SeSFJava Harness: Service and Assertion Checking for Protocol Implementations

Tamer Elsharnouby and A. Udaya Shankar

Abstract

Many formal specification languages and associated tools have been developed for network protocols. Ultimately, formal language specifications have to be compiled into a conventional programming language, and this involves manual intervention (even with automated tools). This manual work is often error-prone because the programmer is not familiar with the formal language. So our goal is to verify and test the ultimate implementation of a network protocol, rather than an abstract representation of it.

We present a framework, called SeSF (Services and Systems Framework), in which implementations and services are defined by programs in conventional languages, and mechanically tested against each other. SeSF is a markup language that can be integrated with any conventional language. We integrate SeSF into Java, resulting in what we call SeSFJava. We present a service-and-assertion checking harness for SeSFJava, called SeSFJava harness, in which distributed SeSFJava programs can be executed, and the execution checked against services and any other correctness assertions. The harness can test the final implementation of a concurrent system. We apply present an application to a data transfer service and sliding window protocol implementation. SeSFJava and the harness has been used in networking courses to specify, develop, and test TCP-like transport protocols and service.

Index Terms

Formal methods, software testing, protocol design, Java, computer networks.

I. Introduction

Many formal specification languages have been developed for network protocols, for example, SDL [1], Estelle [2], [3], Lotos [4] and IOA [5]. Tools [6], [7], [8], [9], [10], [11], [12] have been devised to verify and test the correctness of network protocols against their formal specifications. The formal specification, or "service" as we shall refer to it, describes the desired external behavior of the protocol. Developers use formal languages, as opposed to the conventional programming languages (e.g., C, C++, Java) because of their formal semantics and mathematical elegance, which eases verification and testing of protocols. Yet in order for these languages to be directly used in practical software development, the formal language specification needs to be compiled to a conventional programming language at some point during the development phase. The compilation is done either manually [13] or automatically [14], [15], [16], [17], [18], [19]. Manual translation is prone to errors, often due to the programmer's lack of expertise with the formal language. Moreover, most automatic techniques involve manual intervention during the compilation [20]. Fully automatic compilation brings concerns about the efficiency, code optimization, readability and correctness of the generated code.

Ultimately, most network protocols have to be translated to conventional programming languages, and most translations require manual intervention at some point during the translation. So, our goal is to verify and test the actual implementation of a network protocol rather than a simplified abstract representation of it. In order to achieve this goal, we have developed a framework, called SeSF (**Se**rvices and **S**ystems **F**ramework), that (1) allows definitions of implementations and services in conventional languages, (2) formalizes the notion of an implementation satisfying its services, and (3) provides a means for mechanical testing [21]. SeSF is an imperative, or procedural, version of the formalism in [22]. The main difference between SeSF and most other formalisms [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33] is that SeSF stays close to the programming world. Programming languages, and thus implementations, make use of dynamic creation, naming and deletion of objects, processes and threads, complex I/O control blocks such as network sockets, etc. SeSF explicitly captures such features.

A. SeSF

Like most formalisms, SeSF provides a **compositional methodology** for the design and implementation of concurrent systems. Compositionality means that the design and implementation of a concurrent system can be broken up into the design and implementation of component concurrent systems. We refer to implementations as **systems** and external behavior specifications as **services**. In SeSF, both systems and services are specified by programs in conventional programming languages.

A system specification is intended for execution. Hence, its programs must satisfy the computational, synchronization, and other constraints of the underlying platform– for example, accounting for whether the platform has a single processor, a multi-processor with shared memory, or a set of loosely-coupled message-passing processors.

The service specification states all (and only) the desired properties of the system's execution, unencumbered by internal structure and computational, implementation and synchronization issues. 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.

Roughly speaking, a system **satisfies** its services above and below if the interactions it initiates are allowed by the services, *assuming* the interactions initiated by the system's environment are allowed by the services. 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.

Because services are defined by conventional programming languages, they are **executable**. 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 satisfied.

Using conventional languages for specifying services, instead of a high-level specification language, has certain advantages and disadvantages. One advantage 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 is that service specifications become invariably larger in size, making mechanical verification infeasible, although we think that this is not a big loss because mechanical verification is currently impractical for realistic systems (with parameters and unbounded state). Another concern is that most programming languages suffer from inconsistencies and ambiguities, and one has to avoid such constructs in service specifications. For example, Java has an ambiguous memory model [34], [35], and different Java implementations have different memory models.

B. SeSFJava

SeSF is a markup language that can be integrated with any programming language. We choose to integrate SeSF with Java, resulting in what we call **SeSFJava**. Java is chosen 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. Hence, a SeSFJava program can be compiled and executed as a Java program. But because of the SeSF tags, it can also be tested. We have developed a testing harness, called **SeSFJava harness**, that can execute a distributed system of SeSFJava programs and check whether the resulting execution satisfies the relevant services and any other desired correctness assertions (also specified in SeSFJava).

SeSFJava harness is able to handle general Java programs (e.g., unbounded-state programs) and general safety and progress assertions (e.g., invariant and leads-to assertions). It tests the implementation on its actual platform, without altering the program to run on a simplified platform (e.g., over TCP/IP network sockets rather than a thread-based emulation). It helps the programmer check systems during the development phase; we are not concerned with black-box testing.

C. Data Transfer Application

SeSFJava and the harness has been used in several applications, including TCP-like transport protocols involving both data transfer and connection management [36]. Preliminary versions of SeSFJava and the harness have been used in defining and testing transport protocol projects in computer network courses [37], [38]. Here, for space reasons, we will present an example involving only the data transfer part of the transport protocol application, 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. See fig. 1. 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 here has modulo-N sequence numbers and variable receive window for flow control [39], and is significantly more complex than stop-and-wait or go-back-N protocols. We assume fixed size messages, for space 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 specifies 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.

D. Paper Organization

Section II describes SeSF. Section III introduces SeSFJava and SeSFJava harness. Section IV concludes.

