Bisimulation Analysis of SDL-Expressed Protocols: A Case Study

Marsha Chechik and Hai Wang

Department of Computer Science
University of Toronto

Abstract

Faster, better networks algorithms are often being discovered, and it is desirable to be able to replace an old algorithm by a new in a manner that is completely transparent to the application using it. This paper investigates the technique for ensuring such transparency for protocols expressed in SDL, via bisimulation checking. We discuss the main issues involved in translating SDL into Concurrency Workbench, a tool for performing bisimulation checking, and illustrate the feasibility of the technique by comparing the SDL specification of the Go-Back$N$ protocol with the family of new protocols, called Asynchronous Retransmission Go-Back-N (AR). The latter perform better in environments characterized by high error rates and/or large propagation delays.

1 Introduction

As computer-communication systems become more complex, it is becoming more and more important to ensure the correctness of protocols they rely on. In addition, faster, better algorithms are often discovered, and it is desirable to be able to replace an old algorithm by a new in a manner that is completely transparent to the application using it, i.e., the algorithms are identical from the application's point of view.

In this paper we concern ourselves with checking equivalence of protocols expressed in the Specification and Description Language (SDL), which was first standardized by CCITT/ITU-T [20] in 1976 as Recommendation Z.100. SDL has been widely used in the telecommunications field, mostly for specifying protocols [1, 2, 11].

Typically, designers would like to establish that protocols are free of logical errors, e.g., deadlock, and are semantically correct, i.e., the protocols behave exactly as the specifier intended [27]. Several approaches have been proposed to automatically detect both logic and semantic errors in SDL specifications. The idea is to translate SDL into temporal logic or process algebra for verification. The approaches based on temporal logic [18, 16, 27, 12, 4] perform model checking in which SDL specifications are validated against various correctness properties, such as deadlock freedom and the ability to perform certain sequences of actions. The approaches based on process algebra [15, 26, 28, 3] enable equivalence checking – checking that SDL specifications and their expected behaviors are the same with respect to some notion of equivalence.

Checking equivalence, or refinement, is a common problem in formal software design. Some research on SDL [17] advocates a "forward" approach to developing protocols – requirements are specified as a set of linear-time logic properties, and the model is analyzed for conformance with these properties. Model refinement is done in small controlled steps; manual inspection is used to ensure that the refinement is correct, and then the resulting model is again analyzed for conformance with the requirements properties. However, in our case, when two versions of protocols are already developed, this approach is not effective – fully characterizing the detailed behavior of a protocol using temporal logic is usually infeasible. Thus, we needed to look into the bisimulation method based on process algebras to complete the task at hand.

Existing process-algebraic approaches support translation of only a subset of SDL's control structures, usually abstracting all of its data. In addition, complex data structures, such as queues and stacks, and procedure calls are also not supported. The reason is that the process-algebraic approach does not explicitly support data. Values of variables have to be encoded into a state, which leads to a state-space explosion, but abstracting from data completely is unrealistic and leads to overly-simplistic models. In this paper we describe how to translate SDL models into Concurrency Workbench (CWB) [8, 25] specifications. CWB is an automated tool for analyzing networks of finite-state processes expressed in Milner's *Calculus of Communicating Systems* (CCS) [23]. It supports a variety of verification methods, including model checking [7], equivalence checking, and preorder checking [8]. CWB has been successfully applied to verify many communication protocols such as the Alternating Bit [8] and the CSMA/CD protocols [26]. Although an effective verification tool, CWB is at best an adequate modeling tool. Models are specified at a very low level of abstraction, making them hard to build and hard to read and review. Therefore, we are interested in automated translation of SDL-expressed models into CWB.

In this paper we show the feasibility of using CWB for verification of SDL-specified protocols. We illustrate our technique by a case study involving verifying several sliding window protocols: Go-Back-N and AR, expressed in SDL. The former is the standard protocol, whereas AR, Asynchronous Retransmission Go-Back-N, is a family of new protocols which perform better than Go-Back-N in environments characterized by high error rates and/or large propagation delays. Our goal is to establish that the AR algorithms are equivalent to the Go-Back-N from the application's point of view. The remainder of the paper is organized as follows: Sections 2 and 3 describe the Go-Back-N and the AR protocols, and their modeling in SDL, respectively. Section 4 presents techniques to translate SDL into CCS. Section 5 presents results of translation and verification of the Go-Back-N and the AR protocols. In Section 6, we outline directions for future work and conclude the paper.

2 Protocol Description

In this section we describe the Go-Back-N protocol. We also motivate and propose a family of new protocols, termed Asynchronous Retransmission Go-Back-N (AR).

A major concern of computer-communication systems in which data transmission channels are not ideally error-free is the need to control transmission errors caused by the unreliable channels so that virtually error-free data transmission can be attained. One of the most common approaches to solving this problem is through the automatic repeat request [21], exemplified by the Go-Back-N protocol.

2.1 Go-Back-N Protocol

In the Go-Back-N protocol, the sender continuously transmits packets in order and stores them pending acknowledgment until N unacknowledged packets have been transmitted or a negative acknowledgment (NAK) has been received. Each packet contains a unique identifier. The receiver returns a positive acknowledgment (ACK) together with the identifier of the corresponding packet for each correctly received packet until it detects an erroneous packet. Once the receiver detects an erroneous packet, it returns a NAK together with the identifier of the first incorrectly received packet, and discards the erroneous packet and all subsequently transmitted packets regardless of whether or not they are error-free. A certain time period after the transmission of a packet, either the sender receives the corresponding ACK/NAK or a timeout occurs. If the sender receives an ACK, it removes the packet from the retransmission list. If the sender receives a NAK or a timer on a packet expires, it retransmits this packet and all packets sent after it. Retransmission continues until the first erroneous packet is positively acknowledged. Afterwards, the sender proceeds to transmit new packets.

