Developing a Framework for Simulation, Verification and Testing of SDL Specifications*

Olga Shumsky and Lawrence J. Henschen

Department of Electrical and Computer Engineering
Northwestern University, Evanston, IL 60208
[shumsky,henschen]@ece.nwu.edu

Abstract. This paper describes a simulator for SDL, a formal description technique for distributed, concurrent, communicating systems. The simulator consists of three main components: the translator, the activator, and the process execution and interleaving mechanism. All components are written in a subset of Common Lisp, and their desired properties are proved in ACL2. The simulator is intended as a basis for future verification and testing of SDL specifications.

1 Introduction and Overview

Formal methods and verification have found their way into industrial hardware design. Formal methods are less often applied to software design, because for the most part correcting an error late in the development cycle costs much less than correcting a comparable error in hardware at comparatively late stages of production. However, there are always safety-critical software systems where reliability is a key concern. A number of formal methods techniques have been developed to aid in the design of such systems. However, these formal methods usually fall short of complete verification of the software design.

Therefore we would like to provide a mechanism to conduct complete formal verification of designs specified using existing formal methods. We focus our attention on the formal language SDL, but the approach need not be limited to a particular formal language. We use ACL2[4] to implement a simulator for the formal language, and we are proving, both formally and informally, correctness of the simulator.

The simulator is intended to serve as a basis for complete verification of specifications. We assume that the desired properties of the system described in SDL are known and can be stated in terms understandable to ACL2. We can use the theorem-proving component of ACL2 to prove that these properties hold; if these proof attempts fail, we can then reexamine the specification for possible errors. Once all desired properties of the specification are proved, the specification, together with the simulator, provide an environment for testing of the implementation. Simulation of the specifications on test cases provides correct results to these test scenarios. The simulator can also serve as a test driver: portions of compiled code can replace appropriate chunks of the specification. The hybrid system can then be simulated and its behavior compared to the behavior of specification simulation. Any discrepancies pinpoint errors in the implementation.

The current stage of the project is concerned with implementing the simulator for SDL specifications and proving the simulator correct.

2 Brief description of SDL

Specification and description language (SDL)[3, 7] is a language for specifying distributed concurrent asynchronous communicating systems and their interaction with the environment. The language was originally developed for protocol specification but has since been used on a variety of systems. The main components of SDL are communicating processes, which are represented by extended finite-state machines and grouped into blocks. A transition in response to a signal is represented within the finite-state machine by a number of actions such as updating an internal variable, calling a procedure, creating a new process, sending a signal, setting a timer, and so on. Processes communicate with each other and with the environment outside of the enclosing block via instantaneous communication links called signal routes. Blocks communicate with each other via non-deterministically delaying communication links called channels.

* This project is supported by the Northwestern/Motorola Center for Telecommunications Research.

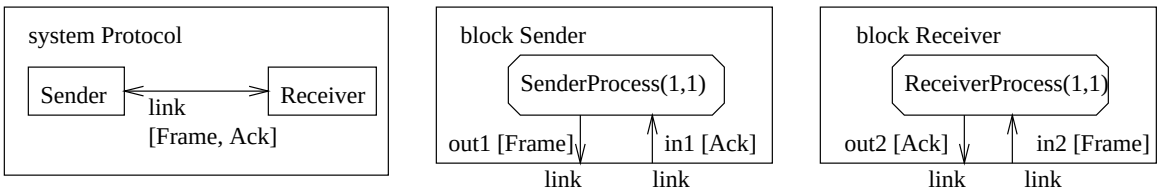

Fig. 1. System Definition in SDL

Figures 1 and 2 show a small example of system specification with SDL. The system is an implementation of a simple communication protocol where the sender transmits numbered messages and awaits an acknowledgment from the receiver. If the acknowledgment does not arrive within a certain time period, the message is resent. Figure 1 is the high-level description of the protocol: the protocol system consists of two blocks `sender` and `receiver` which communicate via a bidirectional channel `link` that is capable of transmitting two types of signals: message frames and acknowledgments. Each block contains a process and signal routes for outgoing and incoming messages. The two numbers in parenthesis indicate the initial number of process instances and the maximum number of process instances at any point during the lifetime of the system. In the example, only one instance of each process exists from the start and throughout the lifetime of the system. The descriptions of the sender and receiver processes are presented in figure 2.

