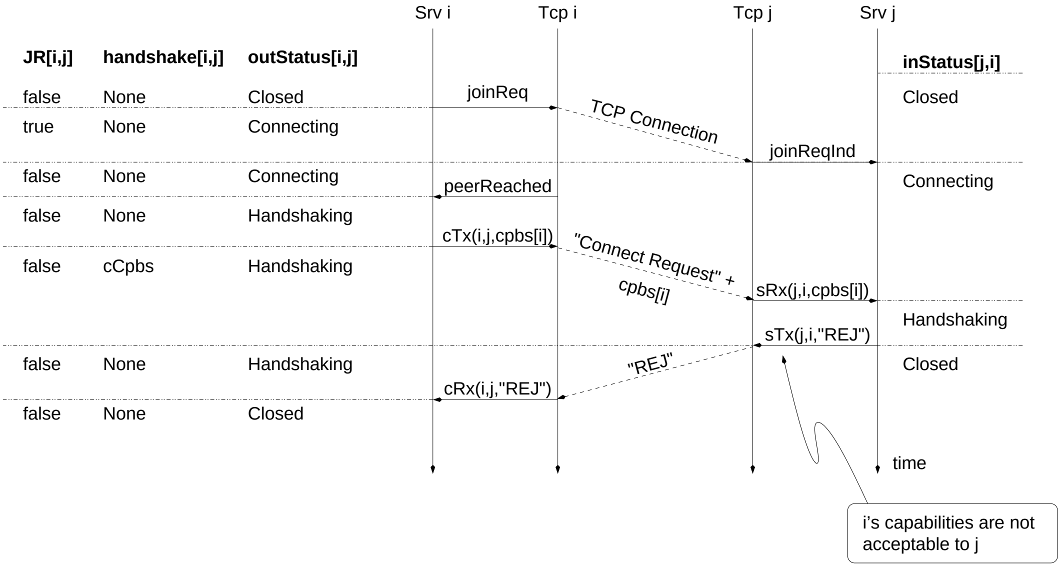


Figure 10.13: Gnu_TCP join/depart scenario 3

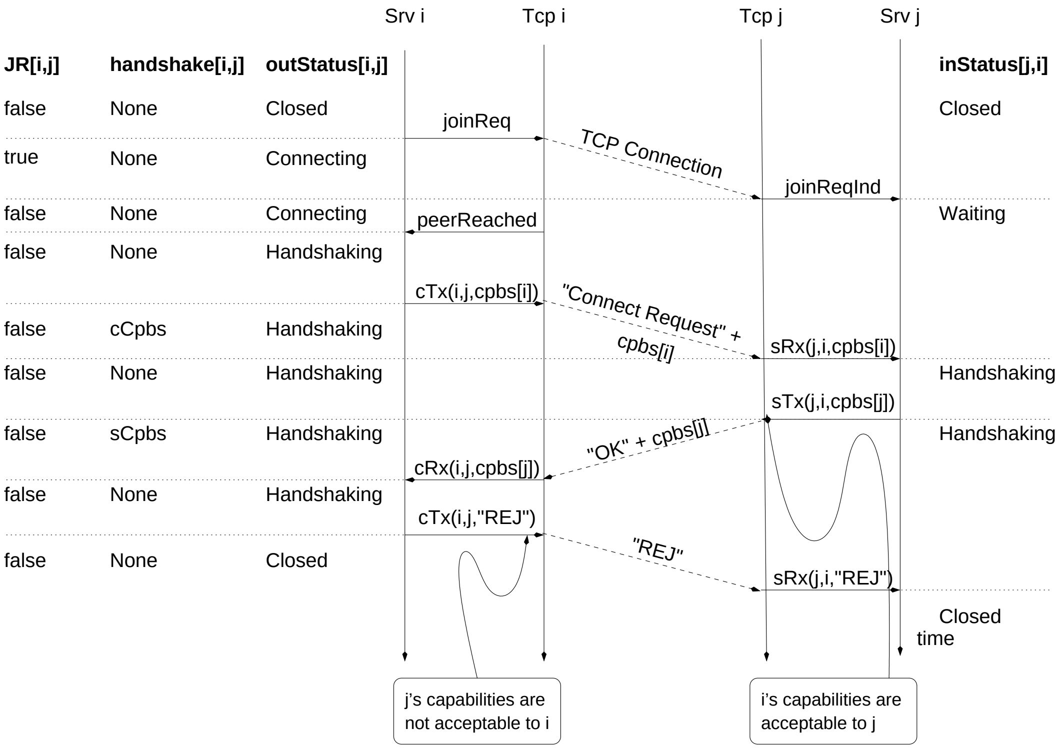
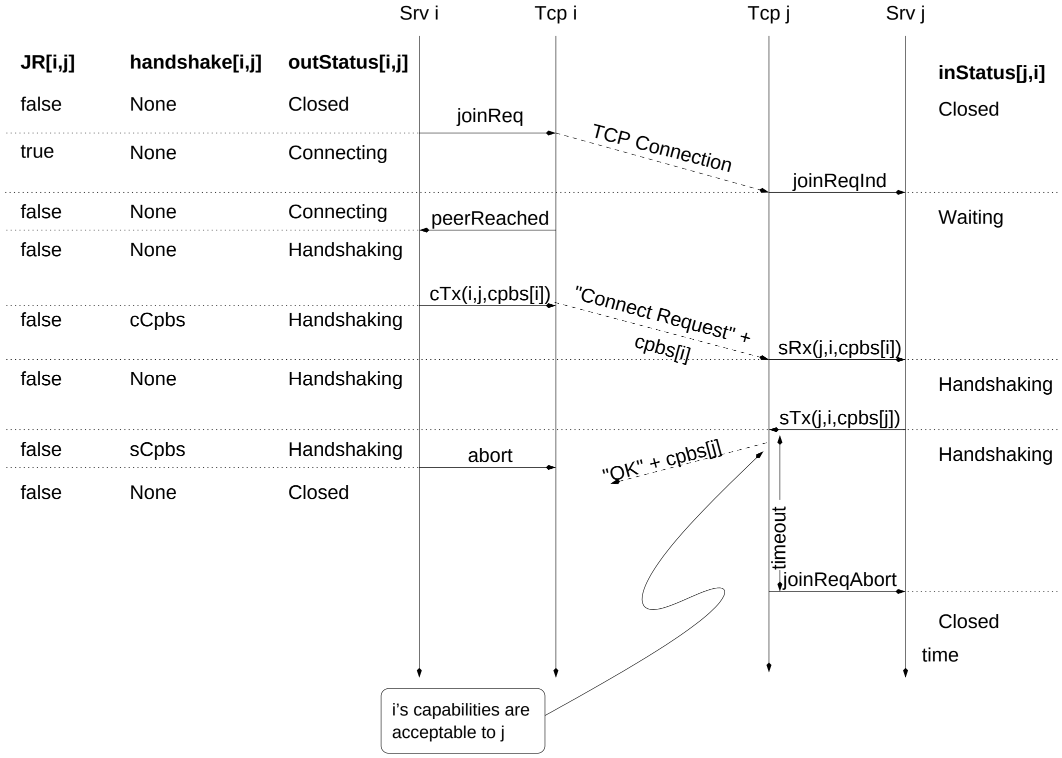


Figure 10.14: Gnu_TCP join/depart scenario 4

Figure 10.14 shows a variation of the scenario mentioned in Figure 10.13. Suppose that srv j accepts i's capability headers, and consequently, it sends "OK" string to tcp i followed by its own capability headers. Tcp i passes the capability headers to srv i, which finds j's capability headers unacceptable. Thus, srv i informs tcp i to send "REJ" string to tcp j as an indication of the rejection of the connection establishment. Finally, tcp i informs srv j of this rejection.

Figure 10.15 shows a scenario when srv i requests connection to node j, but *aborts* (closes abruptly) before the Gnutella connection is established. Tcp j waits for an arbitrary timeout before informing srv j of the failure of the connection attempt. In figure 10.16, node i recovers, activates tcp i and requests the establishment of the connection before j's timeout fires. In this case, tcp j informs srv j of the existence of a new



JR[i,j]	handshake[i,j]	outStatus[i,j]
false	None	Closed
true	None	Connecting
false	None	Connecting
false	None	Handshaking
false	cCpbs	Handshaking
false	None	Handshaking
false	sCpbs	Handshaking
false	None	Closed

Figure 10.15: Gnu_TCP join/depart scenario 5

connection request.

The join/depart component of service Gnu_TCP define the following progress obligations for every pair i and j:

P1 If i has requested a connection establishment, then eventually (1) j is state Waiting (wrt i) or (2) one or both nodes are Closed.

P2 If i is Connecting and j is Waiting, then eventually (1) i is Handshaking, or (2) one or both i and j are Closed.

P3 If i is Handshaking and j is Waiting, then eventually (1) i and j are Handshaking, or (2) one or both nodes are Closed.

P4 If i is Handshaking and j is Handshaking, then eventually (1) i is Connected, or (2)

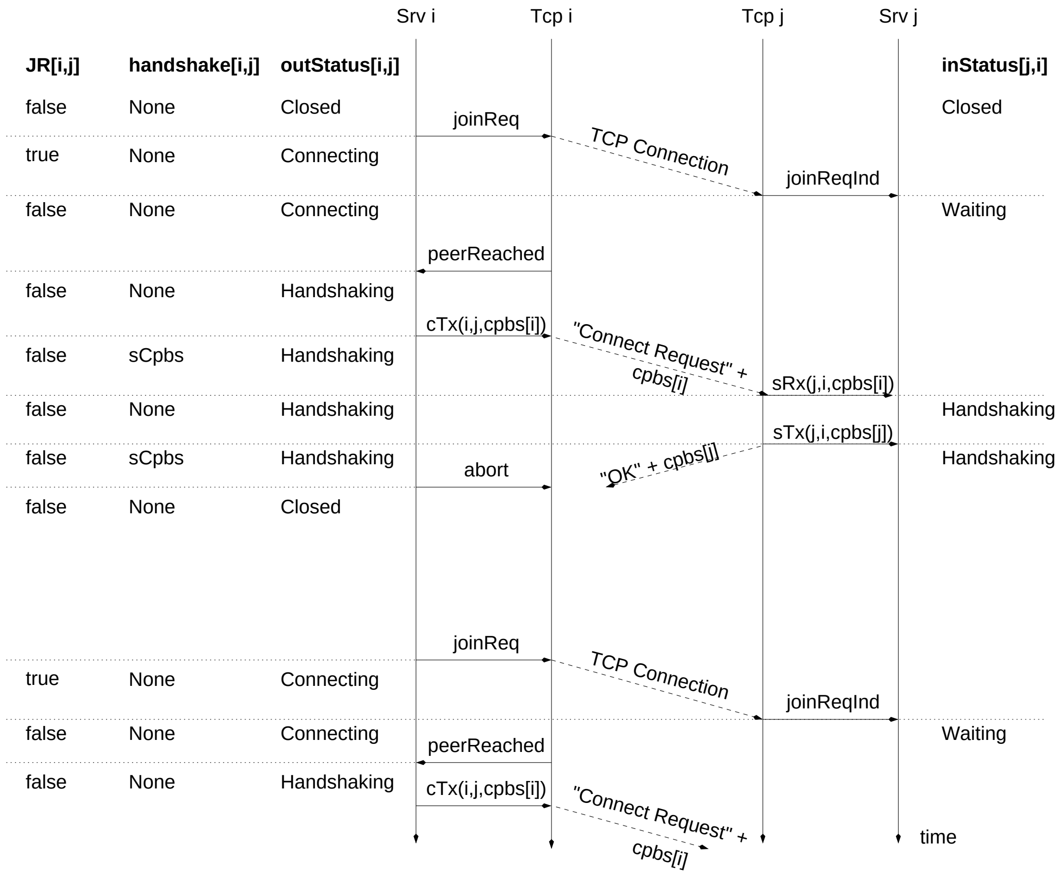


Figure 10.16: Gnu_TCP join/depart scenario 6

one or both nodes are Closed.