II. 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 progress assertions.

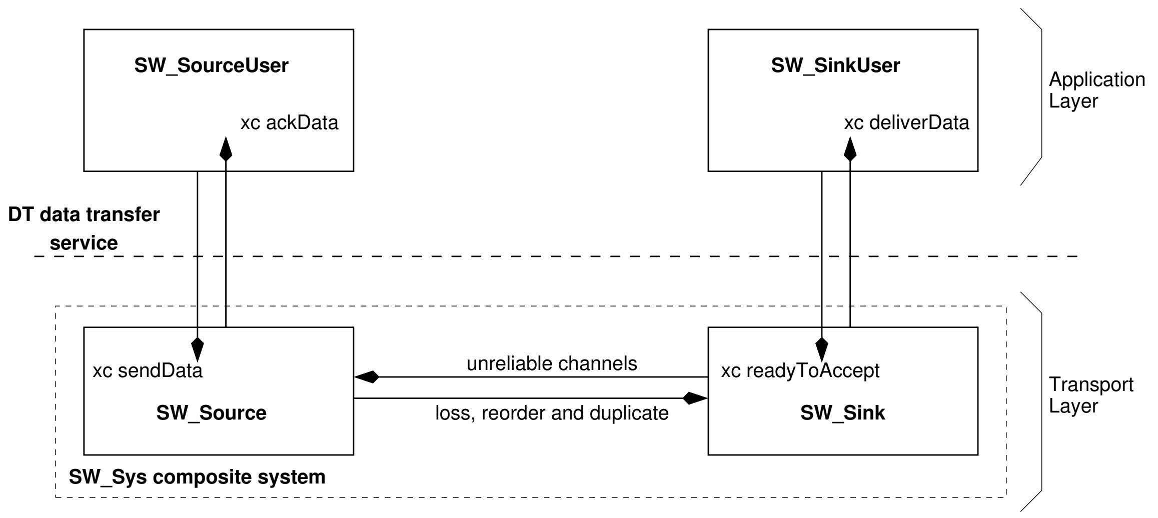


Fig. 1. Data transfer service and protocol system

A. Systems

In SeSF, a **system** is a collection of processes that execute **system programs**. We refer to atomically executed statements as **events**. A system program consists of a header, declarations, externally-controlled events (initiated by the environment), locally-controlled events (initiated by the system), and progress assumptions. The header indicates the system program's name and any parameters and their types. The declarations define constants, types, variables, constructors and functions, as in any procedural language. Fig. 2 and 3 illustrate the system programs for SW_Source and SW_Sink systems, respectively.

The **externally-controlled events**, denoted **xc-events**, are functions that can be called by the environment (e.g., xc-event readyToAccept in fig. 3). The header of an xc-event indicates the **event'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. An event call is **enabled** if the event's enabling condition holds for the parameters (if any) of the call. The **action** is the code that is executed when the event is called. It is the caller's responsibility to only make enabled event calls. It is the callee's responsibility to atomically execute the action if the call is enabled. The action has no event calls, and returns a value if the return type is not void.

The **locally-controlled events**, denoted **lc-events** are the atomic statements in the code executed by the

Description of sliding window protocol (source side):
At any time at the source, let $\mathsf{sendBuf}[0, 1, \ldots, (\mathsf{ng} - 1)]$ denote the sequence of data messages generated by the source. Of these, $\mathsf{sendBuf}[0, 1, \ldots, (\mathsf{ns}-1)]$ have been sent, and $\mathsf{sendBuf}[0, 1, \ldots, (\mathsf{na}-1)]$ have been sent and acknowledged. $\mathsf{sendBuf}[\mathsf{na}, \mathsf{na}+1, \ldots, \mathsf{ns}]$ 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 is $\mathsf{sw} \leq$ constant SW. $[\mathsf{na}..(\mathsf{na} + \mathsf{sw} - 1)]$ constitutes the *send window*.

```
system-program SW_Source { // system header
  // Declarations
  int ng,     // number of data blocks generated by local user, initially 0.
     ns,     // number of data blocks sent at least once, initially 0.
     na,     // number of data blocks acknowledged, initially 0.
     sw,     // send window size, initially SW.
     bufUsed; // occupied portion of buffer in bytes, initially 0;
  Buffer sendBuf;     // Send buffer of SW equally-sized data blocks.
                  // “remove sendBuf[0]” removes first data block from the top and adds empty at the end.
  constant int bufSize;     // buffer size is constant (SW $*$ message size).
  SW_SourceUser sourceuser; // reference to the user application for callback methods

  xc-event void sendData(byte[] data)  { //xc-event header
    ec: bufUsed $+$ data.length $\leq$ bufSize $\wedge$ data.length $\neq 0$;     // data.length is the number of bytes in array data;
    ac: Divide data into messages;
       tmp = number of data messages;
       Store tmp messages in sendBuf; // sendBuf[ng..ng+tmp-1] = data[...];
       ng = ng $\oplus$ tmp;
       bufUsed $+=$ data.length;
  }

   Thread DataSender {
     while $\langle$ (1 $\leq$ ns $\ominus$ na  $<$ min (ng, na $+$ sw) $\ominus$ na) {   // each iteration is executed atomically
          Send Data message (ns);   // via unreliable channel
          Reset timer of retransmission of ns;
          ns $=$ ns $\oplus$ 1;
     } $\rangle$
   }

   Thread Retransmission (int seqNo) {
     while $\langle$ (0 $\leq$ seqNo $\ominus$ na $<$ ns $\ominus$ na $\wedge$ timeout fires) { // each iteration is executed atomically
          Send Data message (seqNo);  // via unreliable channel
          Reset timer of retransmission of seqNo;
     } $\rangle$
   }

   Thread SourceReceiver {
      while(true) {
         Receive message ACK(seqNo, w);     // Blocks till an ACK message is recieved with sequence number (seqNo)
                                             // and window size (w)
         $\langle$
          int tmp = seqNo $\ominus$ na;     // number of newly acked messages
          if (1 $\leq$ tmp $\leq$ (ns $\ominus$ na)) {
             int ackedBytes $=$ tmp $*$ message size;
             remove first tmp messages from sendBuf;
             na $=$ na $\oplus$ tmp;
             sw $=$ w;
             bufUsed $-=$ ackedBytes;
             sourceuser.ackData(ackedBytes);    // output
           } else if (tmp $==$ 0)
             sw $=$ max(sw, w);
         $\rangle$
      }
   }

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

Fig. 2. SW_Source: sliding window source system program

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