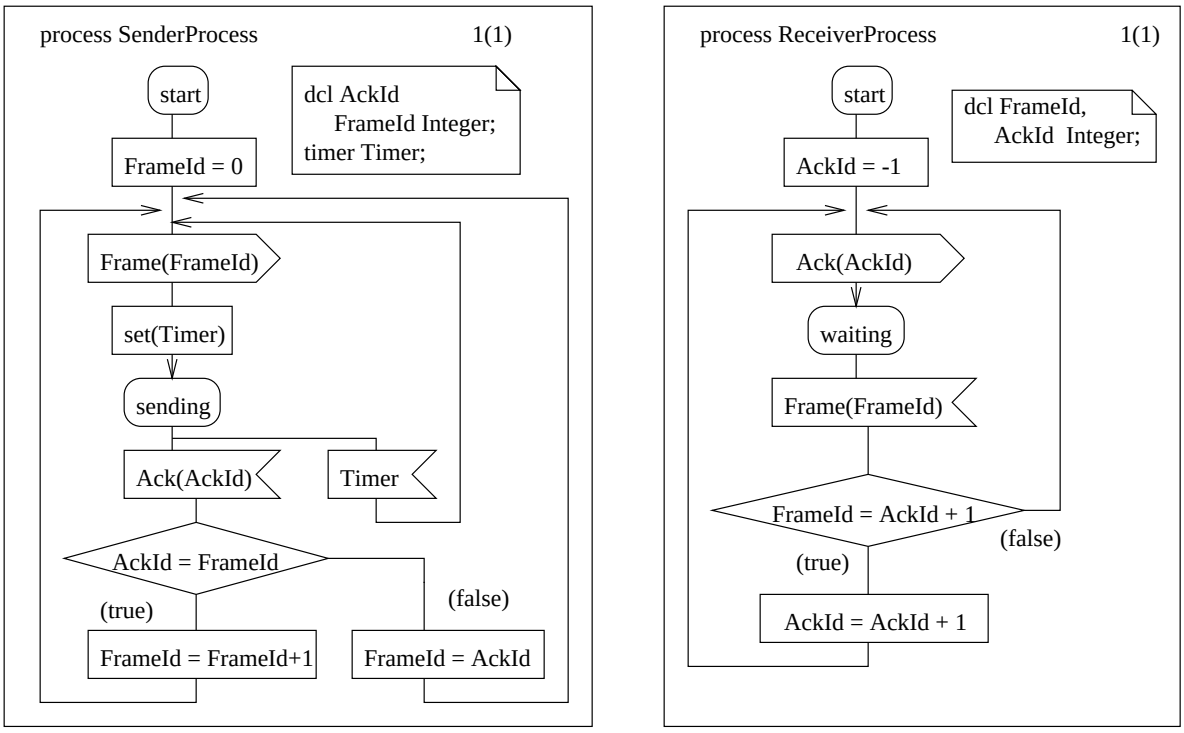

Fig. 2. Definition of Sender and Receiver Processes in SDL

Each process contains a start state, which is an initializing list of actions. In our example, the sender is initialized by setting the id of the next message to 0, sending that message and setting the timer after the message is sent. The main part of the sender process consists of a single state `sending` and a set of actions in response to two possible input signals: an acknowledgment from the receiver or an expired timer. Should any other type of signal be received by the process, the signal will be consumed with no consequences. If an acknowledgment is received, the next message is sent and the timer is reset. If the timeout signal is received,

the last message is resent. The initialization of the receiver consists of setting the id of the last received message and acknowledging that message. Since no messages have been received yet, the id is set to -1. The receiver process consists of one state `waiting` for which only one possible input, a message from the sender, is defined. If the received message has the next expected sequence number, the acknowledgment for the message is sent. Otherwise, the acknowledgment for the last successfully received message is sent.

