A Validation Environment for LOTOS[1]

B. Ghribi and L. Logrippo

University of Ottawa, Computer Science Department
Telecommunications Software Engineering Research Group
Ottawa, Ontario, Canada, K1N 6N5

We present an environment to analyze LOTOS specifications, based on the integration of several tools developed by the LOTOS research group of the University of Ottawa. Main components of this environment are an interpreter, a symbolic behaviour graph generator, and a model checker, which is able to formally verify whether certain properties expressed in temporal logic are true in the symbolic behaviour graph of the specification. This process is illustrated by using a simple "data link" service provider and a "transport" service handler.

Keywords codes: C.2.2 ; D.2.1; D.2.2.4
Keywords: Network Protocols; Requirements/Specifications; Program Verification

1. Introduction

This paper presents a set of mutually integrated tools that have been developed at the University of Ottawa for the validation of LOTOS specifications. This set includes a step-by-step interactive executor, similar to HIPPO [vE89], but with additional features related to those of SMILE [EW92], a behavior graph generator having features similar to those of LOLA [QFP88], and a CTL model checker. We show how the features of the toolkit can be used to validate specifications, with emphasis on the use of the model-checker.

2. The University of Ottawa LOTOS Toolkit

A prototypal environment for the analysis of LOTOS specifications has been developed. It is based on the integration between the following tools :

(i) ISLA [HH88], an interpreter that helps to simulate the specifications and discover design errors by giving the designer the ability to monitor and trace some execution sequences. ISLA provides a wide range of services [GHL88]. First the syntax and the static semantics of a LOTOS specification are checked and an internal representation is generated. This internal form is used to execute the

[1]Appeared in: Protocol Specification, Testing and Verification, XIII, A. Danthine, G. Leduc and P. Wolper (Editors), Elsevier Science Publishers B.V. (North-Holland) 93-108

specification in a step-by-step mode. The designer has to play the role of the environment and resolves non-determinism, by deciding which next action should be executed and by providing the required value expressions. It is possible to go back to previous points in the execution and explore other alternatives. This allows flexibility in moving between states of the transition system.

(ii) SELA [Ash92], a tool that generates the behavior graph of a specification (or process) symbolically, that is, by using symbolic values instead of value expressions. Also, SELA provides the means to translate the generated graph into a monolithic style specification and other formats useful for analysis of the original specification.

(iii) LMC [Ghr92], a model checker for full LOTOS that is capable of verifying temporal logic properties on the graph model generated by SELA. The properties are expressed in the branching temporal logic CTL [CE81, CES86].

The existence of the tools described above enables a design methodology involving several phases (Figure 1.):

1. **Initial phase**: the designer starts with the formulation of some informal requirements of the system to be developed.
2. **Specification phase**: the design is then formally specified in terms of a set of interacting processes in a structured top-down way, by using LOTOS. ISLA is then used to find elementary errors by executing the specification step-by-step.
3. **Expansion phase**: this phase deals with the generation of the symbolic behaviour graph from the LOTOS specification. If the graph is too large, it is cut to some user-specified length.
4. **Verification phase**: at this point, the designer formulates the initial requirements by means of a set of temporal logic properties to be provided to the model checker. The model checker determines if the properties are true or false for the system specified, by exploring the graph generated in the previous phase.