```
system-program SW_Sink {   // system header
  int allowedBytes,      // number of the bytes that SW_Sink is able to foist on user's buffer.
     rw,               // receive window size.
     nr;               // number of data blocks delivered to the local user.
  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 ModifyWindow {
    while ⟨ (rw < RW)
          rw =rw + 1;
        ⟩
  }

  Thread DataDelivery () {
     while ⟨ (recvBuf[0] ≠ null ∧ allowedBytes ≥ 0) {
          data = recvBuf[0];
          allowedBytes − = data.length;
          sinkuser.deliverData(data);      // output
          remove recvBuf[0];
          nr = nr ⊕ 1;
          rw − = 1;
     }   ⟩

  }

  Thread SinkReceiver {
     while (true) {
        Receive message Data (cj, data); // Blocks until a Data message with sequence number (cj) and contents (data).
        ⟨
         int tmp = cj  ⊖ nr;
         if (0 ≤ tmp < rw)
            recvBuf[tmp] = data;
         Send ACK message ACK(nr, rw);
        ⟩
     }
  }

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

Fig. 3. SW_Sink: sliding window sink system program

system. Atomically-executed code is indicated by enclosing it in angled brackets (e.g., see DataDelivery thread in fig. 3; we use large-scale atomicity to keep the example small). An lc-event can make at most one event call in any execution. A 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.

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** [30]. Weak fairness of event e, denoted **wfair**(e), means that if event e is *continuously enabled* beyond a certain point, it will eventually be executed. Strong fairness of event e, denoted **sfair**(e), means that if event e is *enabled infinitely often* beyond a certain point, it will eventually be executed. Weak fairness of a thread denotes weak fairness of all events that this thread encounters.

SeSF uses 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.

An **execution** of a system is a sequence of event executions along with the states traversed, starting from an initial state. A **fault transition** represents an event execution 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-free execution** of a system is an execution that contains no fault transitions; it can be finite or infinite.

Sliding window protocol: We briefly comment on SW_Sys in fig. 2 and 3. 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 SW_Source.sendData. SW_Source divides the received array into messages, and sends those messages to SW_Sink. When SW_Sink receives the data message, it replies with an ACK message. If SW_SinkUser has enough space (var SW_Sink.allowedBytes), SW_Sink delivers the data message to its user; otherwise SW_Sink waits for SW_SinkUser to call SW_Sink.readyToAccept before delivering more data to the user. Whenever SW_Source receives a new ACK (not a duplicate), it calls SW_SourceUser.ackData to inform the user that it has more empty space in the buffer.

B. Services

In SeSF, a service defines the acceptable sequences of interactions between systems in different layers. A service is specified by a **service program**. A service program consists of service's header, declarations, events and progress obligations. The header and declarations are similar to those of systems. Fig. 4 illustrates DT service program.

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 (e.g., dnw-event sendData in fig. 4). **Upw** events correspond to xc events of the system above the service callable by the system below the service (e.g., upw-event ackData in fig. 4). The event header indicates the event's signature, consisting of the type (upw or dnw), return type (which may be void), event name, and parameters (if any) and their types. The event corresponds to an xc event with the same signature. The system program that this xc event belongs to is stated in the event header.

The **progress obligations** of a service define the progress that is expected in executing upw events. Progress obligations consist of fairness assertions (described above) and **leadsto** assertions. A leadsto assertion has the form P leadsto Q, where P and Q are predicates (assertion allDataAcked in fig. 4). Formally, P leadsto Q holds for an execution iff the execution is fault-free and for every state in the execution that satisfies P, either that state satisfies Q or some later state satisfies Q. P leadsto Q holds for a system iff it holds for every **complete** execution of the system, i.e., an execution that satisfies the system's fairness assumptions. Service programs should not impose any progress obligations on dnw events.

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 should not have any faulty executions.

Data transfer service: We briefly comment on program DT. Dnw event DT.sendData corresponds to SW_SourceUser passing data to SW_Source. The 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 advertize its window

```
service-program  DT { // service program's header
   // Declarations
   // 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 zero.
      srcNumAcked;  // number of acked bytes (at source entity), initially 0.

   // Sink side variables.
   int sinkNumDelivered, // number of bytes delivered to sink user, initilly 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;
      ac: // data.length is number of bytes in data array
         srcHist[srcNumSent .. srcNumSent + data.length - 1] = data[0..data.length];
         srcNumSent + = data.length;
         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 ∧
         correctData (data);
      ac: sinkNumDelivered = sinkNumDelivered + data.length;
         sinkBufAvail     = sinkBufAvail − data.length;
   }

   boolean correctData (byte[] data){
      dataSize = data.length;
      return (srcHist[sinkNumDelivered .. sinkNumDelivered + dataSize] == data[0..dataSize]);
   }

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

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

}
```

Fig. 4.   DT: data transfer service program

at any time (via dnw event DT.readyToAccept). Whenever DT.ackData is called, it checks whether the data has been delivered to the user.