This rudimentary example shows a simple communication protocol with the window size of 1. However, unlike a real protocol specification, the description excludes the model of the communication media, omitted for lack of space. Messages traverse routes that are guaranteed not to lose any data, as specified by SDL. Therefore, any error-correcting behavior of the protocol, as defined, may occur only in response to a timeout in the sender.

3 System Simulation

Three components are needed to simulate an SDL specification. First, the specification has to be translated into the format appropriate for processing using Lisp functions. Second, an instance of the system has to be created and its internals initialized, and finally, the individual process executions have to be interleaved to simulate concurrency. The main three components are discussed below.

3.1 Translation and Internal Format

We have defined an internal format into which SDL specifications are translated. Differences between text-based SDL and our format for process specifications are purely syntactic. We store every entity as a list for easy processing: a process is a list of states, a state is a list of transitions, etc. The receiver process in the example in figure 2 is stored as follows:

```
(receiver (1 . 1) (ackid frameid)
          (start (() (task ackid -1)
                     (label 1)
                     (output ack (ackid) () ())
                     (nextstate waiting)))
          (waiting ((frameid (frameid)) (decision ((= frameid (+ ackid 1))
                                                   (task ackid (+ ackid 1))
                                                   (join 1))
                                                  ((<> frameid (+ ack 1))
                                                   (join 1))))))
```

We would like to have some assurances that the translation, superficial as it is, preserves the meaning of the specification. An approach we are exploring is to define an inverse function `untranslate` and prove the following theorem:

```
(implies (wf-specs S) (equivalent (untranslate (translate S)) S))
```

where `equivalent` is a special-purpose equality recognizer. This approach only guarantees that the translation and the un-translation mechanisms are inverse functions with respect to our definition of equivalence. We are still looking for other ways to express that the translation mechanism is correct.

The communication network is also translated, but the differences between our format and SDL are quite significant. In SDL a path between two processes may consists of several links. In our format these occupy a single position in a list and represent each path as a multi-component single entity. Instantaneous paths are stored as tuples `(sender receiver route-name)`. Delaying paths are stored as tuples `(sender receiver (member links) (signal queue))`. For example, the network from figure 1 is represented as `((sender receiver (out1 link in2) nil) (receiver sender (out2 link in1) nil))`.

The effect of storing the network in such a way is essentially flattening the hierarchy of SDL blocks. In figure 3(a) we show the hierarchy (solid lines) of the SDL specification of the system in figure 1(b) and the communication links (dashed lines). In figure 3(b) we show the flattened representation of the network that corresponds to our internal storage format.

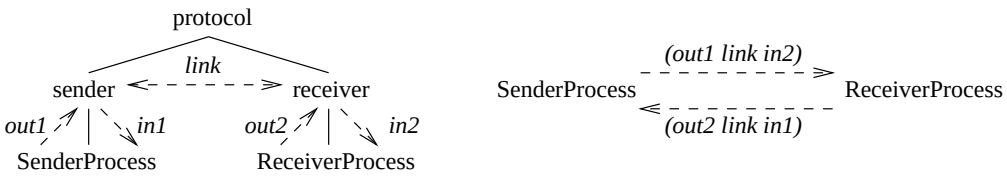

Fig. 3. Network: (a) SDL representation and hierarchy; (b) internal representation.

Demonstrating equivalence of these two representations in not easily accomplished because our notation eliminates the names of intermediate blocks. For example, the list `(out1 link in2)` gives no indication that signals pass through the `sender` block on their way from `SenderProcess` to `ReceiverProcess`. We are carrying on informal proofs that we handle the communication network in accordance with the SDL standard, that is that compressing the multiple-link paths between processes into single objects does not have undesirable side effects, such as affecting the order of signal delivery. We are also developing a special version of the equivalence checker that can be used with the untranslate function to prove the same result within ACL2.