P5 If i is Connected and j is Handshaking, then eventually (1) i and j are Connected, or (2) one or both nodes are Closed.

P6 If joinReq(i, j) occurs, then either peerReached(i, j), joinAborted(i, j) or abort(i, j) will eventually be executed. Node i cannot stay in state Connecting forever.

P7 If joinReqInd(i, j) occurs, then cRx(i, j, cpbs[i]), joinReqAbort(i, j) or abort(i, j) is eventually executed. Node i cannot stay in state Waiting forever.

P8 Client node i cannot stay in state Handshaking forever.

P9 Server node i cannot stay in state Handshaking forever.

P10 If cDepartReq(i, j) occurs, then either cDepartAck(i, j) or abort(i, j) is eventually executed.

P11 If sDepartReq(i, j) occurs, then either sDepartAck(i, j) or abort(i, j) is eventually executed.

P12 If a client node has a reply pending, then it eventually replies or the connection is aborted.

P13 If a server node has a reply pending, then it eventually replies or the connection is aborted.

10.3.2 Query component

A query has a unique 16-byte descriptor id, and a criteria string. Gnutella defines no standard format or matching semantics for the criteria string; its interpretation is completely determined by each node that receives it [32]. This component at a node i is active only if srv i's outStatus[i, j] (defined in the join/depart component) is Connected.

```
class GUID = 16-byte ID;
class Query {
  GUID  id;
  String criteria;
}
```

The Query component of Gnu_TCP service (figure 10.27) defines the following variables for each node $i \in$ Node $- \{\mathsf{null}\}$:

- Set(GUID) Q[i, j]. Initialized to {}. Set of queries transmitted from node i to node j.

- Set(GUID) QRcvd[i, j]. Initialized to {}. Queries received by i from j.

- Set(GUID) QHasHit[i, j]. Initialized to {}. Set of query requests received by i where i has files/data that satisfy the queries in this set.

- Set(GUID) H[i]. Initialized to {}. Set of query requests transmitted to j by node i.

- Set(GUID) HRcvd[i]. Initialized to {}. Set of query requests received by i from j.

Table 10.3 illustrates the events used in the query component. Figure 10.17 describes a scenario for the flow of a query request. Srv i sends a query q to the network through node k via TxQuery(i, j). Tcp k receives it and asks srv k about hits via

upw/dnw	Event	Indication
dnw	TxQuery(i,j,q)	i issues query q
upw	RxQuery(i,j, q)	i receives query q; it returns true if it has hits, false otherwise
dnw	TxHit(i, j, q, hits)	i sends or forwards hits for query q
upw	RxHit(i, j, q, hits)	i receives hits from j for query q

Table 10.3: Events of query component of service Gnu_TCP. The first parameter indicates the servent where the event occurs.

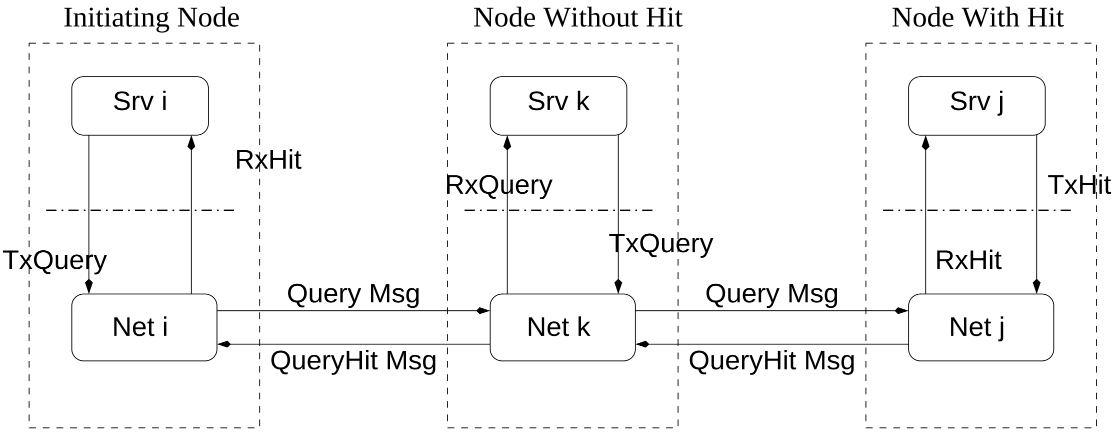


Figure 10.17: Gnu_TCP query scenario

RxQuery(k, i, q). When tcp k knows that srv k has no hits, it forwards q to its neighbors, specifically j. Tcp j receives it and asks srv j about hits. It discovers that srv j has some hits, so it does not forward q to any other node. Later, srv j sends the hits via TxHit(j, k, q.id, hits), and the hits message follows the same route back to tcp i and then to srv i via RxHit (events of receiving and forwarding query hits at node k are omitted in figure 10.17, for figure simplicity).

The query component of service Gnu_TCP define the following progress obligations:

SP1 If i receives a query q and i has an answer, it eventually answers or exits.

10.4 Testing and assertion checking of Furi

There are many open-source Java implementations for Gnutella, for example, Furi [87], Limewire [37], Phex [39] and JTella [53]

We applied the SeSFJava Harness to Furi. Furi is a medium-sized (33,000 lines of code) Java implementation of Gnutella, which was developed two years ago. We chose Furi because of its good documentation and readability. It does not support protocols other than Gnutella, as opposed, for example, to the more popular Limewire which supports Gnutella and other protocols. Furi's program structure is the closest to the Gnutella stack (figure 10.1); that is, there is a set of Java classes that corresponds to Gnutella management system, and an another set that corresponds to the P2P application. During the execution of Furi, we encountered fewer GUI errors compared to other systems (except Limewire).

Unfortunately, applying the Harness to Furi was not straightforward due to the following problems:

- The names of the xc events in Furi differ from the corresponding events in the peer-to-peer services. We developed wrapper classes to overcome this problem.
- Some of the xc events in Furi are blockable events. The traditional way to overcome such problem is to replace each blockable xc event <e> by two events: an xc event that corresponds to the initiation (or call) of <e>, and another lc event that corresponds to the return of <e>. Unfortunately we cannot do that because this involves modifying the implementation (which we try to avoid). Instead, we proceed as follows; assume that the event is a blockable call at the upper level.

We create two events in the upper service: one is the dnw event that corresponds to the call of xc event, and another upw event that corresponds to the return of the xc event. Then, we use the Harness breakpoint tags to ensure the correctness of operation. For example, Furi has methods connectToRemoteHost in file ReadWorker.java. This method corresponds to P2P_Net.joinReq(i, j). It blocks till the TCP connection is established. To counter this problem, we insert a service call (p2pNet.joinReq) before the call to connectToRemoteHost, and another service call (p2pNet.peerReached) after the call. We insert breakpoints to instruct the tester of how to execute the operation.

```
// p2pNet is an alias to service P2P_Net
//# breakpoint(”ReadWorker.start”, VIEW_AND_AUTOMATIC);
// ServiceManager.hostPortName gets the address of the sender node
// mRemoteHost.getHostAddr() gets the address of the remote node
//# p2pNet.joinReq(ServiceManager.hostPortName,
//#             mRemoteHost.getHostAddr()); //Service call
//# breakpoint(”ReadWorker.start”, WAIT);
try
{
  connectToRemoteHost();
  if (mRemoteHost.getStatus() == Host.sStatusTimeout)
  {
    // Connecting has been taken too long.
    throw new Exception(”Timed out.”);
  }
}
catch (Exception e3)
{
   mHostMgr.setHostCaughtConnectionFailed(mRemoteHost.getHostAddr(), true);
   throw e3;
}
//# breakpoint(”ReadWorker.peerReached”, VIEW_AND_AUTOMATIC);
//# p2pNet.peerReached(ServiceManager.hostPortName,
//#                mRemoteHost.getHostAddr()); //Service call
//# breakpoint(”ReadWorker.tcpEstablished”, VIEW_AND_AUTOMATIC);
```

- Some methods include calls to multiple events, where each event is atomic. We handle this by inserting manually calls to the corresponding service events. Breakpoints are inserted to slice these methods in sequence of atomic lc events.

In order to test the implementation, we ran three copies of Furi. Each is on a different machine on the *junkfood* and *UMIACS* clusters of University of Maryland. The three copies interact with each other, connecting and disconnecting continuously. We found the following errors:

- We found many synchronization errors in the connection establishment. Furi allows multiple connections to the same node if two consecutive joinReq calls are made to the same node.

- Furi does not treat the domain name and the IP address of a node as the same node. For example, newton.cs.umd.edu has an IP address of 128.8.129.9. A Furi copy at machine x can connect to newton.cs.umd.edu:1234, and then connect again to 128.8.129.9:1234 without realizing that they are the same.