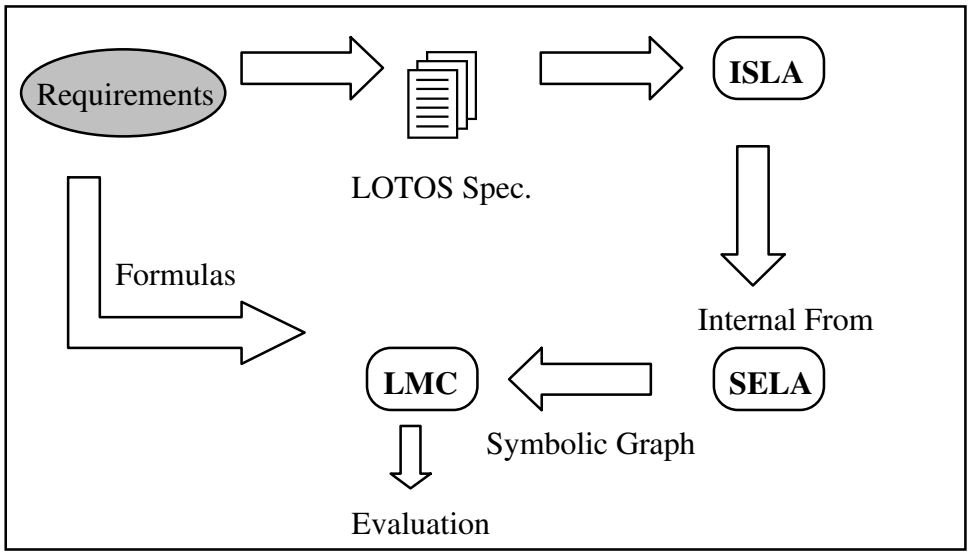

Figure 1. The Validation Environment

3. The Existing Model Checkers for LOTOS

A few model checkers for LOTOS have been reported in the litterature. The first system that we know of is the CÆSAR/CLÉOPÂTRE system [GAR89a, GS90, ROD88, RAS90, FGMRS91]. In this system, LOTOS specifications are first compiled into extended Petri Nets using CÆSAR [GAR89a], then into state graphs for verification purposes. CLÉOPÂTRE represents a validation tool for specifications expressed in the branching-time temporal logic LTAC [QS83]. It includes a model checking module [ROD88] and an explanation module [RAS90], which provides diagnostics based on sequences extracted from the graph when a formula is not valid. This tool produces extremely large models, although they are generated and stored efficiently. For example the alternating bit protocol results in thousands of states, depending on the number of messages one wants to consider, while we shall see that in our approach only 31 states are needed.

The other model checkers are based on the adaptation of existing tools. The core of one of them [DFGR91] are two existing tools, AUTO [dSV90] and EMC (Extended Model Checker) [CES86]. The first builds the labelled transition system and allows proof of equivalence and simplification of terms, while the second checks the validity of CTL formulas [DFGR91]. Similar systems are reported in [JKP89] and [Karj92].

4. Computation Tree Logic (CTL)

CTL is closely related to branching-time logics proposed in [Lamp80, EC81, BAPM83] and was itself proposed in [CE81].

Definition
Let Prop denote a set of atomic propositions, the set of CTL formulas is defined recursively as follows:

- Every atomic proposition p $\in$ Prop is a CTL formula.
- if f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, $AX(f_1)$, $EX(f_1)$, $A[f_1 \upsilon f_2]$, $E[f_1 \upsilon f_2]$.

$AX(f_1)$ means that f_1 holds in every immediate successor of the current state.
$EX(f_1)$ means that f_1 holds in some immediate successor of the current state.
$A[f_1 \upsilon f_2]$ means for every computation path, starting at the current state, there exists a sequence of transitions satisfying f_2 at last, and f_1 for all the other transitions.
$E[f_1 \upsilon f_2]$ means for some computation path, starting at the current state, there exists a sequence of transitions satisfying f_2 at last, and f_1 for all the other transitions.
The following abbreviations are also used in writing CTL formulas:
$AF(f) \equiv A[True \upsilon f]$ means that f holds in the future along every path from the initial state q_0; in other words f is *inevitable*.

$EF(f) \equiv E[True \cup f]$ means that f holds along some future path from q_0; in other words *f potentially holds*.

$AG(f) \equiv \neg AF(\neg f)$ means that f holds at every state for every future path; that is *f holds globally*.

$EG(f) \equiv \neg EF(\neg f)$ means that f holds at every state for some future path.