3.2 Activator

SDL semantics differentiates between a system (process) type, which is a static description, and a system (process) instance, which is created when the system (process) is invoked. Therefore, we need a mechanism to create instances from the static specification. We defined an activator function which generates the appropriate number of instances of each process type. The activator mechanism initializes process ids, variables and signal queues, and puts each instance into the start state. An active process is a tuple `(id type state memory definition queue)`. Upon activation, the receiver process from the running example is in the form shown below. Definition of the state machine is omitted since it is identical to that of the receiver process definition above. The local variables `ackid` and `frameid` are uninitialized. The memory also includes implicit SDL variables `sender`, `self`, `parent`, `offspring`, which record the obvious process ids. Upon instantiation only `self` and `parent` are initialized. The environment, whose process id is 0, serves as the parent of any process created at system invocation.

```
(1 receiverprocess start
 ((ackid . nil) (frameid . nil) (self . 1) (sender . nil) (parent . 0) (offspring . nil))
 ((start ....)) nil).
```

Curiously, SDL does not have a notion of communication path instances. Therefore the communication network is a static entity, and the activator only resets to empty the signal queues on the delaying paths.

The activator mechanism is correct if it generates an appropriate instance of the original system. We defined a recognizer `(wf-type S)` for system types that are well-formed and can be instantiated. The recognizer checks that process types are unique, that initial process counts are in agreement with maximum process counts, and that each process has a defined start state. We also defined a recognizer `(wf-instance A S)` that checks that A is an instance of S. The recognizer checks that all processes in A have a definition in S, that the number of instances of any process type in A does not violate the maximum instance count in S, and that active processes in A have unique process ids. We proved that the activator mechanism is correct:

```
(defthm activate-makes-instance
  (implies (wf-type s) (wf-instance (activate s) s))).
```

3.3 Process simulation, interleaving, and network handling

System simulation is based on simulation of individual process instances, handling of the signals that travel through the communication network between process instances, and process instance interleaving to simulate concurrency.

action	state	memory	queue
after instantiation	start	(ackid . nil) (frameid . nil) (sender . nil)	nil
first initialization	waiting	(ackid . 0) (frameid . nil) (sender . nil)	nil
signal arrives in queue	waiting	(ackid . 0) (frameid . nil) (sender . nil)	frame(1)
signal consumed	waiting	(ackid . 0) (frameid . 1) (sender . 1)	nil
transition completed	waiting	(ackid . 1) (frameid . 1) (sender . 1)	nil

Table 1. Receiver process simulation

Table 1 shows a few steps of the receiver process execution. We only show the relevant portions of the process instance, namely the memory space, state, and the input queue of signals. Assume that process ids assigned to sender and receiver instances are 1 and 2 respectively.

Our aim is to build a syntacticly and semantically correct simulator for SDL specifications. We demonstrate the syntactic correctness by showing that the property `(wf-instance S D)` is preserved after each process and system simulation step. As of now, we do not demonstrate the semantic correctness of the simulator formally. However, we discuss informally the semantics of the SDL entities that appear in the finite-state machines that represent processes and the implementation of these semantics in our simulator. Table 2 presents syntactic definitions of the entities that appear in transitions.

input signal_name (formal parameters)	set(timer), reset(timer)
call procedure_name (formal parameters)	task variable expression
create process_type (formal parameters)	nextstate state_name
output signal_name (formal parameters) (to process) (via path)	join label
decision ((expression (list of actions)) (expression (list of actions)) ...)	stop

Table 2. Syntactic Definitions

Every transition begins by consuming a signal from the input queue. The result of signal consumption, as shown in the example, is to update the variables specified in the formal parameters of the signal by the actual signal parameters. In addition the implicit variable `sender` is set to the id of the process that sent the message. The transition itself may contain any number of the following actions: task, output, procedure call, process creation, timer operations, decision, and join statements. The transition is terminated by the `nextstate` directive or a `stop` directive. We implement the nextstate directive by updating the state field of the process instance with the specified state name. We also make sure that the directive was indeed the last entity in the transition. If additional entities are present, a warning is generated and the entities are ignored. The `stop` directive specifies that the life of the process instance is over. The instance is removed from the list of active processes.