C. Assertions

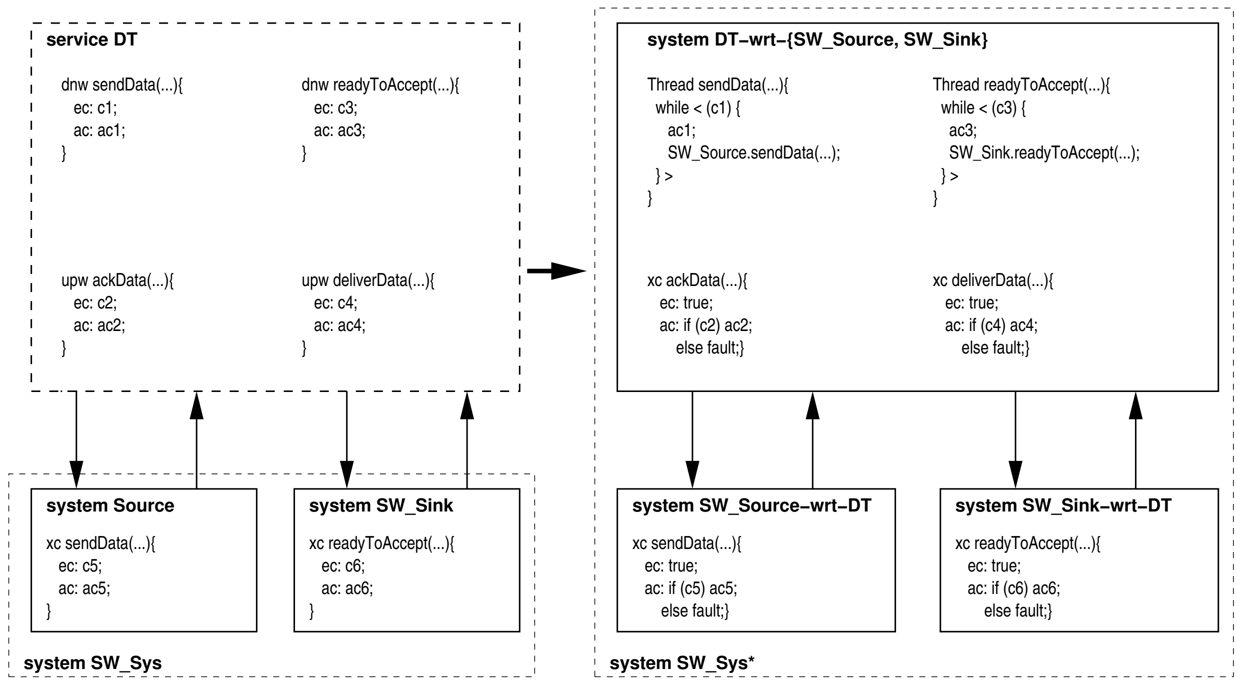
SeSF uses assertions to specify properties of system executions. Assertions are divided into safety and progress assertions. Progress assertions (wfair, sfair, leadsto assertions) have been described before. Safety assertions, in particular, invariant assertions, are needed for reasoning about system executions. An **invariant assertion** has the form inv(P), where P is a predicate. Formally, inv(P) holds for an execution iff the execution is fault-free and every state in the execution satisfies P. For example, inv(na $\leq$ nr $\leq$ ns $\leq$ ng) is an assertion that we need to establish for the SW_Sys system. An assertion holds for a system iff it holds for every execution of the system (and the system has no faulty executions).

D. Service Satisfaction

We now define what it means for a system to satisfy a service. 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 a upw event of V, and every event that M calls in its environment corresponds to a 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. Typically system M is itself a distributed system.

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

Definition [22]: M **satisfies** U **above and** V **below**, or as we prefer say, M **offers** U **uses** V, iff for every execution σ of M that is safe wrt U and V, (1) M is ready to accept every input that extends σ



service DT
dnw sendData(...){ ec: c1; ac: ac1; }
dnw readyToAccept(...){ ec: c3; ac: ac3; }
upw ackData(...){ ec: c2; ac: ac2; }
upw deliverData(...){ ec: c4; ac: ac4; }
system Source
xc sendData(...){ ec: c5; ac: ac5; }
system SW_Sink
xc readyToAccept(...){ ec: c6; ac: ac6; }
system SW_Sys
system DT-wrt-{SW_Source, SW_Sink}
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;}
system SW_Source-wrt-DT
xc sendData(...){ ec: true; ac: if (c5) ac5; else fault;}
system SW_Sink-wrt-DT
xc readyToAccept(...){ ec: true; ac: if (c6) ac6; else fault;}
system SW_Sys*

Fig. 5. Program-version service satisfaction transformations

safely, (2) M does an output only if it extends σ safely wrt U or V, and (3) σ is complete wrt U assuming that σ is complete wrt M and V.

This 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. Accordingly, we use an equivalent **program-version** of service satisfaction [21]. For space reasons, we present the program-version satisfaction conditions for M offers U only.

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

- Define the system M-wrt-U to be M with every output call e(x) changed to a call of the corresponding service event in U.
- Define system U-wrt-M to be U with every upw-event changed to an xc event, every dnw-event changed to an lc-event that also calls the corresponding xc event of M, and every progress assumption set to null.

Program-version definition: Let M^* be the composite system of M-wrt-U and U-wrt-M:

- The safety condition for M offers U holds iff M^* is fault-free.
- The progress condition for M offers U holds iff M^* satisfies the progress obligations of U.

Fig. 5 illustrates the construction of $\mathsf{SW\_Sys}^*$ of $\mathsf{SW\_Source}$-wrt-DT, $\mathsf{SW\_Sink}$-wrt-DT and DT-wrt-$\{\mathsf{SW\_Source}, \mathsf{SW\_Sink}\}$ from $\mathsf{SW\_Sys}$ and DT. (In particular, every output call in $\mathsf{SW\_Source}$ and $\mathsf{SW\_Sink}$ is replaced by a call to the corresponding event of DT by appropriately modifying variables $\mathsf{sourceuser}$ and $\mathsf{sinkuser}$.)

The safety condition for $\mathsf{SW\_Sys}$ offers DT reduces to the following:

1) $\mathsf{SW\_Sys}^*$ does not have undefined values or operations (division by zero, signature-inconsistent call, type mismatch, etc.).
2) $\mathsf{SW\_Sys}^*$ does not call a disabled event, which reduces to the following predicates being invariant:
    - $\mathsf{DT.sendData.ec} \Rightarrow \mathsf{SW\_Source.sendData.ec}$

      (This formalizes the constraint that $\mathsf{SW\_Source.sendData}$ should be enabled whenever its user can call $\mathsf{DT.sendData}$. The predicates below are similarly obtained.)
    - $\mathsf{DT.readyToAccept.ec} \Rightarrow \mathsf{SW\_Sink.readyToAccept.ec}$
    - $\mathsf{SW\_Source}$ is at sourceuser.ackData$(\cdots) \Rightarrow \mathsf{DT.ackData.ec}$
    - $\mathsf{SW\_Sink}$ is at sinkuser.deliverData$(\cdots) \Rightarrow \mathsf{DT.deliverData.ec}$

The progress condition holds iff $\mathsf{SW\_Sys}^*$ satisfies progress obligations $\mathsf{allDataAcked}$ and $\mathsf{dataDelivered}$ assuming weak fairness of $\mathsf{SW\_Sys}$'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 [39]).