Note that some adaptation of the theory of CTL had to be done, mostly in order to take into consideration the fact that LOTOS has finite computations, due to deadlocks. These adaptations are straightforward and were detailed in [Ghr92].

5. The Symbolic Expansion

The symbolic expander SELA applies LOTOS inference rules to a behaviour expression. The possible actions and the set of resulting behaviour expressions can be obtained. By iteratively applying the same rules on the resulting behaviour expressions, the whole behaviour graph can be generated. The expansion of a branch ends when one of the following is met:

- Explicit stop or deadlock resulting from synchronization.
- Duplicate behaviours (identical as chains of characters). SELA checks whether a reached behaviour expression has already been generated in which case a loop is detected (we are planning to generalize this identity criterion by adding some useful equivalence rules).
- Behaviours which are duplicated except for the value of some value expressions. This can be due to recursion, where the current behaviour is the instantiation of a process (or a set of processes in the case of parallelism) which had already been instantiated previously. The detection is achieved by replacing all bound value identifiers (i.e. value identifiers which were bound to some value expressions) appearing in these behaviours by new value identifiers, and comparing the resulting behaviours to see if they are duplicates.
- Upper bounds provided by the user for the width or the depth of the behaviour graph.

Formal parameters can have different values for different instantiations. Therefore the evaluation of the guards or selection predicates is not sufficient to allow the truncation of a branch. Guards and selection predicates that are evaluated to be always true are kept in the corresponding branches whereas those that are always false are eliminated and so are the branches that follow them. In certain cases, it is possible to evaluate to *true* or *false* predicates involving variables (that is, "narrowing" [RKKL85] can be performed to a certain extent, and this capability is being enhanced). SELA does not prune a branch when the guards or predicates cannot be evaluated. The way SELA expands LOTOS behaviours is explained in detail in [Ash92].

6. Formulation of the Properties

The formulas accepted by the model checker are a combination of the CTL operators and of LOTOS actions that include only offer experiments. The user can inject the value expressions to be provided by the environment into the model representing the specification by supplying them within the formulas. We can also deal with values that can be generated internally by the system without the intervention of the environment.

Let *p* denote the basic component of a formula. It can be either a gate name associated with a list of offer events or an exit associated with values. *p* can be written as follows:

$gate_name\ Exp_1\ Exp_2 \ldots Exp_n \quad 0 \leq i$ or
$exit(Value_1, Value_2, \ldots, Value_n)$

where:

gate_name is a gate name that is declared in the formal gate list of the specification or is hidden using the *hide* operator.

Exp_i is an offer experiment of the form $! \ Value_i$ and $Value_i$ represents a value and falls in one of the following three categories:

(i) An explicit value that can be generated internally by the specification such as the sequence number associated with messages in the case of the alternating bit protocol.

(ii) A symbolic value representing values provided by the environment represented by @1, @2,...

(iii) A "don't care" value, represented by a *. Note that * is not the same as ? because in the case of a ?, a variable becomes bound to a value, while in the case of * the value of the variable is not important.

By composing these basic components using CTL operators, we can formulate the initial requirements of the system specified and use LMC to validate them. For example, the following formula: $AG(send!* \Rightarrow AF(receive!*))$ means that whenever something is sent, something will be received eventually. One can also write : $AG(send!succ(0) \Rightarrow EF(receive!succ(0)))$ which means that whenever the value *succ(0)* is sent, there is some sequence leading to receiving that value in the future.

We give two examples to illustrate the validation methodology. The first example concerns a simple data link service provider and the second deals with a transport service handler. We start by giving an informal description of the desired service, then show how LOTOS is used to specify these requirements formally. The symbolic graph obtained is provided and finally examples of properties that can be verified are given. We do not discuss the intermediate step involving the interpreter ISLA since it is used in our case only to check the syntax and static semantics of a LOTOS specification, and to generate an internal form useful for the symbolic expander SELA.