All other directives except for the join statement depend on an evaluation function, which we defined to compute a value based on the values of the local variables in the memory of the process instance. The task directive specifies that the value of a given variable must be updated to the value of a given expression. The output directive indicates that a message must be sent. The address of the message may be implicit or explicit. The id of the receiving process can be stated in the optional `to` field of the output directive. An optional `via` field specifies the communication path to be taken by the message. The address of the message is implicit when the types of messages that the communication paths may carry define which path a message can traverse. In our example, signals are implicitly addressed. The equivalent explicit addressing for the output message in the sender is `(output frame (frameid) (to receiver) (via out1))`. The message also carries an implicit parameter that records the id of the sending process. When the choice of the communication path is not unique, we use the following guidelines for route selection: instantaneous paths are given preference over the delaying paths, shorter paths are preferable to longer paths; in all other cases the path is selected randomly. The address of the signal may be ambiguously defined when the optional `via` and `to` directives do not agree with each other or with the types of signals the specified links may carry. In this case, a warning is generated and the signal is discarded. If a signal is sent through an instantaneous path, it is immediately placed in the input queue of the receiving process. If a signal travels on a delaying path, it is first placed in the

signal queue of the path. We implement the nondeterministically delaying nature of the paths by supplying an oracle to decide whether a signal has to be forwarded to the input queue of the receiving process.

An active process may create a child instance using the `create` directive. The evaluation function is used to compute actual parameters, which are stored in the memory of the newly created process. The parent id is recorded in the new process, and the child id is recorded in the creating process. The new process is put in the start state. The decision directive represents a branch in the flow of the transition. The directive is processed like the `cond` of LISP. Finally the join statement is implemented by searching the process definition for the given label and resuming the processing of directives that follow the label. Implementation of the timer operations and of the procedure calls is ongoing.

To approximate the concurrent execution of active processes in the system, we use process interleaving based on a supplied oracle, as suggested in [5], where the oracle gives the id of the next process to execute. The communication network is examined at regular intervals to propagate signals queued up in the delaying paths. How often the network should be examined is still under investigation. At this point in the implementation we examine the network after every process execution step.

Also under investigation is the granularity of the process execution step. Consider the transition in response to a timeout signal in the sender process from our protocol example:

```
input Timer;
  output Frame(FrameId);
  set(Timer);
  nextstate sending;
```

SDL does not address atomicity of transitions or actions. We are therefore left with a number of options. One is to make the entire transition atomic, that is during one step of the process execution when the timer signal is received, all three actions (output, set, nextstate) are performed. However, as pointed out by the referee, this approach might prevent the simulation from mimicking some possible real-life interleavings of the processes. Another option is to make the action atomic, so that the above transition would take three steps. For some applications, this approach might be too fine-grained. We are implementing a simulator function for both scenarios and will allow the user to substitute either depending on the nature of the simulated system.

3.4 Modeling Time and Timer Operations

Modeling time appropriately is important for correctly handling timer operations, such as calculating timer expiration. Time is external in SDL, that is the standard does not concern itself with how time progresses. The standard specifies that an action may take an indeterminate time to complete, and that a transition may take an indeterminate time to fire even if a signal is available for consumption. Different SDL simulators approach time modeling in different ways[2]. We follow the approach from the IF project[1]. The approach relies on a simplifying assumption that transitions fire immediately if a signal is available for consumption. It is also assumed that the communication network is much slower than the processor. These assumptions then lead to the conclusion that, from the point of view of the process, time progresses only when the process is waiting for a signal to arrive at its input queue.