- The program works fine if all the nodes have the default port 1234. If some of the hosts have different ports, this may result in errors because in some situations, the Furi does not augment the domain name with the port name.

```
service Gnu {
  for the following definitions: i and j ∈ Node − {null}.
  state[i] = {Active, Departing, Inactive}. Initialized to Inactive.
  outStatus[i,j] = {Closed, Connecting, Connected, Closing}. Initialized to Closed.
  inStatus[i,j] = {Closed, Connecting, Connected, Closing}. Initialized to Closed.
  boolean JR[i,j]. Initialized to false.
  boolean JRR[i,j]. Initialized to false.
  boolean acking[i,j]. Initialized to false.

   dnw activate (Node i) {
      ec: state[i] = Inactive;
      ac: state[i]     := Active;
          JR[*,i]   := false;
          JRR[*,i] := false;
          JRRAck[*,i]    := false;
  }

  // App i requests a connection to j.
   dnw joinReq (Node i, Node j) {
      ec: i ≠ j ∧ state[i] = Active ∧ outStatus[i,j]= Closed;
      ac: outStatus[i,j] := Connecting;
          JR[i,j]   := true;
  }

  // Connection request joinReq(i,j) has been accepted, and informs app i of the acceptance.
   upw joinInd (Node i, Node j) {
      ec: i ≠ j ∧ JRR[j,i] ∧ outStatus[i,j] = Connecting;
      ac: outStatus[i,j] := Connected;
          JRR[j,i] := false;
          JRRAck[i,j]    := true;
  }

  // Connection request joinReq(i,j) has been rejected, and srv i informs app i of the rejection.
   upw joinRej (Node i, Node j) {
      ec: i ≠ j ∧ outStatus[i,j] = Connecting;
      ac: outStatus[i,j] := Closed;
          JRR[j,i] := false;
  }

  //  Srv j has requested a connection to srv i.
   upw joinReqInd (Node i, Node j) {
      ec: i ≠ j  ∧ state[i] = Active ∧ JR[j,i] ∧ inStatus[i,j] ∈ {Closed, Connecting, Connected};
      ac: inStatus[i,j] := Connecting;
          JR[j,i]   := false;
          JRR[i,j] := true;
  }

  // Srv i accepts the connection request initiated by node j, and informs app i of the acceptance.
   upw joinReqAck (Node i, Node j) {
      ec: i ≠ j  ∧ JRRAck[j,i] ∧ inStatus[i,j] = Connecting;
      ac: inStatus[i,j] := Connected;
          JRRAck[j,i] := false;
  }

  // Srv i rejects the connection request initiated by srv j, and informs app i of the rejection.
   upw joinReqRej (Node i, Node j) {
      ec: i ≠ j ∧ inStatus[i,j] = Connecting;
      ac: inStatus[i,j]   := Closed;
          JRRAck[j,i] := false;
  }
```

Figure 10.18: Join/depart component of Gnu service in SeSF (Part 1)

```
// Srv i informs the app i that j has departed or is departing the network,
// and i has closed all its connections to j.
upw departInd (Node i, Node j) {
   ec: $i \neq j \wedge$ (outStatus[i,j] = Connected $\vee$ inStatus[i,j] = Connected);
   ac: outStatus[i,j] := Closed;
       inStatus[i,j]  := Closed;
       JR[j,i]   := false;
       JRR[j,i] := false;
       JRRAck[j,i]    := false;
       JR[i,j]   := false;
       JRR[i,j] := false;
       JRRAck[i,j]    := false;
}

// App i is requesting to depart the network.
dnw departReq (Node i) {
   ec: state[i] = Active;
   ac: state[i] := Departing;
       forall (j: $j \in$ Node - {null}) {
          if (outStatus[i,j] = Connected)
            outStatus[i,j] := Closing;
          else
            outStatus[i,j] := Closed;
          if (inStatus[i,j] = Connected)
            inStatus[i,j] := Closing;
          else
            inStatus[i,j] := Closed;
       }
}

// Node i has no Connecting, Connected, or closing connections.
upw departAck(Node i) {
   ec: state[i] = Departing;
   ac: state[i] := Inactive;
       outStatus[i,*] := Closed;
       inStatus[i,*]  := Closed;
       JR[*,i]   := false;
       JRR[*,i] := false;
       JRRAck[*,i]    := false;
}

// App i closes abruptly.
dnw abort (Node i) {
   ec: state[i] $\neq$ Inactive;
   ac: state[i] := Inactive;
       outStatus[i,*] := Closed;
       inStatus[i,*]  := Closed;
       JR[*,i]   := false;
       JRR[*,i] := false;
       JRRAck[*,i]    := false;
}
```

Figure 10.19: Join/depart component of Gnu service in SeSF (Part 2)

```
  for the following definitions, i ≠ j ≠ null ∧ i ≠ j

  // If i's outStatus is Connecting and j's inStatus is Connecting, then eventually (1) i's outStatus is Connected,
  // (2) both i's outStatus and j's inStatus are Closed, or (3) one or both nodes depart.
  progress-obligation P1 {
    (outStatus[i,j] = Connecting ∧ inStatus[j,i] = Connecting ) leadsto
      ((outStatus[i,j] = Connected ∧ inStatus[j,i] = Connecting) ∨ (outStatus[i,j] = Closed ∧ inStatus[j,i] = Closed) ∨
       state[i] ≠ Active ∨ state[j] ≠ Active);
  }

  // If i's outStatus is Connected and j's inStatus is Connecting,
  // then eventually (1) i's outStatus and j's inStatus are Connected, or (2) one or both nodes depart.
  progress-obligation P2 {
    (outStatus[i,j] = Connected ∧ inStatus[j,i] = Connecting ) leadsto
      ((outStatus[i,j] = Connected ∧ inStatus[j,i] = Connected) ∨ state[i] ≠ Active ∨ state[j] ≠ Active);
  }

  // If joinReq(i,j) occurs, then either joinInd(i,j) or joinRej(i,j) will eventually be executed,
  // or node i departs the network. Node i's outStatus cannot stay in state Connecting forever.
  progress-obligation P3 {
    (outStatus[i,j] = Connecting leadsto (outStatus[i,j] = Closed ∨ outStatus[i,j] = Connected ∨ state[i] ≠ Active)
  }

  // If joinReqInd(i,j) is occurred, then either joinReqAck(i,j) or joinReqRej(i,j) is eventually executed,
  // or i departs the network. Node i's inStatus cannot stay in state Connecting forever.
  progress-obligation P4 {
   (inStatus[i,j] = Connecting ∧ i,j ≠ null) leadsto (inStatus[i,j] = Closed ∨ inStatus[i,j] = Connected ∨ state[i] ≠ Active)
  }

  // If departReq(i) occurs, then either departAck(i) or abort(i) is eventually executed.
  // Node i cannot stay in state Departing forever.
  progress-obligation P5 {
    state[i] = Departing leadsto state[i] = Inactive
  }

  // If i is inactive, then all connections to i are eventually closed.
  progress-obligation P6 {
     state[i] = Inactive  leadsto
        (∀ j: (j, i) ∈ E ::(outStatus[j,i] = Closed ∧ inStatus[j,i] = Closed) ∨ state[j] ≠ Active)
  }
}
```

Figure 10.20: Join/depart component of Gnu service in SeSF (Part 3)

```
service Gnu {
  ...
  for the following definitions, i $\in$ Node $-$ {null}.
  boolean querying[i] = {true, false}. Initialized to false.

  // App i issues a query request.
   dnw queryReq (Node i, String query) {
     ec: state[i] = Active $\wedge$ !querying[i];
     ac: querying[i] := true;
  }

  // App i receives set of hits. The set may be empty.
   upw queryHitRcvd (Node i, Set(Hit) hits) {
     ec: state[i] = Active $\wedge$ querying[i];
     ac: querying[i] := false;
  }

  // If i send a query, it will receive an answer.
   progress-obligation SP1{
       querying[i] leadsto $\neg$querying $\vee$ state[i] = Inactive;
  }
}
```

Figure 10.21: Query component of Gnu service in SeSF

```
service Gnu_TCP {
 for all of the following definitions: i and j ∈ Node − {null}.
  outStatus[i,j]       = {Closed, Connecting, Handshaking, Connected, Closing}. Initialized to Closed.
  inStatus[i,j]        = {Closed, Waiting, Handshaking, Connected}. Initialized to Closed.
  boolean JR[i,j]      Initialized to false.
  handshake[i,j]       = {None, cCpbs, sCpbs, Ack}. Initialized to None.
  cRP[i,j]             = {None, Cpbs, sCpbsReply, Close}. Initialized to None.
  sRP[i,j]             = {None, cCpbsReply, Close}. Initialized to None.

 // Srv i requests a connection to j.
  dnw joinReq (Node i, Node j) {
     ec: i ≠ j ∧ outStatus[i,j]= Closed;
     ac: outStatus[i,j] := Connecting;
         JR[i,j]   := true;
  }

 // Srv i learns that it has established a TCP connection with j.
  upw peerReached (Node i, Node j) {
     ec: i ≠ j ∧ outStatus[i,j] = Connecting;
     ac: outStatus[i,j] := Handshaking;
         cRP[i,j] := Cpbs;
  }

  // Client servent i sends a message s to j.
   dnw cTx (Node i, Node j, String s) {
     ec: i ≠ j ∧ outStatus[i,j] ∈ {Handshaking, Connected};
     ac: if (outStatus[i,j] = Handshaking ∧ cRP[i,j] = Cpbs ∧ prefix(s) = ”Connect Request”) { //Sending Cpbs
            handshake[i,j] := cCpbs;
            cRP[i,j] := None;
         } else if (outStatus[i,j] = Handshaking ∧ cRP[i,j] = sCpbsReply ∧ s = ”OK”) { // Sending OK
            outStatus[i,j] := Connected;
            handshake[i,j] := Ack;
            cRP[i,j] := None;
         } else if (outStatus[i,j] = Handshaking ∧ cRP[i,j] = sCpbsReply ∧ s = ”REJ”) { // Sending REJ
            outStatus[i,j] := Closed;
            cRP[i,j] := None;
         } else if (outStatus[i,j] = Connected ∧ s = ”Bye”) {
            outStatus[i.j] := Closing;
            handshake[i,j] := Bye;
            cRP[i,j] := Close;
         }
  }

  // Client servent i receives a message s from j.
   upw cRx (Node i, Node j, String s) {
     ec: i ≠ j ∧ outStatus[i,j] = Handshaking;
     ac: if (handshake[i,j] = sCpbs ∧ prefix(s) = ”OK”) { // sCpbs Received
           cRP[i,j] := sCpbsReply;
           handshake[i,j] := None;
         } else if (prefix(s) = ”REJ”) { // REJ Received
            outStatus[i,j] := Closed;
         }
  }

 // Srv i learns that its connection request to j is rejected.
  upw joinAborted (Node i, Node j) {
     ec: i ≠ j ∧ outStatus[i,j] ∈ {Connecting, Handshaking};
     ac: outStatus[i,j] := Closed;
         if (handshake[i,j] = sCpbs)
           handshake[i,j] := None;
  }
```