2.2 AR Protocols

The Go-Back-N protocol is nearly optimal for channels characterized by low error rates and small propagation delays, and thus is widely used for error control in many classical computer-communication networks. However, it is not efficient for modern non-conventional channels with high error rates and/or large propagation delays, such as mobile and satellite links. Under these conditions, there are many time intervals during which the channel is idle while the sender has at least one packet waiting for an acknowledgment. A number of protocols have been developed to achieve better performance in this type of environment, e.g. Towsley's Stutter Go-Back-N protocol, which has been proven to achieve a better performance when error rates are high and/or propagation delays are large [30]. In this protocol, the sender repeatedly transmits the last unacknowledged packet during the period of time the channel would otherwise be idle.

Based on the design philosophy of the Stutter Go-Back-N protocol, we introduce a family of new protocols, termed Asynchronous Retransmission Go-Back-N (AR) protocols. Like the Stutter Go-Back-N protocol, all AR Protocols try to improve performance by utilizing these idle periods. Let p be the mean error rate and let d be the mean propagation delay during a given time interval. Also, let r be the mean data rate during that time interval. We define

$$K = F(p, d, r) \tag{1}$$

where F is a mapping from $\Re^3$ to a set of all integers between 1 and N inclusive. Generally, an AR protocol is the same as the basic Go-Back-N protocol except that it repeatedly retransmits the last K unacknowledged packets, if any, during the time that the channel is idle. One design criterion for the function F is that the total transmission time of K packets should be less than the mean propagation delay[1]. Depending on how F is defined, we have a family of protocols. In fact, the basic Go-Back-N protocol can be treated as an AR protocol with K equal to 0. When K is equal to 1, the AR protocol is the Stutter Go-Back-N protocol. If K is a constant, then we obtain a deterministic non-adaptive protocol. If K is a variable that depends on a deterministic function F, then we have a deterministic adaptive protocol. If K depends on a non-deterministic function F, we obtain a random protocol.

[1] Otherwise, the transmission takes too much time, and the protocol no longer yields any performance savings.

3 Modeling Protocols in SDL

SDL, a formal specification language able to describe functions of an application in an implementation-independent way, has been widely used in the telecommunications industry. In SDL, an application is modeled as a *system* consisting of *processes* which communicate with each other and the environment by sending and receiving *signals* via *channels*. Each process is regarded as a finite state machine. A signal serves both as a primitive for synchronization and as a vehicle for exchanging information between processes. Local variables and procedures can be used to describe the internal behavior of a process. Like any programming languages, SDL supports both simple and complex data types. A complete description of SDL can be found in [20].

We modeled the Go-Back-N and the AR protocols using an integrated SDL toolset, SDT, by Telelogic [29]. During the modeling, we abstracted away from message data, keeping track only of the sequence number of the message. Moreover, since the number of distinct packets should be at least $N + 1$ [13], their sequence numbers can be represented by an integer between 0 and N, inclusive. Thus, the sender needs to use $N + 1$ timers. Additionally, we have to keep track of a queue of unacknowledged packets for the sender. Infinite SDL channels can not be used for this purpose, so we define an abstract data type of a bounded queue of size $N+1$ instead. Finite channels without a delay, as described in Section 4 below, can work equally well.

3.1 Modeling the Go-Back-N Protocol

The sender has two states, DataTransfer and Retransmission, and two local variables, Vs and Va, keeping track of the sequence number of a packet to be sent next, and the number of unacknowledged packets, respectively. It also maintains a queue of sent packets. When the sender receives a sig-