The implementation of this approach in our simulator would amount to maintaining a clock tick counter in every instance. The counter would be incremented when the oracle points to the process to simulate execution but there are no signals in the local input queue. Timer operations would depend on the local tick counter. At the beginning of each transition active timers would be compared to the local tick counter to determine if any timers have expired. If so, a timeout signal, which is a priority signal, would be generated and placed on top of the signal input queue.

4 Using the Simulator for Verification and Testing

Our goal is not only to build a simulator of SDL specifications but also to create an integrated framework for main stages of software system development. One such stage is verification in which correctness properties of the system are proved. Our example protocol is correct if the sender and the receiver have

a coinciding opinion about the id of the last successfully transmitted frame, as recorded in the local variables `AckId` and `FrameId` of the receiver and sender, respectfully. The sender and the receiver agree when the values of these variables are equal, or when `FrameId` is greater than `AckId` by at most 1. The latter case corresponds to the situation when the sender has already dispatched a frame, but the message is still traveling through the network and has not yet reached the receiver. We provide a set of access functions to reference items, such as variable values and process instance states, in a simulation. For example, function `(instance process-type system-instance)` extracts instances of the given type, while function `(variable-value var-name process-instance)` examines the value of the specified variable in a given instance. Using these functions we can state the desired protocol correctness condition as the following ACL2 theorems:

```
(defthm sender-receiver-agree-1
  (<= (variable-value 'ackid (instance 'receiver (simulate S oracle)))
      (variable-value 'frameid (instance 'sender (simulate S oracle)))))

(defthm sender-receiver-agree-2
  (let ((v1 (variable-value 'ackid (instance 'receiver (simulate S oracle))))
        (v2 (variable-value 'frameid (instance 'sender (simulate S oracle)))))
    (implies (< v1 v2) (= (+ 1 v1) v2)))).
```

Those are the kinds of correctness properties that we intend to prove about systems defined in SDL. We are still building the functionality to be able to attempt these proofs.

The simulator is also intended to be used as a testing tool. Suppose for example that the receiver has been implemented in the target language and compiled. We can test the compiled code by replacing the simulation of the receiver by a call to the code. The rest of the system (the sender) is simulated by the ACL2 code described above. The behavior of the hybrid system system (the signals and the sequence of variable values of the sender process) can be compared to the behavior of the pure simulation on the same input. Since, by that point, the simulator's correctness is established, and the confidence in the correctness of the SDL specification is high due to proved properties, any discrepancies in the results point to possible errors in the complied code.

5 Other Approaches to Verification of SDL Specifications

Other approaches to verification of SDL specifications are based on model checking. For example, in the IF[1] project at Verimag, SDL specifications undergo several translations. The resulting PROMELA representation is used with the SPIN model checker. Another approach relied on using a BDD-based symbolic model checker, which is a part of the SVE (System Verification System) tool developed at Siemens[6].

References

1. M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: A validation environment for timed asynchronous systems. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of CAV'2000*, Lecture Notes in Computer Science 1855, pages 543–547. Springer Verlag, July 2000.
2. M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for real-time: What is missing? In *Proceedings of SAM2000, Grenoble Col de Porte, France, http://www-verimag.imag.fr/~graf/SAM2000*, June 2000.
3. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1997.
4. M. Kaufmann and J S. Moore. ACL2: A Computational Logic for Applicative Common Lisp. http://www.cs.utexas.edu/users/moore/acl2, 2000.
5. J S. Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. http://www.cs.utexas.edu/users/moore/publications/acl2-papers, February 1998.
6. Franz Regensburger and Aenne Barnard. Formal verification of SDL systems at the Siemens mobile phone department. In Bernhard Steffen, editor, *Proceedings of TACAS'1998*, Lecture Notes in Computer Science 1384, pages 439–455. Springer Verlag, 1998.
7. K. J. Turner, editor. *Using Formal Description Techniques: An introduction to Estelle, Lotos and SDL.* John Wiley and Sons, Chichester, 1992.