Figure 10.22: Join/Depart component of Gnu_TCP service in SeSF (Part 1)

```
// Srv i receives a connection request from j.
upw joinReqInd (Node i, Node j) {
    ec: $i \neq j \wedge JR[j,i] \wedge inStatus[i,j] \in \{Closed, Waiting, Handshaking, Connected\}$;
    ac: inStatus[i,j] := Waiting;
        JR[j,i]   := false;
}

// Server servent i receives j's message
upw sRx (Node i, Node j, String s) {
    ec: $i \neq j \wedge inStatus[i,j] \in \{Waiting, Handshaking, Connected\}$;
    ac: if (inStatus[i,j] = Waiting $\wedge$ handshake[j,i] = cCpbs $\wedge$ prefix(s) = "Connect Request") { // cCaps Received
          inStatus[i,j] := Handshaking;
          handshake[j,i] := None;
          sRP[i,j] := cCpbsReply;
      } else if (inStatus[i,j] = Handshaking $\wedge$ handshake[j,i] = Ack $\wedge$ s = "OK") { // OK Received (final leg of handshake)
          status[i,j] := Connected;
          handshake[j,i] := None;
        } else if (inStatus[i,j] = Handshaking $\wedge$ s = "REJ") { // REJ Received
          inStatus[i,j] := Closed;
        } else if (inStatus[i,j] = Connected $\wedge$ handshake[j,i] = Bye $\wedge$ s = "Bye") {
          inStatus[i,j] := Closing;
          handshake[j,i] := None;
          sRP[i,j] := Close;
        }
}

// Server servent i sends a message s to j.
dnw sTx (Node i, Node j, String s) {
    ec: $i \neq j \wedge$ inStatus[i,j] = Handshaking $\wedge$ sRP[i,j] = cCpbsReply $\wedge$ handshake[j,i] = None;
    ac: sRP[i,j] := None;
        if (prefix(s) = "OK") { //Accepting j's capabilities
          handshake[j,i] := sCpbs;
        } else if (s = "REJ") { //Rejecting j's capabilities
          inStatus[i,j] := Closed;
        }
}

// Srv i learns of the rejection to j's connection request.
upw joinReqAbort (Node i, Node j) {
    ec: $i \neq j \wedge inStatus[i,j] \in \{Waiting, Handshaking\}$;
    ac: inStatus[i,j] := Closed;
        if (handshake[j,i] $\in$ {cCpbs, Ack})
         handshake[j,i] := None;
}
```

Figure 10.23: Join/Depart component of Gnu_TCP service in SeSF (Part 2)

```
// Client servent i requests to disconnect its connection to j
 dnw cDepartReq (Node i, Node j) {
    ec: outStatus[i,j] $=$ Closing $\wedge$ cRP[i,j] $=$ Close;
    ac: cRP[i,j] $:=$ None;
 }

// Client servent i's request to disconnect is fulfilled.
 upw cDepartAck(Node i, Node j) {
    ec: outStatus[i,j] $=$ Closing $\wedge$ cRP[i,j] $=$ None;
    ac: outStatus[i,j] $:=$ Closed;
 }

// Server servent i requests to disconnect with j.
 dnw sDepartReq (Node i, Node j) {
    ec: sStatus[i,j] $=$ Closing $\wedge$ sRP[i,j] $=$ Close;
    ac: sRP[i,j] $:=$ None;
 }

// Server servent i's request to disconnect is fulfilled.
 upw sDepartAck(Node i, Node j) {
    ec: inStatus[i,j] $=$ Closing $\wedge$ sRP[i,j] $=$ None;
    ac: inStatus[i,j] $:=$ Closed;
 }

// i terminates abruptly.
 dnw abort (Node i, Node j) {
    ec: i $\neq$ j $\wedge$ (outStatus[i,j] $\neq$ Closed $\vee$ inStatus[i,j] $\neq$ Closed);
    ac: outStatus[i,j] $:=$ Closed;
         inStatus[i,j] $:=$ Closed;
         JR[j,i] $:=$ false;
         cRP[i,j] $:=$ None;
         sRP[i,j] $:=$ None;
          if (handshake[j,i] $\in$ {cCpbs, Ack})
            handshake[j,i] $:=$ None;
          if (handshake[i,j] $=$ sCpbs)
            handshake[i,j] $:=$ None;
}
```

Figure 10.24: Join/Depart component of Gnu_TCP service in SeSF (Part 3)

```
For all the following definitions, i ≠ j ≠ null ∧ i ≠ j

// If i has requested a connection establishment, then eventually
// (1) j is state Waiting (wrt i) or (2) one or both nodes are Closed.
 progress-obligation P1 {
   (outStatus[i,j] = Connecting ∧ inStatus[j,i] ∈ {Closed, Waiting, Handshaking, Connected}) leadsto
     ((outStatus[i,j] = Connecting ∧ inStatus[j,i] = Waiting) ∨ outStatus[i,j] = Closed ∨ inStatus[j,i] = Closed)
 }

// If i is Connecting and j is Waiting, then eventually (1) i is Handshaking, or (2) both i and j are closed.
 progress-obligation P2 {
   (outStatus[i,j] = Connecting ∧ inStatus[j,i] = Waiting) leadsto
   ((outStatus[i,j] = Handshaking ∧ inStatus[j,i] = Waiting) ∨ outStatus[i,j] = Closed ∨ inStatus[j,i] = Closed)
 }

// If i is Handshaking and j is Waiting, then eventually (1) i and j are Handshaking,
// or (2) one or both nodes are closed.
 progress-obligation P3 {
   (outStatus[i,j] = Handshaking ∧ inStatus[j,i] = Waiting) leadsto
     ((outStatus[i,j] = Handshaking ∧ inStatus[j,i] = Handshaking) ∨ outStatus[i,j] = Closed ∨ inStatus[j,i] = Closed)
 }

// If i is Handshaking and j is Handshaking, then eventually (1) i is Connected,
// or (2) one or both nodes are closed.
 progress-obligation P4 {
   (outStatus[i,j] = Handshaking ∧ inStatus[j,i] = Handshaking)  leadsto
   ((outStatus[i,j] = Connected ∧ inStatus[j,i] = Handshaking) ∨ outStatus[i,j] = Closed ∨ inStatus[j,i] = Closed)
 }

// If i is Connected and j is Handshaking, then eventually (1) i and j are Connected,
// or (2) one or both nodes are closed.
 progress-obligation P5 {
   (outStatus[i,j] = Connected ∧ inStatus[j,i] = Handshaking)  leadsto
   ((outStatus[i,j] = Connected ∧ inStatus[j,i] = Connected) ∨ outStatus[i,j] = Closed ∨ inStatus[j,i] = Closed)
 }

// If joinReq(i, j) occurs, then either peerReached(i, j), joinAborted(i, j) or abort(i, j) will eventually be executed.
// Node i cannot stay in state Connecting forever.
 progress-obligation P6 {
    outStatus[i,j] = Connecting leadsto (outStatus[i,j] = Closed ∨ outStatus[i,j] = Handshaking)
 }

// If joinReqInd(i, j) occurs, then cRx(i, j, cpbs[i]), joinReqAbort(i, j) or abort(i, j) is eventually executed.
// Node i cannot stay in state Waiting forever.
 progress-obligation P7 {
    inStatus[i,j] = Waiting  leadsto (inStatus[i,j] = Closed ∨ inStatus[i,j] = Handshaking)
 }

 // Client node i cannot stay in state Handshaking forever.
  progress-obligation P8 {
    outStatus[i,j] = Handshaking leadsto (outStatus[i,j] = Closed ∨ outStatus[i,j] = Connected)
  }
```

Figure 10.25: Join/Depart component of Gnu_TCP service in SeSF (Part 4)

```
   // Server node i cannot stay in state Handshaking forever.
    progress-obligation P9 {
      inStatus[i,j] = Handshaking leadsto (inStatus[i,j] = Closed $\vee$ inStatus[i,j] = Connected)
    }

  // If cDepartReq(i, j) occurs, then either cDepartAck(i, j) or abort(i, j) is eventually executed.
   progress-obligation P10 {
     outStatus[i,j] = Closing leadsto outStatus[i,j] = Closed;
   }

  // If sDepartReq(i, j) occurs, then either sDepartAck(i, j) or abort(i, j) is eventually executed.
   progress-obligation P11 {
     inStatus[i,j] = Closing leadsto inStatus[i,j] = Closed;
   }

  // If a client node has a reply pending, then it eventually replies or the connection is aborted.
   progress-obligation P12 {
     cRP[i,j] $\neq$ None leadsto cRP[i,j] = None $\vee$ outStatus[i,j] = Closed;
   }

   // If a server node has a reply pending, then it eventually replies or the connection is aborted.
   progress-obligation P13 {
     sRP[i,j] $\neq$ None leadsto sRP[i,j] = None $\vee$ inStatus[i,j] = Closed;
   }
}
```