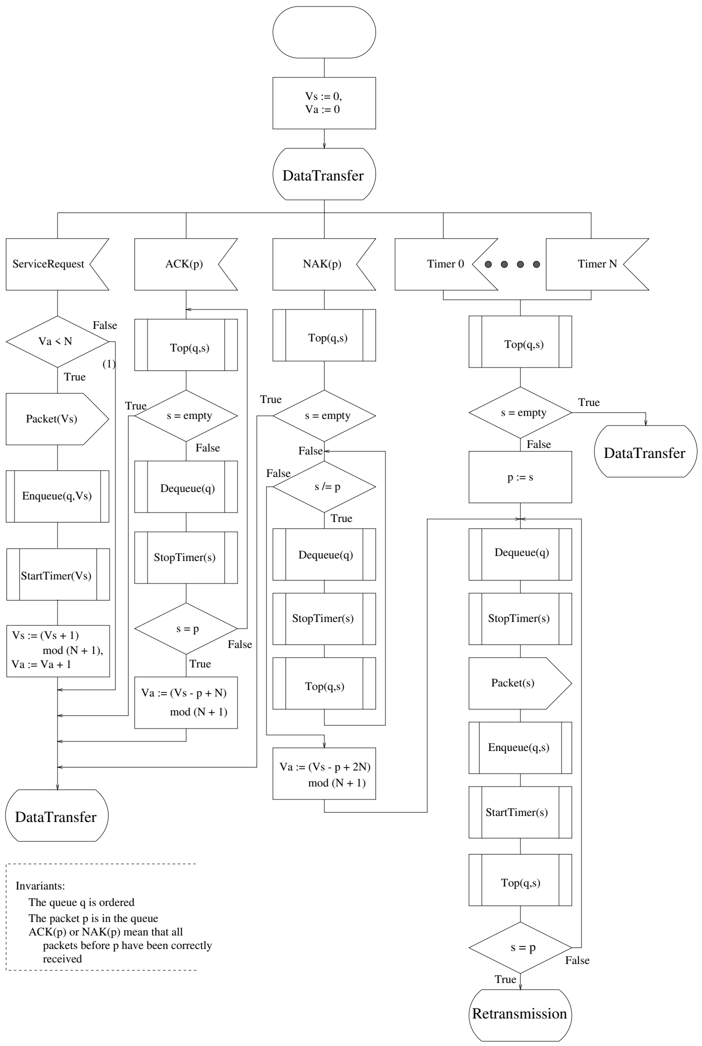

Figure 1: The SDL specification for the state DataTransfer of the sender.

nal ServiceRequest (from the application) and the number of unacknowledged packets is less than the size of the window (this corresponds to the window being open), it sends the packet to the receiver, activating the corresponding timer, putting the packet into the queue, and updating Va and Vs accordingly. Otherwise, it ignores the request.

When the sender receives a positive acknowledgment (ACK) or a negative acknowledgment (NAK) on the packet with sequence number p, it assumes that all packets with sequence numbers less than p have been correctly received. In both cases, the sender removes all packets before p from the queue and stops the timers associated with them. In the case of an ACK, it does the same with packet p. Additionally, it updates the number of unacknowledged packets. In the case of a NAK, the sender retransmits packet p and all packets following it in the queue, resets the timers on these packets and goes into the Retransmission state.

The retransmission queue allows the sender to maintain the invariant that the expired timer is associated with a packet at the head of the queue. Hence, when a timeout occurs, the sender retransmits all packets in the queue, resets all timers associated with these packets and goes into the Retransmission state. Procedures StartTimer and StopTimer find an appropriate timer and set or re-

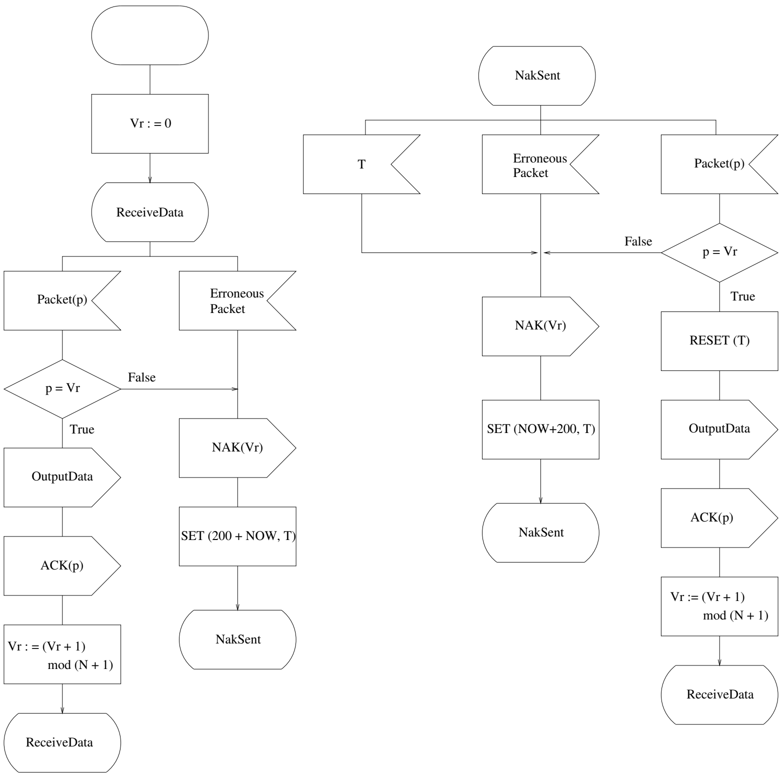

Figure 2: The SDL specification of the receiver.

set it, respectively. Procedures Top, Enqueue, and Dequeue allow the explicit use of the queue q. The SDL specification for the state DataTransfer of the sender is shown in Figure 1. The specification for the state Retransmission is similar except that user requests are not accepted.

The receiver has two states ReceiveData and NakSent, and one local variable Vr indicating the sequence number of the expected packet. The receiver starts in the state ReceiveData, where it awaits the arrival of a legitimate or an erroneous (corrupted) packet. If a legitimate packet is the one the receiver is expecting to see, it gives the information to the application (OutputData), acknowledges the receipt of the packet and updates the expected number of the next packet. If an erroneous packet is received or a legitimate packet is out of sequence, the receiver transmits a NAK indicating the packet number it is expecting, restarts the timer on that NAK and goes to the state NakSent. Since the receiver only expects one legitimate packet at a time, it needs just one timer for the corresponding NAK. In the state NakSent, the receiver waits until a packet is received or the timer expires. Unless a legitimate packet is received, the receiver sends another negative acknowledgment and restarts the timer. In our model, the timer goes for 200 units (an arbitrarily chosen number) before expiring. The SDL specification of the receiver is shown in Figure 2.