E. Service-and-Assertion Checking Harness

The program-based formulation of service satisfaction paves a way to mechanically test a system against services. Consider a system M that is encapsulated by a service U above and a service V below. One can test that M offers U uses V by (1) constructing the composite system M^* of M-wrt-$\{\mathsf{U}, \mathsf{V}\}$, U-wrt-M and V-

wrt-M, and (2) executing M^* and checking whether the generated execution satisfies the program-version conditions.

One of our goals is to test M^* on the same platform as M. Otherwise, we would have to modify M's platform-dependent constructs, e.g., I/O, communication, synchronization, etc. Such modifications would not only be very onerous, but they would most likely change M to a point that defeats the very purpose of testing.

In practice, in order to check program-version conditions, we execute M^* according to the interleaving model, with the interactions between M-wrt-$\{\mathsf{U},\mathsf{V}\}$ and U-wrt-M and between M-wrt-$\{\mathsf{U},\mathsf{V}\}$ and V-wrt-M being executed atomically. To do this, we introduce a **serializer** module that interacts with different components of M^* to allow one eventr at a time. We will elaborate on this in sec. III-C

Figure 6 gives the overall structure and operation of the SeSFJava harness. System and service program files are fed to the preprocessor, which generates the composite system program, the assertion checker (checks service and system assertions), and the "serializer and assertion checker" (SAC) module. The composite system is executed under serializer control, which ensures atomicity of the interactions. Data relevent to the assertion checking is sent to the assertion checker, system and service properties are checked, and violations are recorded. Users can interact with the composite system during its execution to view the results of evaluating assertions and/or to influence the flow of execution.

III. SeSFJava By Example

This section introduces SeSFJava and the SeSFJava harness using the data transfer example. We consider the configuration of fig. 1, with applications SW_SourceUser and SW_SinkUser, and the associated transport entities SW_Source and SW_Sink. For space reasons, we will not talk about the application level of this example. The following subsections describe the composite system SW_Sys, the DT service, and testing SW_Sys against DT.

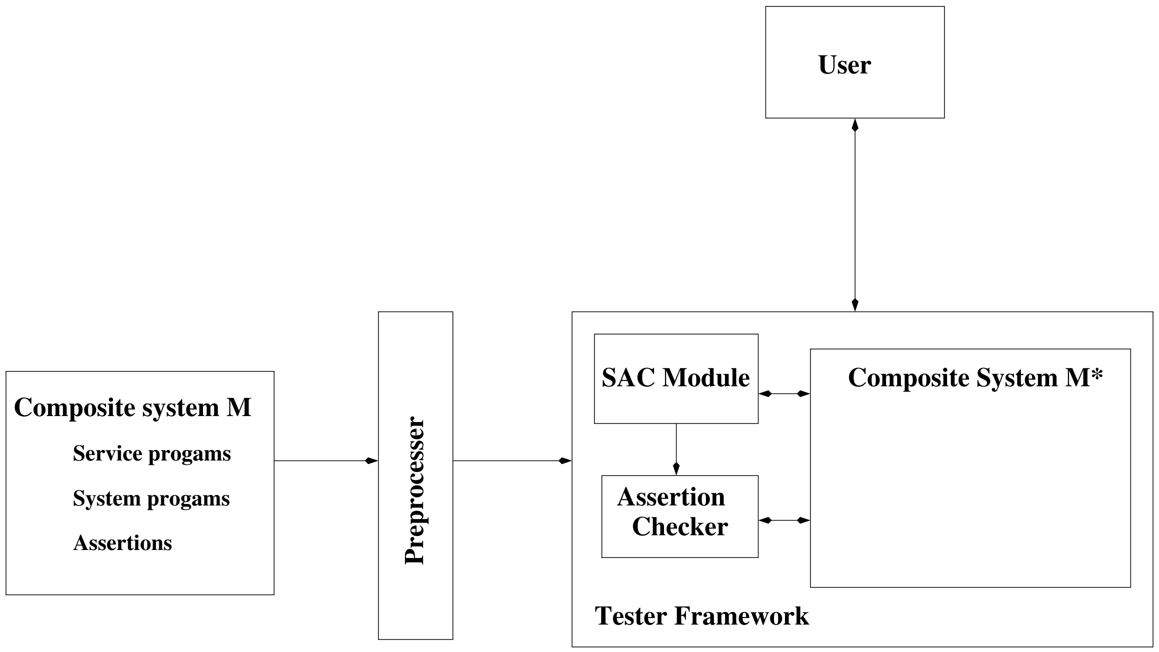


Fig. 6. SeSFJava harness: operation overview

A. SeSFJava 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 SW_Source system program (figure 7). It has different kinds of SeSF tags:

- Tags of the form "//# system_program" precede and identify the system program, in this case, the system program class SW_Source.
- Tags of the form "//# xc_event;" precede and identify the xc events of the program. There is one xc event, namely sendData(data), corresponding to the user passing data to be sent to the sink.
- Tags of the form "//# ec: <predicate>" specify the enabling condition of the associated event. Enabling conditions must always evaluate to true or false; they should not, for example, throw an exception.

```
//# system_program;
class  SW_Source{
  //# HarnessInterface harness = ...;
  SW_SourceUser sourceuser; /// Callback reference
  NetworkSocket nSocket;
  Vector      sendBuf = new Vector ();
  int         bufSize = 32*1024,
    bufUsed, ns, na, ng, sw; // = 0
  Object lock = new Object();  // lock objec
  . . .

  int moduloNSub (int a, int b){
    return (a - b) % N + ((a - b) < 0 ? N : 0;
  }

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

  void sendDataMsg(int j)  {
    synchronized(lock){
      if (!sendBuf.isEmpty() &&
          moduloNSub(j, na) < moduloNSub(ng, na) &&
          moduloNSub(j, na) ¡ sw){
            // construct data message and send it via
            // network socket.
            // Reset the timer.
            . . .
        }
      }
    }
  }

      // Thread is a class the continuously imerTask exe-
cutes method run.
  class DataSender extends Thread {
    . . .
    public void run() {
      while (true){
        //# breakpoint(...);
        synchronized(lock){
          . . .
          sendDataMsg(ns);
        }
        . . .
      }
    }
  }
```

```
// TimerTask is class that executes method run
// whenever its timer fires.
class Retransmission extends TimerTask {
  . . .
  public void run() {
    //# breakpoint(...);
    sendDataMsg(j); // retransmit j
    //# breakpoint(...);
  }
}

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

//# progress_assumption default {
//#   beginAssertion {
//#     wfair(DataSender.isAlive()) &&
//#     wfair(DataDelivery.isAlive() &&
//#     wfair(SourceReceiver.isAlive);
//#   }
//# }

}
```

Fig. 7. SeSFJava SW_Source system program (file SW_Source.java)

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

The JVM should, supposedly, ensure weak fairness for all created threads. SW_Sink system program (figure 8) is organized in a similar fashion. It has one xc event: readyToAccept(n).