Figure 10.26: Join/Depart component of Gnu_TCP service in SeSF (Part 5)

```
service Gnu_TCP {
  ...
  for the following definitions, $i \in$ Node $- \{$null$\}$.
  Set(GUID) Q[i,j]. Initialized to $\{\}$.
  Set(GUID) QRcvd[i,j]. Initialized to $\{\}$.
  Set(GUID) QHasHit[i,j]. Initialized to $\{\}$.
  Set(GUID) H[i,j]. Initialized to $\{\}$.
  Set(GUID) HRcvd[i,j]. Initialized to $\{\}$.

  function boolean connected(Node i, Node j) {
    return (outStatus[i,j] $=$ Connected $\vee$ inStatus[i,j] $=$ Connected)
  }

  // Srv i sends a query request to j
  dnw TxQuery (Node i, Node j, Query q) {
     ec: $i \neq j \wedge$ connected(i,j) $\wedge$ q.id $\notin$ Q[i,j] $\cup$ QRcvd[i,j];
     ac: Q[i,j] := Q[i,j] $\cup$ $\{$q.id$\}$;
  }

  // Tcp i informs srv i of the arrival of a query request.
  // Srv i returns true if the set of files that satisfy the query is not empty.
  upw boolean RxQuery (Node i, Node j, Query q)) {
     ec: $i \neq j \wedge$ connected(i,j) $\wedge$ q.id $\in$ Q[j,i] $\wedge$ (q.id $\notin$ Q[i,j] $\cup$ QRcvd[i,j);
     ac: QRcvd := QRcvd[i] $\cup$ $\{$q.id$\}$;
         if ( return $=$ true)
            QHasHit[i] := QHasHit[i] $\cup$ $\{$q.id$\}$;
  }

  // Tcp i receives hits from tcp j.
  // Multiple replies (either from different nodes or from the same node)
  // may arrive in response to the same query request.
  upw RxHit (Node i, Node j, GUID id, Set(Hit) hits) {
     ec: $i \neq j \wedge$ connected(i,j) $\wedge$ id $\in$ Q[i,j] $\wedge$ id $\in$ H[j,i];
     ac: HRcvd[i,j] := HRcvd[i,j] $\cup$ $\{$id$\}$;
  }

  // Srv i transmits hits to srv j.
  // Srv i may send multiple replies to the same query (if the set of replies exceeds the MTU of a msg).
  dnw TxHit (Node i, GUID id, Set(Hit) hits) {
     ec: $i \neq j \wedge$ connected(i,j) $\wedge$ id $\in$ Q[i,j] $\wedge$ id $\notin$ H[i,j] $\cup$ HRcvd[i,j];
     ac: H[i,j] := H[i,j] $\cup$ $\{$id$\}$;
  }

  // If i receives a query with "id" and i has a hit, it eventually answers or the connection is closed.
  progress-obligation SP1 {
      (id $\in$ QHasHit[i,j]) leadsto (id $\in$ H[i,j] $\vee$ $\not\text{c}$onnected(i,j))
  }

}
```

Figure 10.27: Query component of Gnu_TCP service in SeSF

Chapter 11

Conclusions and Future Work

We have integrated the SeSF framework for concurrent and distributed systems into Java. The resulting framework, called SeSFJava, can be used to define executable services (i.e., external specification) of concurrent and distributed systems.

We have also implemented a Harness for testing systems against services and against safety and progress assertions, where systems, services, and assertions are specified in SeSFJava. SeSFJava Harness is able to handle general programs, general services, and general safety and progress assertions. SeSFJava Harness can test systems on their actual platforms. It can handle both multi-threaded systems and multi-process systems.

Finally, we have presented two major applications of SeSFJava and the Harness. The first was to the TCP transport protocol, and the second was to a Gnutella network. We wrote the intended services of Gnutella, and tested an open-source implementation, namely Furi, against the services.

The TCP transport protocol application was also done in the context of a senior-level undergraduate introductory networking course at University of Maryland (CMSC417). The use of SeSF significantly increased the percentage of students who completed the projects, reduced their email queries about the specification, and reduced the grading time.

There are several possible areas of future work. One is to extend to web services. Web services, such as stock tickers and inventory check services, are packaged as publicly accessible software components that are invoked by programs. XML is the standard format used for data transmission of web services, and XML query languages are used for manipulation of the data. W3C, in addition to companies (e.g., Microsoft, IBM), is pushing to standardize the web services.

Researchers have used model checkers to check the correctness of these services [60, 24]. But model checkers, while appropriate for finite state machines, do not capture the tree-structure of XML data and the high expressiveness of XML query languages. In addition, most of these methods require the translation of the web services into intermediate languages suitable for analysis. We propose to integrate SeSF and the testing harness with the WSDL [2] interface specification of web services and with their behavioral descriptions (e.g., BPEL [23], WSCI [1]).

Another possible area of work is in testing device drivers. The main reason of crashes of commercial operating systems is the malfunction of device drivers, for example, 85% of crashes of Windows XP are due to errors in device driver [83]. Techniques can be developed to check the correctness of these drivers and to prevent a failed driver from corrupting the kernel. Ours could define SeSF executive services for the interface between the operating system kernel and the device driver. Such services should capture not only the syntax of the interface but also its behavior. This would permit OS developers to test the drivers against their services and thus reduce the number of crashes of the OS.

Appendix A

Preprocessed Code of AccountExample

A.1 BankSystem.java

```
import java.util.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

class BankSystem extends UnicastRemoteObject implements BankInterface  {
   final static int MANUAL              = 0;
   final static int VIEW_AND_AUTOMATIC = 1;
   final static int AUTOMATIC           = 2;
   final static int VIEW                = 3;
   final static int END                 = 4;
   final static int WAIT                = 5;
   void breakpoint(String breakpointName, int mode) {
      breakpoint(breakpointName, null, mode);
   }

   void breakpoint(String breakpointName, String comment, int mode) {
      try {
         Hashtable watches = null;
         Hashtable assertions = checkAssertions(true);
         MarshalledInformation  breakpointData =
            new MarshalledInformation("BankSystem", MarshalledInformation.SYSTEM,
                                breakpointName, mode, comment, watches, assertions);
         breakpointData.threadName = Thread.currentThread().getName();
         tester.breakpoint(breakpointData);
      } catch(RemoteException re){
         re.printStackTrace();
      }
   }

   Hashtable checkAssertions(boolean debugInfo){
      Hashtable assertions = new Hashtable();
         // Assertion: balanceCheck
         balanceCheck(debugInfo);
         if (balanceCheckResult)
            assertions.put("balanceCheck", "(true)");
         else
            assertions.put("balanceCheck", "(false)");
      return assertions;
   }
```

```
static AccountInterface account;
static TesterInterface tester;

static int balance;//init to 0
static final int N =10;
static BankSystem bank;
static Object lock =new Object();
static ClientInterface client []=new ClientInterface [N];//init to null

BankSystem()throws RemoteException {
    account.BankSystem();
    breakpoint(”Bank.constructor”, VIEW_AND_AUTOMATIC);
}

public static void main(String argv [])throws Exception ,RemoteException {
    tester =(TesterInterface) Naming.lookup(”AccountTester”);
    account =(AccountInterface) Naming.lookup(”Account”);
    System.out.println(”Everything  found in rmiregistry”);
    bank =new BankSystem();
    Naming.rebind(”Bank”, bank);
}

public void update(int id,int n,String loc) throws RemoteException {
    try {
        account.update(id, n, loc);
        breakpoint(”xc_event: update”, ”params:(”+”)”, VIEW);
        if (!(id >= 0 &&id < N &&client [id] == null) )
            throw new Error(”Enabled Condition Failure: update ”);
        synchronized(lock) {
            try {
                client [id] =(ClientInterface) Naming.lookup(loc);
            }catch(Exception e) {
                e.printStackTrace();
            }
            new DecThread(id,n).start();
        }
    } catch(RemoteException re) {
        re.printStackTrace();
    }
}

class DecThread extends Thread {
    int id;
    int n;
    DecThread(int id ,int n) {
        this.n =n;
        this.id =id;
    }

    public void run(){
        try {
            breakpoint(”Bank.breakpoint1”, MANUAL);
            synchronized(lock) {
                if (n >= 0 ||balance >= −n) {
                    balance + = n;
                    client [id].ack(id);
                } else
                    client [id].nack(id);
                client [id] =null;
            }
            breakpoint(”Bank.breakpointEnd”, END);
        }catch(RemoteException re) {
            re.printStackTrace();
        }
    }
}//End Thread

static boolean balanceCheckAntecedent1;
```

```
  static boolean balanceCheckConsequent1;
  static boolean balanceCheckPending1;
  static boolean balanceCheckAssertion1;
  static boolean initialPreprocessing1 = true;
  static void initbalanceCheck(){
    balanceCheckAntecedent1=false;
    balanceCheckConsequent1=false;
    balanceCheckPending1=false;
    balanceCheckAssertion1=false;
  }
  static boolean balanceCheckResult = true;
  void  balanceCheck(boolean debugInfo) {
    if(initialPreprocessing1)
      initbalanceCheck();
    balanceCheckAssertion1 = (( balance >=  0) );
    initialPreprocessing1 = false;
    balanceCheckResult = true;
    if (balanceCheckAssertion1)
      System.out.println((debugInfo)? ”Assertion( ” + ”balanceCheckAssertion1” + ”) is valid” : ””);
    else
      balanceCheckResult = false;
  }
}//End system
```

A.2 Account_wrt_Bank.java

```
import java.util.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

class Account_wrt_Bank extends UnicastRemoteObject implements ClientInterface  {
  final static int MANUAL             =  0;
  final static int VIEW_AND_AUTOMATIC =  1;
  final static int AUTOMATIC          =  2;
  final static int VIEW               =  3;
  final static int END                =  4;
  final static int WAIT               =  5;
  void breakpoint(String breakpointName, int mode) {
    breakpoint(breakpointName, null, mode);
  }

  void breakpoint(String breakpointName, String comment, int mode) {
    try {
      Hashtable watches =  null;
      Hashtable assertions =  new Hashtable();
      MarshalledInformation  breakpointData =
        new MarshalledInformation(”Account_wrt_Bank”, MarshalledInformation.SYSTEM,
                     breakpointName, mode, comment, watches, assertions);
      breakpointData.threadName =  Thread.currentThread().getName();
      tester.breakpoint(breakpointData);
    } catch(RemoteException re){
      re.printStackTrace();
    }
  }

  static AccountInterface account;  //Remote pointer to counter ’s user.’
  static TesterInterface tester;

  Object lock = new Object();
  BankInterface bank;
```

```
static final int N = 5;   //Number of clients
int balance = 0;
boolean pending []= new boolean[N];
int amount []= new int[N];

Account_wrt_Bank() throws RemoteException {
    try {
        //Binding ”Account”.
        Naming.rebind(”Account_wrt_Bank”,this);
        System.out.println(”Account bound to registry”);
        bank = (BankInterface) Naming.lookup(”Bank”);
        new Whirl().start();
    }catch(Exception e) {
        e.printStackTrace();
        throw new RemoteException();
    }
}

public static void main(String argv [])throws Exception, RemoteException {
    if (System.getSecurityManager()==null)
        System.setSecurityManager(new RMISecurityManager());
    tester  = (TesterInterface) Naming.lookup(”AccountTester”);
    account = (AccountInterface) Naming.lookup(”Account”);
    Account_wrt_Bank client = new Account_wrt_Bank();
}

public void update(int id, int n, String loc) throws RemoteException {
    synchronized(lock) {
        amount[id]  = n;
        pending[id] = true;
        bank.update(id, n, ”Account_wrt_Bank”);
    }
}

class Whirl extends Thread {
    Random r = new Random();
    public void run(){
        while(true) {
            try {
                synchronized(lock) {
                    int id = r.nextInt(5);
                    if (id >=0 &&id <N &&!pending [id]) {
                        update(id, r.nextInt(80) -40, ”Account_wrt_Bank”);
                    }
                }
                sleep(10);
                yield();
            }catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}

public void ack(int id) throws RemoteException {
    try {
        account.ack(id);
        breakpoint(”xc_event: ack”, ”params:(”+”)”, VIEW);
        synchronized(lock) {
            if(!(id >= 0 &&id < N && pending[id] && (amount [id] >= 0 || balance >= −amount[id])))
                throw new Error(”Account_wrt_Bank.ack Enabling Condition failed.”);
            pending[id] = false;
            balance = balance + amount [id];
        }
    } catch(RemoteException re) {
        re.printStackTrace();
    }
}
```

```
   public void nack(int id) throws RemoteException {
      try {
         account.nack(id);
         breakpoint(”xc_event: nack”, ”params:(”+”)”, VIEW);
         synchronized(lock) {
            if (!(id >= 0 && id < N && pending[id] && balance < −amount [id]) )
               throw new Error(”Account_wrt_Bank.nack Enabling Condition failed.”);
            pending [id] = false;
         }
      } catch(RemoteException re) {
         re.printStackTrace();
      }
   }
}
```