3.2 Modeling the AR Protocols

The SDL model of the receiver in the AR protocols is the same as that of the basic Go-Back-N protocol. Since an AR protocol retransmits the last K packets while the channel is idle, and since the total transmission time of those K packets has to be less than the propagation delay time, the retransmission of the last K packets can be viewed as an atomic operation in the SDL model. Hence, the SDL model of the sender in the AR protocols differs from that of the basic Go-Back-N protocol in only two places:

1. When the window is closed ($Va = N$)

and the application requests to send another packet (ServiceRequest), then, instead of following branch (1) in Figure 1, the sender of the AR protocol goes into the Retransmission state. We assume that the networking application supplies a steady stream of input requests, and thus the sender is not idle if it is in the DataTransfer state and the window is open.

2. When in state Retransmission, the sender of the AR protocol keeps retransmitting the last K packets if there is no input, i.e., no ACKs, NAKs or timeouts. We refer to this as "no real input" event.

The SDL model of the AR protocol keeps track of the last K transmitted packets using K variables, each denoting the sequence number of the corresponding packet. K here is assumed to be a constant. We use SDL's spontaneous transition to model the above "no real input" event.

The seemingly minor differences between the two protocols may be a determining factor in proving or disproving correctness. In particular, they can provide very different external functionality, e.g., one may deadlock while the other is deadlock-free, etc.

In the rest of this paper we explore techniques for ensuring that the AR and the Go-Back-N protocols are indistinguishable from the networking application's point of view.

4 Translation Techniques

In this section we describe techniques to translate SDL models into Concurrency Workbench (CWB).

The semantics of CWB is based on Milner's CCS [23]. As in CCS, input and output signals of a process correspond to observable *actions*, and the process with a particular state is defined by an *agent* over such actions. A state change of a process is expressed explicitly as a transition from one agent to another, and hence specifications of systems are represented by networks of agents. CWB does not support variables, and thus all actions and system states are represented by constants. Agents run independently and synchronize when output of one becomes input to another.

Researchers trying to verify SDL models discovered that it cannot be effectively done without some simplifying assumptions [17]. In particular, SDL models, by definition, are not finite. SDL allows the use of infinite channels and infinite-domain variables and data structures. Thus, we decided to pick a non-trivial subset of SDL (motivated by the constructs used in modeling the Go-Back-N protocols) for translation into CCS. In this section, we discuss the subset of SDL used, modeling the environment, and translating timers and some abstract data types.

4.1 Subset of SDL

The subset of SDL we can analyze includes local variables and arrays, procedures, input and output constructs, timers, control structures, assignments and arithmetic operations in task statements, and a selected set of abstract datatypes. In addition, we assume that there are no recursive procedures or pointers, all variables are finite-domain, and the length of all arrays has been explicitly defined. These assumptions are made to ensure that statements about SDL specifications are formally decidable, without the use of abstraction.

In addition, we make a number of assumptions to enable reasoning about channels. First, we need to specify a bound on the size of message channels and the maximum number of fields per message [17] and determine the behavior of the channel with respect to propagation delays. Following the assumptions made in SDT, we assume that channels communicate information without a delay (*no-delay*). If a delay is involved, the design should contain a separate process modeling the delay explicitly. The above assumption makes the channel in our subset of SDL behave like a bounded queue, described in detail in Section 4.3.

We specify our SDL models in SDT and assume that they pass the SDT's syntax and semantics checker. In addition, we assume that these models have been checked for deadlock-freedom.

4.2 Modeling Environment

Verification of the protocols would be incomplete without modeling and reasoning about the environment. The environment is often unpredictable, and its effective modeling requires the use of the non-deterministic choice construct any. For example, the environment in our analysis of Go-Back-N and AR protocols is represented by a (unreliable) communication channel. This channel either delivers packets or corrupts/looses them. The SDL model of the medium is shown in Figure 3.

The channel receives a packet p or a positive or negative acknowledgment of a packet. It non-deterministically chooses whether to forward the packet/acknowledgment, loose it, or send a corrupted message, represented in Figure 3 by ErroneousPacket. Note that, according to this specification, all packets may be lost, and the effect of corrupting an acknowledgment is equivalent to loosing it in the channel, because the sender and the receiver ignore messages that they cannot decode. p is the unique identifier of the packet, ranging from 0 to N, inclusive. Figure 4 presents the global view of the protocols.

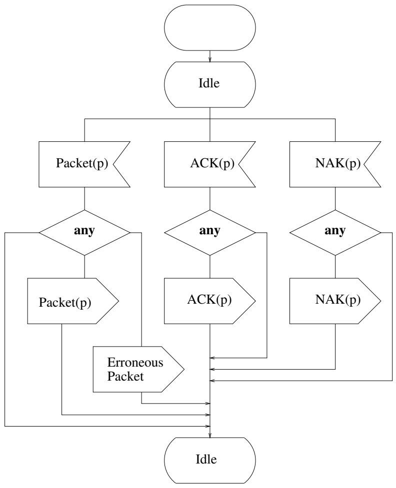

Figure 3: The SDL specification of the medium.

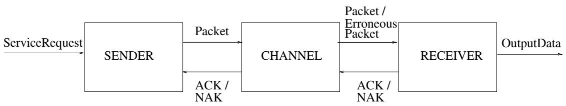

Figure 4: Global view of the protocols.

4.3 Translation in a Nutshell

Since the CWB does not support variables, we have to explicitly instantiate the SDL variables to obtain all possible values. Still, semantics of SDL projects itself to process algebras fairly easily [3, 28]. SDL's inputs and outputs are mapped into CWB inputs and outputs, respectively, and values of relevant local variables are encoded into CWB states[2]. Message passing is facilitated via encoding the state of the channel (queue) into CWB states and using synchronization for control-flow. Since we assume that SDL models do not have recursion, SDL procedures can be expanded in-line. When arithmetic operations are involved, they have to be explicitly performed to determine the appropriate CWB action. Loops are translated by creating a CWB state corresponding to the join node of the loop, determining the variables that the loop depends on, and encoding these into the state.