Fairness assertions require special handling. Consider wfair(X), where X is a thread. A finite execution

```
//# system_program;
class  SW_Sink {
  //# HarnessInterface harness = ...;
  SW_SinkUser sinkuser;
  NetworkSocket nSocket;
  Vector recvBuf = new Vector();
  int rw, nr,
    allowedBytes = 32 * 1024;
  int RW = . . .;
  Object lock = new Object();
  . . .

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

  class ModifyWindow extends TimerTask {
    public void run (){
      //# breakpoint(...);
      synchronized(lock){
       if (rw < RW)
          rw =rw + 1;
      }
    }
  }

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

```
  class SinkReceiver extends Thread {
    . . .
    public void run() {
       while (true) {
         // receive data message with (seqNo, data)
         . . .
         //# breakpoint(...);
         synchronized(lock){
           int tmp =(seqNo − 1 − nr) % Constants.N ;
           tmp = tmp < 0 ? tmp + Constants.N : tmp;
           if (((seqNo − nr − 1) % Constants.N >= 0)
               && tmp < rw && data.length ! = 0 &&
                recvBuf.elementAt(tmp) == null) {
                recvBuf.set(tmp, data); // recvBuf[tmp] = data;
                // SendACK
                . . .
           }
           . . .
           deliverDataToUser();
         }
       }
    }
  }

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

Fig. 8.   SeSFJava SW_Sink system program (files SW_Sink.java)

σ satisfies 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 Java system function X.isAlive() returns true at the end of σ (meaning that the thread's control pointer is in the thread's run method). (See default assertions in fig. 2 and 3.)

B. SeSFJava Service Programs

The DT service program (figure 9) defines the permissible interactions between sliding window system programs (offerer of the service) and the user system (user of the service). It has different kinds of SeSF tags:

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

//# service_program;
class DT  extends UnicastRemoteObject implements DTInterface {
   // Source side variables.
   ByteArrayOutputStream srcHist = new ByteArrayOutputStream ();
   int srcBufSize     = 32 *1024, // assume that SW * message size == 32KB
      srcBufUsed;
   long srcNumSent, srcNumAcked;    = 0;

   // Sink side variables.
   long sinkNumDelivered, // = 0
      sinkBufAvail = 32 * 1024 ; // assume that RW * message size == 32KB

   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;
      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
   //# dnw_event: SW_Sink;
   public synchronized void readyToAccept(long n)  throws RemoteException {
      //# ec: true;
      sinkBufAvail = n;
   }

   //# 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;
      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 {
    //#     (srcNumAcked < sinkNumDelivered)  leadsto (srcNumAcked == sinkNumDelivered)
    //#   }
    //# }

    //# progress_obligation dataDelivered {
    //#   beginAssertion {
    //#    ((sinkNumDelivered <= srcNumSent) && (sinkkBufAvail > 0))
    //#    leadsto ((srcNumAcked == sinkNumDelivered) || (sinkBufAvail == 0)
    //#   }
    //# }

}
```

Fig. 9.   Data transfer service program (file DT.java)

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface DTInterface extends Remote {
   void sendData(byte []data)     throws RemoteException;
   void ackData(int n)           throws RemoteException;

   void readyToAccept(long n)     throws RemoteException;
   void deliverData(byte []data)  throws RemoteException;
}
```

Fig. 10. DTInterface interface (file DTInterface.java)

- Tags of the form "//# service_program" precede and identity the service program, in this case, the system program class DT.
- Each service event is preceded by a tag indicating the system of the corresponding xc event. So the tag //#dnw : SW_Source; preceding event sendData indicates that dnw event DT.sendData is mapped to xc event SW_Source.sendData, and they both have the same signature. Note that no event creates threads or processes. The signature of each service event is the same as that of the corresponding xc event.
- Tags of the form "//# progress_obligation" define the progress obligations required for DT service.

Interface DTInterface (figure 10) defines the headers of all the methods available in DT.

There is a a difference between assertion allDataAcked in SeSF (fig. 4) and the assertion allDataAcked in SeSFJava (fig. 9). We cannot apply SeSF.allDataAcked to SeSFJava.allDataAcked because we have to check for every integer value of n, which is infeasible. So, we find an assertion that models the same constraint. Since checking runtime execution is finite, SeSFJava.allDataAcked can be used instead of SeSF.allDataAcked.

C. Service And Assertion Checking Harness

To test SW_Sys against DT, we (1) create a Harness process to control the execution, (2) construct composite system $\text{SW\_Sys}^{*\prime}$ 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-$\{\text{SW\_Source}, \text{SW\_Sink}\}'$ (a version of DT-wrt-$\{\text{SW\_Source}, \text{SW\_Sink}\}$ that interacts with the harness),

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

Fig. 11. HarnessInterface interface (file HarnessInterface.java)

(3) execute $\mathsf{SW\_Sys}^{*\prime}$ along with the Harness, and (4) check whether the generated execution becomes faulty.

The harness is a process that resides on an arbitrary machine. In our example, the Harness is bound to an RMI (Remote Method Invocation in Java) port, namely “DTHarness”. The harness has interface HarnessInterface (figure 11).