Appendix B

Conversion versus Embedded Markup Language

In this dissertation, we have implemented a preprocessor that accepts files written in SeSFJava, that is, Java files with SeSF markup tags embedded within. Prior to this, we designed a converter that accepts files written in a SeSF markup programming language and converts them to Java files. Figures B.1, B.2 and B.3 illustrate the SeSF version of the BankSystem, ClientSystem and AccountService in the AccountExample of chapter 4. We notice the following:

- The files cannot be compiled using a standard Java compiler.
- The systems make calls to upper and lower services only. The means of communication from a system to another (upper or lower) is not specified.
- Exception handling and synchronization methods are not specified explicitly.

The converter has to insert an RMI call from a system to another, for example, whenever BankSystem calls event ack in AccountService, the call is replaced by an RMI call to the corresponding event ack in system ClientSystem. The converter handles synchronization and exceptions. The programmer is limited to only small templates of code,

```
system_program BankSystem {
  UpperService: AccountService;
   static int balance;
   static final int N = 10;           // number of clients
   int client[] = new client[N];      // true if the client[i] has pending update
  BankSystem() {}

  xc_event void update(int id, int n) {
     ec: id >= 0 && id < N && !client[id];
     ac: client[id] = true;
         start UpdateThread(id, n);
  }

  Thread UpdateThread (int id, int n) {
     try {
       breakpoint("Bank.bpBegin", BEGIN);
       beginAtomic
         if (n >= 0 || balance >= -n) {
           balance + = n;
           AccountService.ack(id);
         } else
           AccountService.nack(id);
         client[id] = false;
       endAtomic
       breakpoint("Bank.bpEnd", END);
     } catch (RemoteException re) { re.printStackTrace(); }
  } //End Thread
} //End System
```

Figure B.1: BankSystem SeSF system program

otherwise the converter cannot convert the files to Java. Although the converter mirrors faithfully the SeSF theory, it has many drawbacks:

- The correctness proof of the conversion from SeSF to Java is almost impossible.

- The converter cannot cope with the diversity and dynamic nature of programming languages. For example, when we implemented the converter, calls where limited to RMI calls only, which ignores other possibilities like TCP and MPI (Message Passing Interface) calls. Also, we used simple synchronization templates which limited the programmers from using more sophisticated methods.

- The performance of the generated code and its readability are questionable.

```
system-program ClientSystem {
  LowerService: AccountService;
    Random  r = new Random();          // random number generator
    boolean wait = false;              // true if it has pending requests, false otherwise

  public static void main(String argv[]) throws Exception {
     ClientSystem client = new ClientSystem();
     client.execute(Intger.parseInt(argv[0]));
  }

  void execute(int id) throws Exception {
     for(int i = 0; i < 50; i++){
        breakpoint(”Client.bpInc”, MANUAL);
       wait = true;
       AccountService.update(id, r.nextInt(80) - 40);
       // Wait for ack or nack
       beginAtomic
          while (wait){
            breakpoint(”Client.bpWait”, WAIT);
            wait;
          }
       beginAtomic
     }
     breakpoint(”Client.bpEnd”, END);
  }

  xc-event void ack(int id) {
     ec: true;
     ac: wait = false;
        notify;
  }

  xc-event void nack(int id) {
     ec: true;
     ac: wait = false;
         notify;
  }
}
```

Figure B.2: ClientSystem SeSF system program

```
service-program AccountService {
  //# Harness harness;
   static final int N = 10;                    // number of clients
   int balance;
   boolean  pending[] = new boolean[N];        // pending[i] is false if it has no pending request
   int  amount[] = new int[N];                 // amount[i] is the update value of user i last request
   dnw-event void BankSystem:update(int id, int n) {
      ec: id >= 0 && id < N && !pending[id];
      ac: amount[id]  = n;
         pending[id] = true;
   }

   upw-event void ClientSystem:ack(int id) {
      ec: id >= 0 && id < N && pending[id] && (amount[id] >= 0 || balance >= -amount[id]);
      ac: pending[id] = false;
         balance  + = amount[id];
   }

   upw-event void ClientSystem:nack(int id) {
      ec: id >= 0 && id < N && pending[id] && balance < -amount[id];
      ac: pending[id] = false;
   }

   progress-obligation pA {
     forall i: 0 - > (N-1)
        beginAssertion
          pending[i] leadsto !pending[i]
        endAssertion
      endfor
  }
}
```

Figure B.3: AccountService SeSF service program

These drawbacks are common to all frameworks that implement the conversion method [81, 85, 22, 84, 28, 66]. This convinced us to favor the markup language methodology.

Appendix C

Complete SeSFJava Programs of Data Transfer Protocol

C.1 SW_Source.java

```
import java.net.*;
import java.lang.*;
import java.io.*;
import java.util.*;
//# import java.rmi.RemoteException;
//# import java.rmi.server.UnicastRemoteObject;
//# import java.rmi.*;

//# system_program: SW_Source
class  SW_Source{
   //# HarnessInterface harness = SourceUser.harness;
   //# varOf(DT) dt;            // Tells the harness that variable dt is an alias of DT
   //# static DTInterface dt;
   //# {
      //# try {
      //#   dt = (DTInterface) Naming.lookup(”DT”);
      //# } catch(Exception e) {System.out.println(”Error”);}
   //# }
   //# final static int mode = VIEW_AND_AUTOMATIC;

   SourceUser  dtsource;
   NetworkSocket nSocket;
   Vector        sendBuf = new Vector ();
   final static int msgSize = 128;
   final static int bufSize = 32*1024;
   final static int SW = bufSize / msgSize;
   int bufUsed, ns, na, sw = SW;
   Timer rTimer = new Timer();
   volatile boolean contWork = true ;
   Object lock = new Object();  // lock object
   final static byte D   = (byte) 1;
   final static byte ACK = (byte) 2;
   final static int headerSize  = 3;
   final static int msgTypeByte = 0;
   final static int seqNoByte0  = 1;
   final static int seqNoByte1  = 2;
   //# watch ns,na,sw;    // Harness monitor any changes in these variables

   SW_Source(int localPort, String remoteDN, int remotePort){
      nSocket    = new NetworkSocket(localPort, remoteDN, remotePort);
      new SourceReceiver().start ();
```

```
    new DataSender().start();
}

//# xc_event;
public void sendData(byte []data)  {
    //# breakpoint(”breakpoint.SW_Source.sendData(”+data.length +”)”, mode);
    synchronized(lock){
        //# ec: bufUsed + data.length <= bufSize && data.length != 0 && data.length % msgSize == 0;
        int length = data.length;
        int pos = 0;
        while (length > 0){
            int effPayload = (length > msgSize)? msgSize : length;
            byte[] msgBuf  = new byte[effPayload];
            System.arraycopy(data, pos, msgBuf, 0, effPayload);
            sendBuf.addElement(msgBuf);
            ns     = ns + 1;
            pos    = pos + effPayload;
            length = length - effPayload;
        }
        bufUsed + = data.length;
    }
}

public void closeSource()  {
    //# breakpoint(”breakpoint.SW_Source.closeSource”, mode);
    synchronized(lock){
        sendBuf.clear();
        contWork = false;
        rTimer.cancel();
        nSocket.close();
    }
}

void sendDataMsg(int j)  {
    synchronized(lock){
        if (!sendBuf.isEmpty() && (j − na) < (ns − na) && (j - na) < sw){
            // Make buffer
            byte tS []  = (byte []) sendBuf.elementAt(j - na);
            int length  = tS.length + headerSize ;
            byte[] datablock = new byte [length];
            System.arraycopy(tS, 0, datablock, headerSize, tS.length);
            datablock[msgTypeByte] = D ;
            datablock[seqNoByte0] = (byte) (j  & 0xFF);
            datablock[seqNoByte1 ] = (byte) (j >> 8);
            nSocket.send(datablock, datablock.length);
            rTimer.schedule(new Retransmission(j), new Date((new Date()).getTime()+ 4000));
        }
    }
}

void receiveACK(int seqNo,int w) {
    boolean ackTheData = false;
    int     ackedBytes = 0;
    synchronized(lock){
        int tmp = seqNo − na;
        if (tmp >= 1){ // && tmp <= (ns − na)){
            for (int i = 0; i < tmp; i++){
                ackedBytes + = ((byte [])sendBuf.elementAt(0)).length;
                sendBuf.removeElementAt(0);
            }
            na = na + tmp;
            sw = w;
            bufUsed − = ackedBytes;
            ackTheData = true;
            sendDataMsg(seqNo);
        } else if (tmp == 0)
            sw = sw > w ? sw : w;
    }
```

```
         if (ackTheData)
            dtsource.ackData(ackedBytes);
      }

      class DataSender extends Thread {
         public void run() {
            Thread.currentThread().setName(”SW_Source.DataSender”);
            int j =0 ;
            while (contWork){
               //# breakpoint(”breakpoint.SW_Source.DataSender.run(”+ j + ”)”, mode);
               sendDataMsg(j);
               synchronized(lock){
                  if (!sendBuf.isEmpty() && (j − na) < sw && (j − na) < (ns − na))
                     j = j + 1;
               }
            }
            //# breakpoint(”SW_Source.DataSender”, END);
         }
      }

      class Retransmission extends TimerTask {
         int earlyJ ;
         Retransmission (int aJ ){
            earlyJ =aJ ;
            Thread.currentThread().setName(”SW_Source.Retransmission”);
         }
         public void run (){
            //# breakpoint(”breakpoint.SW_Source.Retransmission(”+ earlyJ+”)”, mode);
            sendDataMsg(earlyJ);
            //# breakpoint(”breakpoint.SW_Source.Retransmission.End”, END);
         }
      }

      class SourceReceiver extends Thread {
         public void run(){
            Thread.currentThread().setName(”SW_Source.SourceReceiver”);
            //# breakpoint(”breakpoint.SW_Source.SourceReceiver.start)”, mode);
            while (contWork ){
               byte recBuf[] = new byte [100];
               DatagramPacket dp = new DatagramPacket (recBuf, 100);
               try {
                  //# breakpoint(”breakpoint.SW_Source.SourceReceiver.wait”, WAIT);
                  nSocket.receive (dp, 1000);
                  int seqNo  = ((int) (recBuf[seqNoByte1] & 0xFF) << 8) +
                     (int) (recBuf[seqNoByte0] & 0xFF);
                  int w =(int) recBuf[headerSize];
                  //# breakpoint(”breakpoint.SW_Source.SourceReceiver(”+seqNo + ”, ” + w +”)”, mode);
                  receiveACK (seqNo, w);
               }catch (InterruptedIOException iiooe ){
                  // Handle Congestion if needed
               }catch(Exception e){
                  e.printStackTrace();
               }
            }
            //# breakpoint(”SW_Source.SourceReceiver.End”, END);
         }
      }
   }
```