For example, consider translating the medium specified in extended SDL in Figure 3. When N is 3, the medium can send and receive messages with sequence numbers ranging between 0 and 3. These messages can also be correctly or incorrectly acknowledged. Thus, the medium can be translated as shown in Figure 5.

The process described here was feasible for analyzing the Go-Back-N protocol for small values of N. However, since all CWB verification algorithms are exponential in the number of states, inputs and outputs, it is realistic to expect that in order to keep verification feasible, even finite-domain variables have to be abstracted or ignored altogether, as in [28].

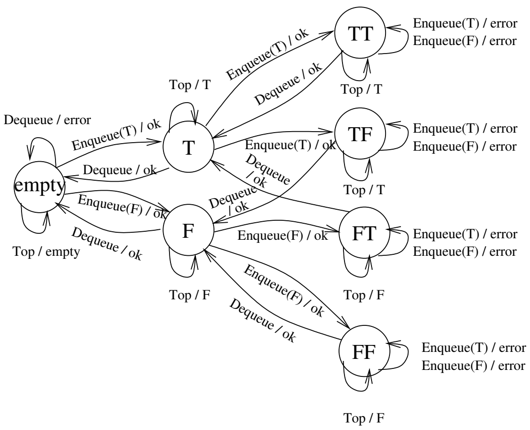

Figure 6: A 2-slot boolean queue.

4.4 Abstract Data Types

SDL models often contain abstract data types. The most common among them is a channel, which is effectively an unbounded FIFO queue, with or without delays. In our description of Go-Back-N

[2] A simple data-flow check allows us to determine the variables that a state depends on.

```
agent Link =
  send0.('r0correct.Link + 'r0error.Link + Link) +
  send1.('r1correct.Link + 'r1error.Link + Link) +
  send2.('r2correct.Link + 'r2error.Link + Link) +
  send3.('r3correct.Link + 'r3error.Link + Link) +
  sack0.('rack0.Link + Link) +
  sack1.('rack1.Link + Link) +
  sack2.('rack2.Link + Link) +
  sack3.('rack3.Link + Link) +
  snak0.('rnak0.Link + Link) +
  snak1.('rnak1.Link + Link) +
  snak2.('rnak2.Link + Link) +
  snak3.('rnak3.Link + Link);
```

Figure 5: Translation of the medium.

and AR protocols, we used the queue explicitly, via commands Top, Enqueue and Dequeue. We treat a queue as a state machine parameterized by the type of data stored in it. For example, Figure 6 depicts a state-transition diagram of a 2-element queue containing boolean data. The queue receives input Top, Enqueue(value), Dequeue and gives output – a top value, error if an operation cannot be performed, or ok if the operation is performed correctly. Each state in this Figure encodes the elements in the queue: none (EMPTY), singleton elements (T and F) and two elements (TT, ..., FF). For example, when the queue receives the command ENQUEUE(F) while in state T, it transitions into state TF and returns the output OK.

We allow to specify three types of queues: arbitrary queues of a given size, queues that do not contain repeated elements, and ordered queues without repetitions. The more is known about the type of the queue, the fewer states will the resulting CWB model have, the faster will the verification go. An arbitrary queue of size k containing elements of a datatype with domain of size M has $1+M^1+M^2+ ... + M^k = \frac{1-M^{k+1}}{1-M}$ states in its state machine representation. The same queue that contains elements without repetitions has $1 + M + M(M-1) + \dots \frac{M!}{(M-k)!}$ states. Finally, an ordered queue has only $1 + M + M + ...M = 1 + Mk$ states.

Unfortunately, there is no exact heuristic for determining the adequate size of these finite ADTs. There is a trade-off between the possibility for an overflow of an ADT and the amount of memory available for verification.

At the moment, our translation algorithm can only support stacks and queues with fixed behavior. These are translated into CWB as separate agents running in parallel with the rest of the system and synchronizing on inputs and outputs.

Although the above treatment of ADTs is adequate for analysis of the Go-Back-N protocol, it is not general. SDL standard Z.105 allows one to specify ADTs using the axiomatic style. Translating ADTs expressed in this way is part of our future research plans.

4.5 Translating Timers

Most SDL models contain timers, and thus we need effective means of analyzing them. The timers can be abstracted away after a simple check that each set timer has a "timer expired" event associated with it. This check can be performed via a simple dataflow analysis of the SDL model. We abstract from the value of the timer, replacing it by a CWB action `tau` which can be non-deterministically chosen as an input. All timers that lead to the same behavior are replaced by a single timer. A more sophisticated treatment of SDL timers will probably not be effective because of their semantics: the timers are never explicitly decremented.

Note that the above abstraction is strictly conservative and can be done only because we do not check the protocols for timing-equivalence. In our case, the protocol models use timers just to avoid infinite waiting, so such an intension is fully warranted. However, in a general case applying the above abstractions removes all timing information from the models, and timing-equivalence violations will go undetected.

5 Analyzing Go-Back-N and AR Protocols

In this section we describe the results of analyzing the Go-Back-N and the AR protocols.