The first step is to construct composite system $\mathsf{SW\_Sys}^{*\prime}$ (figure 12). Section II-D described how to get SW_Source-wrt-DT, SW_Sink-wrt-DT and DT-wrt-$\{\mathsf{SW\_Source}, \mathsf{SW\_Sink}\}$. In addition to those modification, we need these components to connect to the harness. This leads to the following modifications:

- Construct SW_Source-wrt-DT$'$, referred to as SW_Source$'$, from SW_Source-wrt-DT as follows:
  - Tags //#HarnessInterface harness $= \ldots;$ 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. Breakpoints will be explained later in this section.
- Similarly, construct SW_Sink-wrt-DT$'$, referred to as SW_Sink$'$, from SW_Sink-wrt-DT.
- Construct DT-wrt-$\{\mathsf{SW\_Source}, \mathsf{SW\_Sink}\}'$, referred to as DT$'$, from DT-wrt-$\{\mathsf{SW_Source}, \mathsf{SW_Sink}\}'$ as follows:
  - For every upw event, insert a call to method checkAssertions, and log information to log file.
  - Every dnw event is changed to a thread that repeatedly checks the enabling condition of this

dnw event, and executes its action whenever this condition holds.

- Construct SW_Sys*′ of SW_Source′, SW_Sink and DT′.

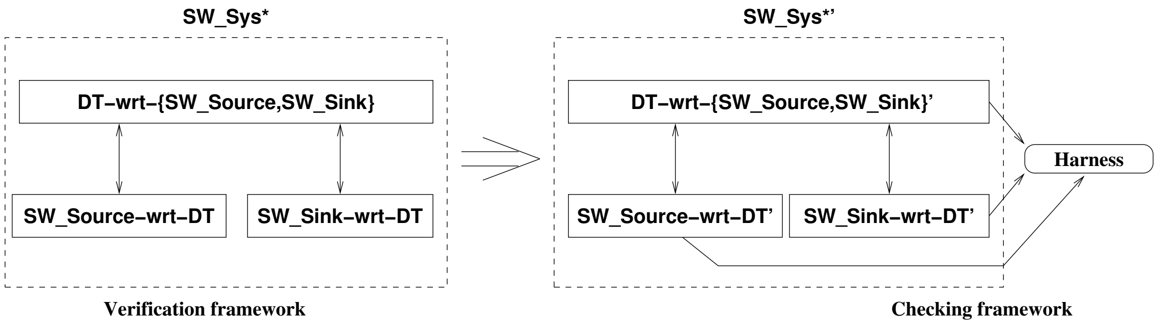


Fig. 12. SW_Sys* and SW_Sys*′ composite systems.

Once SW_Sys*′ 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 SW_Sys*′ are usually much more stringent than those of SW_Sys*.

Let *I* refer to the platform on which SW_Sys is intended to execute; that is, SW_Sys's programs involve *I*-specific constructs for IO, communication, synchronization, concurrency, and so on. Because SW_Sys′ is obtained by a simple redirection of SW_Sys's output calls, SW_Sys′ also must be executed on *I*. However, *I* invariably cannot ensure atomicity of the interactions between SW_Sys′ and other components in the system (e.g., DT′). This is because DT, and hence DT′, would, for any nontrivial service, make use of *unreasonable* atomicity. Thus *I* alone cannot serve as a testing platform.

We need to augment *I* so that SW_Sys′-DT′ 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 SW_Sys′ and DT′ such that at any time, at most one thread of SW_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 is responsible for sending its state to the SAC module.

Breakpoints are inserted manually to indicate where the thread transitions take place.

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 SW_Source satisfies assertion inv(SW_Source.sw $>=$ 0) is as follows. First, whenever SW_Sys$'$ encounters a checking location, it sends SW_Source.sw to the Harness (via method checkAssertions). Second, whenever the harness receives this field, it checks whether the predicate SW_Source.sw $>=$ 0 holds. If the predicate fails once, then the invariant does not hold.

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

1) SeSFJava Harness is started as a separate process, binds itself to RMI port “DTHarness”.
2) DT$'$ process is created, and looks up for harness’s port.
3) SW_Sys$'$ process is created. It looks up for port “DTHarness” using RMI lookup command. So, both systems are hooked up with the harness.
4) The developer can use the harness either in batch mode, leaving the harness to run for a while and then analyzing the log file, or in interactive mode, influencing the flow of the execution manually.

IV. Conclusion

The work presented has three components: (1) integrating SeSF into Java, resulting in SeSFJava; (2) developing a harness for checking systems against services and against safety and progress assertions, where systems, services, and assertions are specified in SeSFJava; and (3) applying SeSFJava harness to a data transfer protocol and service.

SeSFJava harness is able to handle general programs, general services, and general safety and progress assertions. The harness can test systems on their actual platforms. It 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. In the thread-based case, the component systems are all threads of a single process.

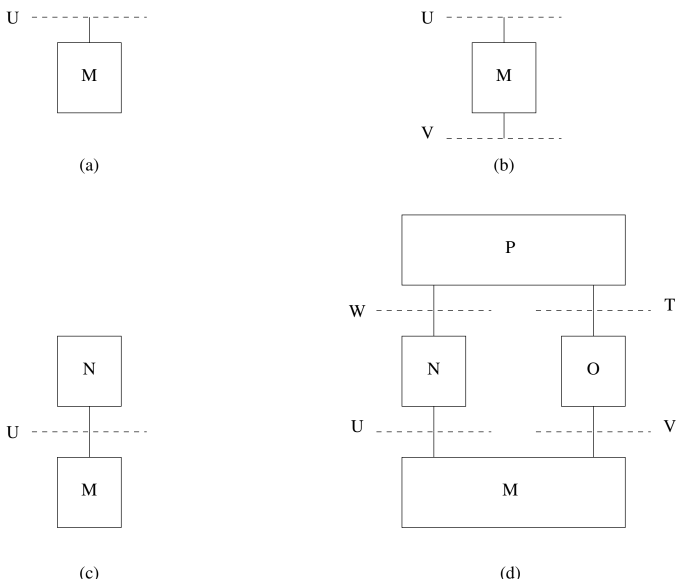


Fig. 13. Configurations

SeSFJava harness can test various configurations of systems and services: M offers U (fig. 13(a)), M offers U uses V (fig. 13(b)), a system MN with an internal service U (fig. 13(c)), or a general layered system (fig. 13(d)).