C.2 SW_Sink.java

```
import java.util.*;
import java.net.*;
import java.lang.*;
import java.io.*;
import java.rmi.*;
//# import java.rmi.RemoteException;
//# import java.rmi.server.UnicastRemoteObject;
//# import java.rmi.Naming;

//# system_program: SW_Sink
class  SW_Sink {
    //# HarnessInterface harness = SinkUser.harness;
    //# varOf(DT) dt;         // Tells the harness that variable dt is an alias of DT
    //# static DTInterface dt;
    //# {
        //# try {
        //#   dt = (DTInterface) Naming.lookup(”DT”);
        //# } catch(Exception e) {System.out.println(”Error”);}
    //# }
    //# final static int mode = VIEW_AND_AUTOMATIC;
    SinkUser dtsink;
    NetworkSocket nSocket;
    Vector recvBuf = new Vector ();
    final static int bufSize = 32 * 1024;
    final int msgSize = 128;
    final int RW = bufSize / msgSize;
    int nr, allowedBytes = bufSize;
    Timer dataTimer   = new Timer ();
    Timer ackTimer    = new Timer();
    volatile boolean receiverWork = true;

    Object lock = new Object();

    final static byte NULL = (byte) 0;
    final static byte D    = (byte) 1;
    final static byte ACK  = (byte) 2;

    final static int headerSize     = 3;
    final static int msgTypeByte    = 0;
    final static int seqByte0       = 1;
    final static int seqByte1       = 2;
    final static int windowSizeByte = 3;

    //# watch nr, allowedBytes;    // Harness monitor any changes in these variables

    SW_Sink (int localPort, String remoteDN, int remotePort){
        nSocket    = new NetworkSocket(localPort, remoteDN, remotePort);

        for (int i = 0; i < RW; i++)
            recvBuf.addElement (null);
        new SinkReceiver().start();
        dataTimer.scheduleAtFixedRate (new DataDelivery(), new Date(), 300);
        ackTimer.scheduleAtFixedRate (new AckSender(), new Date(), 400);
    }

    //# xc_event;
    void readyToAccept(int n) {
        //# breakpoint(”breakpoint.SW_Sink.readyToAccept(” + n + ”)”, mode);
        //# ec: true;
        allowedBytes = n;
    }


    void receiveD(int cj,byte []data)  {
        synchronized(lock){
            if ((cj − nr >= 0) && (cj − nr) < RW && data.length ! = 0) {
                int tmp = cj − nr;
                if (recvBuf.elementAt(tmp) == null)
                    recvBuf.set(tmp, data);
            }
        }
    }

    //# xc_event;
    public void closeSink()  {
        //# breakpoint("breakpoint.SW_Sink.closeSink", mode);
        //# ec: true;
        receiverWork = false;
        nSocket.close();
        ackTimer.cancel();
        dataTimer.cancel();
    }

    class AckSender extends TimerTask {
        AckSender() {
            Thread.currentThread().setName("SW_Sink.AckSender");
        }
        public void run() {
            // SendACK
            //# breakpoint("breakpoint.SW_Sink.AckSender(" + nr + ")", mode);
            byte reply []        = new byte [headerSize + 1];
            reply[msgTypeByte]   = ACK;
            reply[seqByte0]      = (byte) (nr & 0xFF);
            reply[seqByte1]      = (byte) (nr >> 8);
            reply[windowSizeByte]= (byte) RW;
            nSocket.send(reply, headerSize + 1);
            //# breakpoint("breakpoint.SW_Sink.AckSender.End", END);
        }
    }

    class DataDelivery extends TimerTask {
        DataDelivery() {
            Thread.currentThread().setName("SW_Sink.DataDelivery");
        }
        public void run()  {
            byte[] delData;
            //# breakpoint("breakpoint.SW_Sink.DataDelivery.start", mode);
            for(;;) {
                //# breakpoint("breakpoint.SW_Sink.DataDelivery.loop(" + nr + ")", mode);
                synchronized(lock){
                    if (!(recvBuf.elementAt(0) != null && allowedBytes > 0))
                        break;
                    byte data []=(byte [])recvBuf.firstElement ();
                    if (data.length <= allowedBytes){
                        recvBuf.removeElementAt(0);
                        recvBuf.addElement(null);
                        nr = nr + 1;
                        allowedBytes − = data.length;
                        delData = data;
                    }else {
                        byte rec[] = new byte [(int) allowedBytes];
                        byte residue[] = new byte [data.length − (int)allowedBytes];
                        System.arraycopy(data, 0, rec, 0, rec.length);
                        System.arraycopy(data, rec.length, residue, 0, residue.length);
                        recvBuf.removeElementAt(0);
                        recvBuf.add(0, residue);
                        allowedBytes = allowedBytes − rec.length;
                        delData = rec;
                    }
                }
                dtsink.deliverData(delData);
            }
            //# breakpoint("breakpoint.SW_Sink.DataDelivery.End", END);
        }
    }

    class SinkReceiver extends Thread {
        SinkReceiver(){
          setName(”SW_Sink.SinkReceiver”);
        }

        public void run() {
          //# breakpoint(”breakpoint.SW_Sink.SinkReceiverStart”, mode);
          while (receiverWork) {
            byte recBuf []= new byte [headerSize + msgSize];
            DatagramPacket dp = new DatagramPacket(recBuf, headerSize + msgSize);
            try {
              //# breakpoint(”breakpoint.SW_Sink.SinkReceiver.waitForMsg”, WAIT);
              nSocket.receive(dp, headerSize + msgSize);
              int cj =((int) (recBuf[seqByte1] & 0xFF) << 8) + (int) (recBuf[seqByte0] & 0xFF);
              byte data[] = new byte[dp.getLength() − headerSize];
              System.arraycopy(recBuf, headerSize, data, 0, data.length);
              //# breakpoint(”breakpoint.SW_Sink.SinkReceiver.MsgRcvd(” + cj + ”)”, mode);
              receiveD(cj, data);
            }catch (InterruptedIOException iioe) {
              // Do Nothing congestion control
            }catch (IOException ioe){
              ioe.printStackTrace();          }
            // this.yield ();
          }
          //# breakpoint(”breakpoint.SW_Sink.SinkReceiver.END”, END);
        }
      }
    }
```

C.3 DT.java

```
import java.util.*;
import java.io.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

//# service_program: DT
class DT  extends UnicastRemoteObject implements DTInterface
  //     The following part describes Data Transfer (DT) service.
  //     DT service specifies a reliable data transfer service  from a
  //     source entity to a sink entity, that is,
  //       - Safety: data is delivered in the same sequence without loss or
  //              duplication.
  //     DT assumes that source and sink are always connected and correctly
  //     initialized.
  //
  //     DT has two groups corresponding to source and sink entities
  //     and events associated with each group.
  //     - Group ”Source” :
  //       Four events are associated with Source:
  //         - constructor(localPort, remoteDN, remotePort, availBufSize)
  //           constructs source entity with parameters: entity’s local
  //           port, remote  domain name, remote port and entity’s buffer
  //           size (in bytes).
  //         - sendData (data)
  //           sends data from local user to source entity to be
  //           delivered to remote user.
  //         - ackData (n)
  //           notifies the entity user that ”n” bytes have been acked by
  //           remote user.
  //         - close();
//                closes the entity.
//      - Group "Sink" :
//          It has four events:
//             - constructor(localPort, remoteDN, remotePort, sinkBufAvail)
//                constructs sink entity with parameters: entity's local
//                port, remote domain name, remote port and entity user
//                 avail buffer size (in bytes).
//             - readyToAccept (n)
//                informs sink entity that its user can accept cumulative
//                amount of data (in bytes) equals to "n".
//             - deliverData(data)
//                delivers "data"  to local user, such that, data is
//                delivered in sequence without loss or duplication.
//             - close();
//                closes the entity.
//
//  Variables :
//  ------------
//  srcHist         : source entity history (<=4GB).
//  srcBufSize      : srcBuf size in bytes.
//  srcBufUsed      : occupied portion of srcBuf in bytes.
//  srcNumSent      : number of bytes accepted from source's local user.
//  srcNumAcked     : number of acked bytes (at source entity).
//  sinkBufAvail    : number of bytes that sink user can accept.
//  sinkNumDelivered : number of bytes delivered to sink user.
//--------------------------------------------------------------------------

final static int msgSize = 128;

// Source entity variables.
ByteArrayOutputStream srcHist = new ByteArrayOutputStream ();
long srcBufSize     = 32 *1024;
int  srcBufUsed     = 0;
long srcNumSent     = 0;
long srcNumAcked    = 0;

// Sink entity variables.
long sinkNumDelivered; // = 0
int sinkBufAvail = 32 *1024 ;

//# Tester tester;
DT() throws RemoteException
   try
      Naming.rebind("DT", this);
    catch (Exception e)
      throw new RemoteException();



//---------------------------------------------------------------------
// Methods called by source side (SW_SourceUser.java and SW_Source.java)
//---------------------------------------------------------------------

//  Sends data from source user to source entity to deliver it to
//  remote user.
//# dnw_event: SW_Source;
public synchronized void sendData(byte []data)  throws RemoteException
   //# ec: srcBufUsed + data.length <= srcBufSize && data.length > 0 && data.length % msgSize == 0;
   srcHist.write(data, 0, data.length);
   srcNumSent + = data.length;
   srcBufUsed + = data.length;



//# dnw_event: SW_Source;
public synchronized void closeSource()  throws RemoteException
   //# ec: true;
//# upw_event: SW_SourceUser;
public synchronized void ackData(int n)  throws RemoteException
    //# ec: srcNumAcked + n  <= srcNumSent;
    //  Notifies user that n bytes have been acked.
    srcBufUsed  = srcBufUsed − n ;
    srcNumAcked = srcNumAcked + n ;


//--------------------------------------------------------------------
// Methods called by sink side (SW_SinkUser.java and SW_Sink.java)
//--------------------------------------------------------------------


//# dnw_event: SW_Sink;
public synchronized void readyToAccept(int n)  throws RemoteException
    //# ec: true;
    //  Informs the entity that user can accept n more bytes of data.
    sinkBufAvail = n;



//# dnw_event: SW_Sink
public synchronized void closeSink()  throws RemoteException
    //# ec: true;



//# upw_event: SW_SinkUser;
public synchronized void deliverData(byte []data)  throws RemoteException
   //# ec: sinkNumDelivered + data.length <= srcNumSent && data.length <= sinkBufAvail && data.length > 0 && correctData (data);
    sinkNumDelivered = sinkNumDelivered +data.length ;
    //  Delivers ”data” received to entity user.
    sinkBufAvail     = sinkBufAvail −data.length ;


boolean correctData (byte[] data)
    byte[] srcData = srcHist.toByteArray();
    for (int i = 0; i < data.length; i++)
        if (srcData [((int) sinkNumDelivered) + i] != data [i])
            return false ;
    return true ;


//#  progress_obligation allDataAcked
//#    beginAssertion
//#       ((sinkNumDelivered > srcNumAcked) leadsto (srcNumAcked == sinkNumDelivered))
//#    endAssertion
//#

//# progress_obligation dataDelivered
//#    beginAssertion
//#       ((srcNumSent > sinkNumDelivered) && (sinkBufAvail > 0)) leadsto (sinkNumDelivered == srcNumSent)
//#    endAssertion
//#
```