5.1 Translating the Specifications

Using the technique described in the previous section, we can easily translate the SDL specifications of the Go-Back-N and the AR protocols into CCS. Since the communication medium does not include any local variables, the resulting CCS model consists of a single state, as shown in Section 4.2.

The SDL model of the receiver of the Go-back-N protocol consists of two states, and we need to encode the local variable Vr, indicating the expected sequence number of a packet and ranging from 0 to N. Hence, the CWB model has $2(N+1)$ states. For clarity, we name these states ReceiveData$i$ and NakSent$i$, where $i = 0, 1, ..., N$. Figure 7 shows the state transition diagram of the receiver. Here OutputData means delivering a correctly received packet to the networking application; timeout represents the occurrence of a timeout[3]; rerror means that the receiver received an erroneous packet, while r$i$correct means that the receiver received packet i correctly; sack$i$ and snak$i$, where $i = 0, 1, ..., N$, stand for sending a positive or a negative acknowledgment for packet i, respectively. The receiver starts in the state ReceiveData0.

The SDL model of the sender of the Go-Back-N protocol has two states, DataTransfer and Retransmission, and the four variables, Vs, Va, p, and s, where Vs is the sequence number of the next packet to be sent, Va is the number of unacknowledged packets, and p and s are temporary working variables. However, global states depend only on Vs and Va; the remaining variables are just used in loops and do not affect the global states, as our translation procedure of Section 4 was able to determine. Since both Vs and Va range from 0 to N, $2(N+1)^2$ states are required in the CWB model to represent the two states and the two variables. We name these states DataTransferi, j and Retransmissioni, j where i and j correspond to Vs and Va, respectively. As shown in Figure 1, the SDL model also includes three loops, each of which involving all four variables: Vs, Va, p, and s. Therefore, the CWB model has another $3(N+1)^4$ states, called AckLoopi, j, p, s, NakLoopi, j, p, s and RetxLoopi, j, p, s.

The CWB model defines the queue, used by the sender to keep track of unacknowledged packets, as a separate agent, run in parallel with the sender, as described in Section 4.3. In the SDL model, the sender uses $N+1$ timers which lead to the same set of actions, and thus we represent these timers using a single action timeout. Figure 8 shows the state transition diagram for the sender, in which ServiceRequest means accepting a packet from the application; timeout represents the occurrence of a timeout generated by an arbitrary timer; send$i$ means that the sender sends packet i; rack$i$ and rnak$i$, where $i = 0, 1, ..., N$, stand for receiving a positive or a negative acknowledgment for packet i, respectively; NONE means that there are no observable actions. The sender starts in the state DataTransfer$0, 0$. In this Figure, we use x = y to indicate a condition that x and y have the same value, and x := y to indicate an assignment of y to x.

As mentioned in Section 3, the only difference between the AR and the Go-Back-N protocols occurs when the channel is idle: in the former protocol the sender keeps retransmitting packers while in the latter it waits for an acknowledgment or a timeout. The two versions of the AR protocols differ just by values of K. As with the Go-Back-N protocol, the SDL specifications of the AR protocols can be automatically translated into CCS using techniques described in Section 4, assuming that K is a constant.

5.2 Choosing the Type of Analysis

Before describing the verification of protocols, we want to discuss the type of analysis that guarantees that the protocol models are indistinguishable from the application's point of view. That means, among other things, that all safety and liveness properties of one model hold in the other.

We start by giving an informal definition of bisimulation.

Definition 1: *Two systems are bisimilar if they offer the same external actions in the initial states, and each external action offered leads the systems to subsequent bisimilar states.*

Note that bisimulation is stronger than trace equivalence – the latter preserves only safety and

[3] We use timeout here for clarity. The actual CWB model uses tau to represent a timeout.

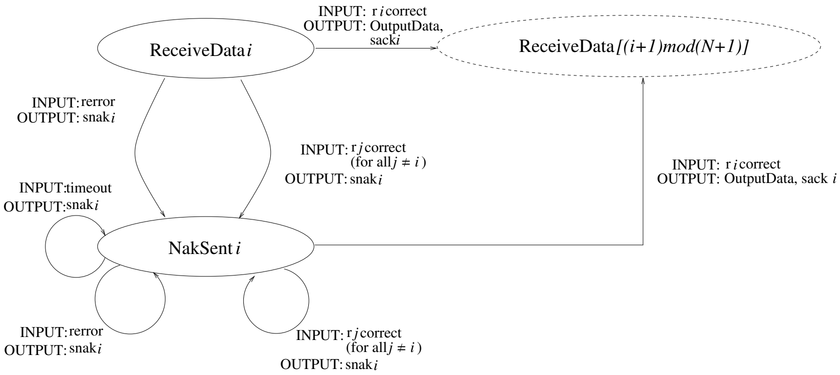

Figure 7: The state transition diagram of the CWB model of the receiver.