Preliminary versions of SeSFJava and the harness have been used in computer network courses to define TCP-like transport layer protocols.

REFERENCES

[1] *Specification and Description Language SDL*, CCITT ITU-T Recommendation Z. 100, International Telecommunications Union, 1993.

[2] S. Budkowski and P. Dembinski, "An introduction to estelle: A specification language for distributed systems," in *Computer Networks ISDN Systems*, vol. 14, no. 1, 1991, pp. 3–24.

[3] *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21, —, 1997.

[4] T. Bolognesi and E. Brinksma, "Introduction to the iso specification language lotos," *Computer Networks and ISDN Systems*, vol. 14, pp. 25–59, 1986.

[5] S. J. Garland and N. Lynch, "IOA: A language for specifying programming and validating distributed systems," December 2000. [Online]. Available: http://theory.lcs.mit.edu/tds/papers/Garland/ioaManual.ps.gz

[6] R. Cleaveland, J. Parrow, and B. Steffen, "The concurrency workbench: A semantics-based tool for the verification of concurrent systems," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 1, pp. 36–72, January 1993. [Online]. Available: citeseer.nj.nec.com/article/cleaveland94concurrency.html

[7] R. Cleaveland, J. Gada, P. Lewis, S. Smolka, O. Sokolsky, and S. Zhang, “The Concurrency Factory - practical tools for specification,” 1994. [Online]. Available: citeseer.nj.nec.com/cleaveland94concurrency.html

[8] J. Fernandez, C. Jard, T. Jéron, and C. Viho, “An experimental in automatic generation of test suites for protocols with verification technology,” *Science of Computer Programming - Special Issue on COST247, Verification and Vaidation Methods for Formal Descriptions*, vol. 29, no. 1-2, pp. 123–146, 1997.

[9] G. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, May 1997.

[10] J. Rushby, “Specification, proof checking, and model checking for protocols and distributed systems with PVS,” Tutorial presented at {FORTE X/PSTV XVII ’97}, Menlo Park, CA, November 1997. [Online]. Available: http://www.csl.sri.com/papers/forte97/

[11] S. Schneider, “Abstraction and testing,” in *FM’99, Vol. I, LNCS 1708.* Springer-Verlag Berlin Heidelberg, 1999, pp. 738–757.

[12] S. Bensalem, V. Genesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rues, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari, “An overview of SAL,” in *Fifth NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000.

[13] *SandiaXTP – An Object-oriented Implementation of XTP 4.0*, Derieved from the Meta-Transport Library: User Manual Release 1.4, Sandia National Laboratories, 1995.

[14] R. Sijelmassi and B. Strausser, “The pet and dingo tools for deriving distributed implementations from Estelle,” *Computer Networks and ISDN Systems*, vol. 25, pp. 841–851, 1993.

[15] J. Thees and R. Golzhein, “The eXperimental Estelle compiler - automatic generation of implementations from formal specifications,” in *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, Clearwater Beach, Florida, March 1998.

[16] 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*, vol. 7, no. 11, 2001.

[17] J. Tauber, “Verifiable code genreration from abstract I/O automata models for distributed computing,” Ph.D. dissertation, Massachusetts Institute of Technology, March 2001.

[18] D. Hansel, R. Cleaveland, and S. A. Smolka, “Distributed prototyping from validated specifications,” *12th International Workshop on Rapid System Prototyping*, June 2001.

[19] A. Prinz and M. Lwis, “Generating a compiler for SDL from the formal language definition,” in *Lecture Notes in Computer Science*, vol. 2708. Springer-Verlag Heidelberg, January 2003, pp. 150–165.

[20] J. Thees, “Protocol implementation with estelle – from prototypes to efficient implementations,” 1998. [Online]. Available: citeseer.nj.nec.com/thees98protocol.html

[21] A. U. Shankar, *Concurrent Systems and Services: Design, Verification and Testing.* under constrction, 2002.

[22] S. S. Lam and A. Shankar, “A theory of interfaces and modules i-composition theorem,” *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 55–71, January 1994.

[23] K. Chandy and J. Misra, *A Foundation of Parallel Program Design.* Addison-Wesley, Reading, M.A., 1988.

[24] J. Misra, *A Discipline of Multiprogramming.* Springer-Verlag, 2001.

[25] L. Lamport, “The temporal logic of actions,” DEC SRC Report 57, Tech. Rep., 1991, april 1990, Revised April 1991.

[26] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[27] C. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985.

[28] A. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[29] Z. Manna and A. Pnueli, “Adequate proof principles for invariance and liveness properties of concurrent programs,” *Science of Computer Programming*, vol. 4, pp. 257–289, 1984.

[30] ——, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[31] R. Back and R. Kurki-Suonio, “Distributed cooperation with action systems,” *ACM Trans. on Prog. Lang. and Syst.*, vol. 10, no. 4, pp. 513–554, October 1988.

[32] N. Lynch and M. Tuttle, “Hierarchical correctness proofs for distributed algorithms,” in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., August 1987.

[33] S. S. Lam and A. Shankar, “A relational notation for state transition systems,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 755–775, July 1990.

[34] W. Pugh, “The Java memory model is fatally flawed,” *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 445–455, 2000. [Online]. Available: citeseer.nj.nec.com/pugh00java.html

[35] J. Manson and W. Pugh, “The Java memory model simulator,” in *Workshop on Formal Techniques for Java-like Programs, in Association with ECOOP*, June 2002.

[36] T. Elsharnouby. (2003) SeSFJava and SeSFJava harness. [Online]. Available: http://www.cs.umd.edu/~sharno/SeSFJava/

[37] ——. (2003) Class homepage on computer networks (CMSC417) at University of Maryland. [Online]. Available: http://www.cs.umd.edu/class/spring2003/cmsc417/

[38] Y. Yang. (2003) Class homepage on computer networks (CS433) at Yale University. [Online]. Available: http://zoo.cs.yale.edu/classes/cs433/assignments/prog1/index.html

[39] A. U. Shankar, “Verified data transfer protocols with variable flow control,” *ACM Transactions on Computer Systems*, vol. 7, no. 3, pp. 281–316, August 1989.