BIBLIOGRAPHY

[1] Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/2002/NOTE-wsci-20020808/.

[2] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1), 2000.

[4] Bowen Alpern, Jong-Deok Choi, Ton Ngo, and Manu Sridharan. DejaVu: Deterministic Java replay debugger for Jalapeno JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00) (Demo)*, October 2000.

[5] Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: a formal model for dynamic systems. In *CONCUR'01, the International Conference on Concurrency Theory*, Aalborg, Denmark, August 2001.

[6] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. In *Second ACM SIGACT-SIGCOPS Symposium on Principles of Distributed Computing*, pages 131–142, Montreal, August 1983.

[7] R.J.R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[8] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast, 2002.

[9] Saddek Bensalem, Vijay Genesh, Yassine Lakhnech, Cesar Munoz, Sam Owre, Herald Rues, John Rushby, Vlad Rusu, Hassan Saidi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In *Fifth NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000.

[10] D. Brand and P. Zafiropulo. On communicating finite state machines. *J. ACM*, 30(2):323–342, April 1983.

[11] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[12] K.M. Chandy and J. Misra. *A Foundation of Parallel Program Design*. Addison-Wesley, Reading, MA., 1988.

[13] Jong-Deok Choi, Bowen Alpern, Ton Ngo, Manu Sridharan, and John Vlissides. A perturbation-free replay platform for cross-optimized multithreaded application. In *15th International Parallel and Distributed Processing Symposium*, April 2001.

[14] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. *Proceedings of the International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[15] Edmund M. Clarke and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages*, 21(4), July 1994.

[16] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[17] R. Cleaveland, J. Gada, P. Lewis, S. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory - practical tools for specification, 1994.

[18] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[19] Clip2.com. The Gnutella protocol specification v.0.4, March 2001. http://www.clip2.com/ GnutellaProtocol04.pdf.

[20] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code.

In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[21] Doron Drusinsky. The temporal rover and the ATG rover. In *SPIN*, pages 323–330, 2000.

[22] R. Eschbach, U. Glässer, R. Golzhein, M. Lwis, and A. Prinz. Formal defintion of SDL-2000 - compiling and running SDL specifications as asm models. *Journal of Universal Computer Science*, 7(11), 2001.

[23] Business Process Execution Language for Web Services (BPEL) 1.1. http://www.ibm.com/developerworks/library/ws-bpel.

[24] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT: A tool for formal analysis of web services. In *16th International Conference on Computer Aided Verification*, July 2004.

[25] Stephen J. Garland and Nancy Lynch. IOA: A language for specifying programming and validating distributed systems, December 2000.

[26] Kenneth J. Goldmann, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The programmers' playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

[27] Alex Groce and William Visser. Model checking Java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, July 2002.

[28] David Hansel, Rance Cleaveland, and Scott A. Smolka. Distributed prototyping from validated specifications. *12th International Workshop on Rapid System Prototyping*, June 2001.

[29] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[30] John Hatcliff and Matthew Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *Proceedings of CONCUR 2001,* June 2001.

[31] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[32] Dennis Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 176–181. ACM Press, 2001.

[33] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[34] Gerard Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[35] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, November 1990.

[36] KaZaa homepage. http://www.kazaa.com/.

[37] Limewire homepage. http://www.limewire.com/.

[38] Napster homepage. http://www.napster.com/.

[39] Phex homepage. http://phex.kouk.de/.

[40] Igor Ivkovic. Improving gnutella protocol: Protocol analysis and research proposals. Technical report, LimeWire LLC, 2001.

[41] M. Kim, M. Viswanathan, I. Lee, H. Ben-Abdellah, S. Kannan, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the European Conference on Real-Time Systems*, York, UK, June 1999.

[42] James Kurose and Keith Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2001.

[43] Simon. S. Lam and A.Udaya Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, 16:755–775, July 1990.

[44] Leslie Lamport. The temporal logic of actions. Technical report, DEC SRC Report 57, 1991. April 1990, Revised April 1991.

[45] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[46] I. Lee, S. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, June 1999.

[47] Seungjoon Lee, Rob Sherwood, and Bobby Bhattacharjee. Cooperative peer groups in nice. In *IEEE Infocom*, April 2003.

[48] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *2000, Proceedings of the International Symposium on Software Testing and Analysis*, pages 26–38, August 2000.

[49] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., August 1987.

[50] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4:257–289, 1984.

[51] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[52] Jeremy Manson and William Pugh. The Java memory model simulator. In *Workshop on Formal Techniques for Java-like Programs, in Association with ECOOP*, June 2002.

[53] Ken McCrary. JTella, 2000. http://jtella.sourceforge.net/.

[54] K. L. McMillan. The SMV system, February 1992.

[55] Raymond Miller. Passive testing of networks using a a CFSM specification. In *IEEE International Performance, Computing and Communications Conference*, pages 111–116, February 1998.

[56] Raymond E. Miller and Khaled A. Arisha. Fault coverage in networks by passive testing. In *International Conference on Internet Computing*, pages 413–419, 2001.

[57] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[58] Jaydev Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.

[59] Sandra Murphy and A. Udaya Shankar. Connection management for the transport layer: Service specification and protocol verification. *IEEE Transactions on Communications*, 39(12):1762–1775, December 1991.

[60] Shin Nakajima. Verification of web service flows with model-checking techniques. In *Proceedingsof the First International Symposium on Cyber Worlds*, pages 378 – 385, November 2002.

[61] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM TOPLAS*, 4:455–495, July 1982.

[62] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[63] JunCheol Park and Raymond Miller. A compositional approach for designing multifunction time-dependent protocols. In *IEEE International Conference on Network Protocols*, pages 105–112, October 1997.

[64] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th ACM Symposium on the Foundation of Computer Science*, pages 46–57, November 1977.

[65] Amir Pnueli. *The Temporal Semantics of Concurrent Programs*, volume 70, pages 1–20. Springer-Verlag, July 1979.

[66] Andreas Prinz and Martin Lwis. Generating a compiler for SDL from the formal language definition. In *Lecture Notes in Computer Science*, volume 2708, pages 150–165. Springer-Verlag Heidelberg, January 2003.

[67] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[68] Nicholas Rescher and Alsadair Urquhart. *Temporal Logic*. Springer-Verlag, New York, 1971.

[69] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[70] John Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. Tutorial presented at FORTE X/PSTV XVII ’97, November 1997.

[71] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, 26(3), March 1993.

[72] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[73] F.B. Schneider and G.R. Andrews. Concepts for concurrent programming. In *Current Trends in Concurrency, LNCS 224*, pages 669–716. Springer-Verlag, New York, 1986.

[74] Steve Schneider. Abstraction and testing. In *FM'99, Vol. I, LNCS 1708*, pages 738–757. Springer-Verlag Berlin Heidelberg, 1999.

[75] Beth A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78, June 1995.

[76] A. Udaya Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7(3):281–316, August 1989.

[77] A. Udaya Shankar. Modular design principles for protocols with an application to the transport layer. *Proceedings of IEEE*, 79(12):1687–1707, December 1991.

[78] A. Udaya Shankar. *Concurrent Systems and Services: Design, Verification and Testing*. in preparation, 2005.

[79] N. Shankar. PVS: combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-*

Aided Design (FMCAD '96), volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.

[80] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR'00: Concurrency Theory, Lecture Notes in Computer Science, Number 1877*, pages 1–16. Springer-Verlag, State College, PA., August 2000.

[81] R. Sijelmassi and B. Strausser. The pet and dingo tools for deriving distributed implementations from Estelle. *Computer Networks and ISDN Systems*, 25:841–851, 1993.

[82] Ion Stoica, Robert Morris, David Karger, M. Francs Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*, pages 149–160. ACM Press, 2001.

[83] Michael Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[84] Joshua Tauber. *Verifiable Code Generation from Abstract I/O Automata Models for Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, March 2001.

[85] J. Thees and R. Golzhein. The eXperimental Estelle Compiler - automatic generation of implementations from formal specifications. In *Proceedings of the 2nd Work-*

shop on Formal Methods in Software Practice, Clearwater Beach, Florida, March 1998.

[86] Tomas E. Uribe. Combinations of model checking and theorem proving. In *Frontiers of Combining Systems*, pages 151–170, 2000.

[87] William Wong. Furi homepage, 2003. http://schnarff.com/gnutelladev/source/furi/.

[88] Pamela Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17:212–225, 1991.