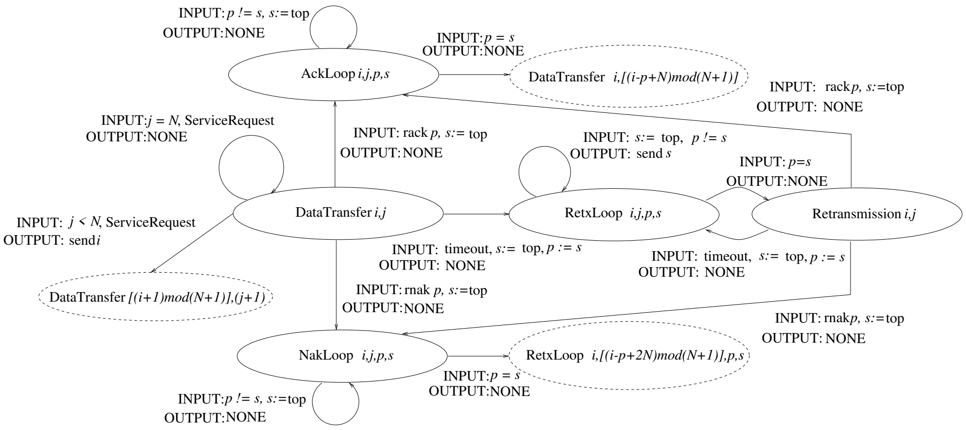

Figure 8: The state transition diagram of the CWB model of the sender.

not liveness properties of the systems, and thus the application can distinguish between two descriptions of trace-equivalent systems. The above definition also implies that if two systems are (weakly) bisimilar and one does not diverge or deadlock after a series of actions, neither does the other. Thus, it is sufficient to determine that the protocol models are bisimilar, and that one model is deadlock-free.

Further discussion of the types of bisimulation and other behavioral equivalences is outside the scope of this paper. For more information, refer to [14, 24].

5.3 Verifying the Protocols

The two versions of the Go-Back-N protocol are the same when their externally-observable actions are limited to ServiceRequest and OutputData. We attempted to analyze CWB models with $N = 1, 2, 3$ and $K = 1, ..N$, and summarize our results below.

First of all, we found that the CWB models with different types of queues are equivalent as long as the queues are used the same way. The same graph is generated for the bisimulation analysis of a given protocol, regardless of the type of queues used. The reason is that the protocol sender assumed that the packets were ordered, and thus the states representing the queue with out-of-order elements were unreachable. In fact, under the assumption that the networking application sends packets in order, we did not need to have a queue at all – it is possible to hand-code the CWB models of the protocols using just Va and Vs, as shown in [32]. However, such a model could not be automatically generated from the SDL description of the protocol.

	Number of States	**Time for Deadlock Freedom**	**Time for Equivalence Checking**
Go-Back-1	951	0.86(sec)	—
$K = 1$	951	0.84(sec)	1.63(sec)
Go-Back-2	6810	3.39(sec)	—
$K = 1$	6810	3.84(sec)	5.33(sec)
$K = 2$	6810	4.04(sec)	5.15(sec)
Go-Back-3	42240	10.06(sec)	—
$K = 1$	42240	10.20(sec)	35.39(sec)
$K = 2$	42240	10.47(sec)	32.56(sec)
$K = 3$	42240	10.63(sec)	33.61(sec)

Table 1: Summary of verification results.

We checked each model for deadlock freedom, using the CWB command fd, and verified weak observational equivalence (CWB command eq) between the AR and the corresponding Go-Back-N protocols, $K = 1, .., N$. Note that since we explicitly modeled the environment, our protocols are only equivalent (bisimilar) in the assumed environment, that is, the equivalence is *context-sensitive*. Table 1 shows the size of the resulting models and the running times for performing the checks. The time measurements are the user time as given by the Unix `time` command. The verification was done on a lightly-loaded PC (2 550 MHz Pentium III processors) with 1 Gb of RAM using Edinburgh Concurrency Workbench Version 7.1.

6 Conclusions and Future Work

In this paper, we introduced a new family of communication protocols called AR, and used Concurrency Workbench to analyze bisimulation equivalence between the Go-Back-N and the AR protocols. The protocols were originally modeled in SDL, and we extended the existing translation procedures between SDL and CWB to handle more complex SDL structures. The new procedure includes reasoning about procedures, timers, and some abstract data types, useful in expressing interprocess communication. The procedure has been shown to be general enough to handle non-trivial problems. We also presented results of analyzing the protocols in CWB. Note that it SDL models can be easily translated into Promela for analyzing them with the SPIN model checker [19]. Promela was originally designed for analysis of SDL protocols, and thus such a translation is very natural [31], although problems similar to the ones discussed earlier in this section still arise. SPIN can be used to check the model for deadlock-freedom and can verify it against temporal logic properties. Extensions of SPIN [9, 10] allow to check a class of relations which includes testing equivalences. However, SPIN cannot check weak equivalence between two SDL models. Although at this point CWB is not as fast and scalable as other model-checkers, e.g. SMV [22] or SPIN, we were able to use it effectively for analyzing equivalence between our protocols.

Since equivalence checking is essential in ensuring that "better" protocols remain usable from the application's point of view, better checking techniques are necessary. We feel that the emerging popularity of reasoning about logical formulas using binary decision diagrams can significantly increase the size of CWB models that can be analyzed.

Still, the most essential problem in ensuring effective verification of SDL-expressed protocols is combating the state explosion problem. We feel that the most effective path towards this goal is via domain-specific abstractions, e.g., abstractions of SDL timers, datatypes, channels, etc. A closely-related problem involves interpreting results of verification on abstracted models [6, 5]. An ability to find usable abstractions would allow reasoning about infinite-state SDL models. This step is essential in order to make the analysis outlined in this paper applicable to real-world examples.

Acknowledgments

We would like to thank Andre Wong and Wei Zhou for their help with SDL tools. Discussions with Stefan Leue were extremely helpful. We are also grateful to Hakan Erdogmus and anonymous referees whose suggestions improved the presentation of this paper. The research was supported in part by the Natural Science and Engineering Research Council of Canada.

About the Authors

Marsha Chechik received her Ph.D. from the

University of Maryland in 1996 and is currently an assistant professor in the Department of Computer Science at the University of Toronto. She can be reached at the address Department of Computer Science, University of Toronto, 10 King's College Rd., Toronto, ON M5S 3G4. Her e-mail address is `chechik@cs.toronto.edu`. Dr. Chechik's interests are mainly in the application of formal methods to improve quality of software. She is also interested in other aspects of software engineering and in computer security.

Hai Wang is a Ph.D. student in the Department of Computer Science at the University of Toronto, working under the supervision of Prof. Ken Sevcik. He can be reached at the same address. His e-mail is `hai@cs.toronto.edu`. Hai's research interests are in the areas of database and performance modeling.

References

[1] B.A.N. Aulia and P.H.A. Venemans. "A Skeleton for the Specification of OSI Protocols in SDL". In *Proceedings of the Fourth SDL Forum*, Lisbon, Portugal, October 1989.

[2] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall, 1991.

[3] J. A. Bergstra and C. A. Middelburg. "Process algebra semantics of φSDL". In *Proceedings of the Second Workshop on Algebra of Communicating Processes*, pages 309–346, May 1995.

[4] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. "IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *LNCS*, pages 307–327. Springer-Verlag, September 1999.

[5] M. Chechik. "On Interpreting Results of Model-Checking with Abstraction". (submitted for publication), May 2000.

[6] M. Chechik and W. Ding. "Lightweight Reasoning about Program Correctness". CSRG Technical Report 396, University of Toronto, March 2000.

[7] Rance Cleaveland. "Tableau-Based Model Checking in the Propositional Mu-Calculus". *Acta Informatica*, 27:725–747, 1990.

[8] 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.

[9] H. Erdogmus. "Formal Verification Based on Relation Checking in SPIN". In *Workshop on Formal Methods in Software Practice*, San Diego, California, January 1996.

[10] H Erdogmus. "Architecture-Driven Verification of Concurrent Systems". *Nordic Journal of Computing*, 4:380–413, 1997.

[11] Christian Facchi, Markus Haubner, and Ursula Hinkel. "The SDL Specification of the Sliding Window Protocol Revisited". Technical Report TUM-I9614, Technische Univerität München, 1996.

[12] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. "CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox". In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, New Brunswick, New Jersey, USA, 1996. Springer Verlag.

[13] F. Halsall. *Data Communications, Computer Networks and Open Systems*. Addison Wesley, Reading, Massachusetts, 3 edition, 1992.

[14] Matthew Hennessy and Robin Milner. "Algebraic Laws for Nondeterminism and Concurrency". *Journal of ACM*, 32(1):137–161, January 1985.

[15] T. Higashino, K. Ninomiya, T. Kimoto, K. Taniguchi, and M. Mori. "Automated Verification of Equivalence of Protocol Machines". In *Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing, and Verification*, pages 235–246, 1990.

[16] G. J. Holzmann. "Practical Methods for Formal Validation of SDL Specifications". *Computer Communications*, 15(2):129–134, March 1992.

[17] G. J. Holzmann. "The Theory and Practice of a Formal Method: NewCoRe". In *Proc. IFIP World Computer Congress*, Hamburg, Germany, August 1994. (invited paper).

[18] G. J. Holzmann and J. Patti. "Validating SDL Specifications: an Experiment". In *Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing, and Verification*, pages 317–326, Enschede, The Netherlands, June 1989.

[19] G.J. Holzmann. "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[20] ITU-T. *Specification and Description Language (SDL): ITU-T Recommendation Z.100*. International Telecommunication Union, 1996.

[21] S. Lin, Jr. D. J. Costello, and M. J. Miller. "Automatic-Repeat-Request Error-Control Schemes". *IEEE Communication Magazine*, 22(12):5–17, December 1984.

[22] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[23] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[24] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[25] Faron Moller and Perdita Stevens. "The Edinburgh Concurrency Workbench (Version 7)", September 1996.

[26] J. Parrow. "Verifying a CSMA/CD-protocol with CCS". In *Proceedings of the IFIP WG6.1 Eighth International Symposium on Protocol Specification, Testing, and Verification*, pages 373–384, Atlantic City, NJ, June 1988.

[27] H. Saito, T. Hassegawa, and Y. Kakuda. "Protocol Verification System for SDL Specifications Based on Acyclic Expansion Algorithm and Temporal Logic". In *Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 511–526, 1992.

[28] H.-A. Schneider and D. Taubner. "Process Algebra Techniques for Verification of SDL-Diagrams". In *Proceedings of the Eighth International Conference on Software Engineering for Telecommunication Systems and Services*, pages 107–111, Florence, Italy, March/April 1992.

[29] Telelogic. "Telelogic SDT Home Page". `http://www.telelogic.com/solution/tools/sdt.asp`, September 1998.

[30] D. Towsley. "The Stutter Go-Back-N ARQ Protocol". *IEEE Transactions on Communications*, COM-27(6):869–875, June 1979.

[31] Heikki Tuominen. "Embedding a Dialect of SDL in Promela". In *Proceedings of 5th and 6th Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 245–260, November 1999.

[32] Hai Wang and Hwei Sheng Teoh. "Automatic Verification of Asynchronous Retransmission Go-Back-N ARQ Protocols Using the Concurrency Workbench". In M. Chechik, editor, *Automated Verification: A Collection of Reports*. University of Toronto, Technical Report CSRG-374, April 1998.