7. A Data Link Service Provider

This example was first introduced in [QFP88]. It consists of a simple, one directional data link service provider (Figure 2).

Informal Description of the Service Provider

A transmitter is connected through a semi-duplex lossy line to a receiver. The alternating bit protocol is used for error detection. This means that every message sent by the transmitter is associated with a sequence number 0 or 1.

The transmitter: When a data item is provided by the environment to the *transmitter* through gate *get*, the latter one sends it to process *line* along with a sequence number. A timer is set while the *transmitter* is waiting for an acknowledgment. Should a timeout occur before receiving the acknowledgment, the *transmitter* has no choice but to retransmit the original data. An acknowledgment with the wrong sequence number is ignored.

The receiver: The *receiver* is ready to receive a data item from process *line*, together with its sequence number. If the sequence number is as expected, the data item is delivered to the environment through gate *give* and an acknowledgment is sent with the sequence number of the next set of data. Otherwise, if the data item is old (having a wrong sequence number), an acknowledgment is sent to process *line* with the expected sequence number.

The line: Process *line* synchronizes on gate *send* with process *transmitter* when the message is of type *info*, and with process *receiver* when the message is of type *ack*. The synchronization works the opposite way for gate *receive*.

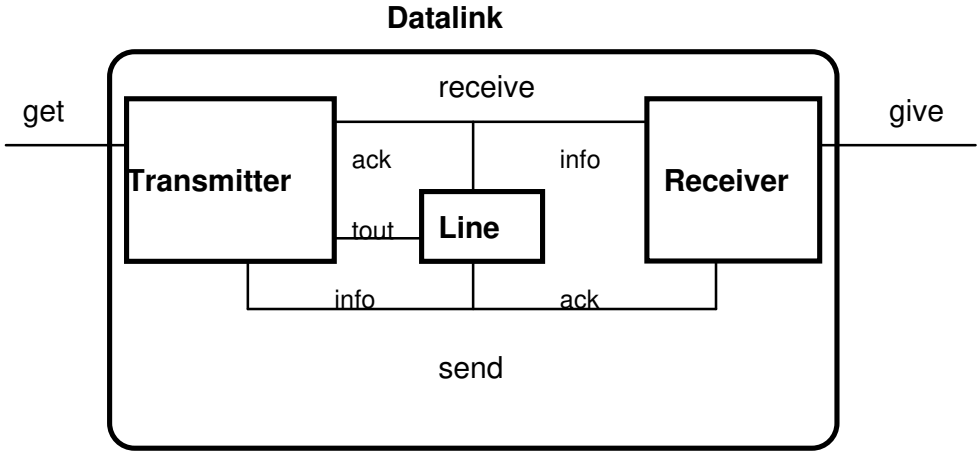

Figure 2. The Data Link Service Provider

The LOTOS Specification

We give here the main behaviour of the specification. The whole specification can be found in [QFP88] .

Since the gates *tout*, *send*, and *receive* are not used to communicate with the environment, we hide them by using the *hide* operator. Both the *transmitter* and the

receiver are executing in parallel independently (interleaving) but they have to synchronize with the *line* on the gates *tout*, *send* and *receive*. This is because the *line* is semi-duplex, so we have to ensure that it is accessed only by the *transmitter* when the latter needs to send a data item or receive an acknowledgment, or by the *receiver* when it needs to receive a data item or send an acknowledgment. This is described by the following behaviour:

hide *tout, send , receive* ***in***
((transmitter [*get, tout, send, receive*] *(0)*
  ||| *receiver* [*give, send, receive*] *(0))*
| [*tout, send, receive*]
| *line*[*tout, send, receive*]
)

Both the transmitter and the receiver are initialized with the same sequence number *0*.

The Symbolic Graph

It was possible to represent the service with *31* states and *38* transitions (Figure 3). The edges are unidirectional from the top to the bottom except for those representing cycles.

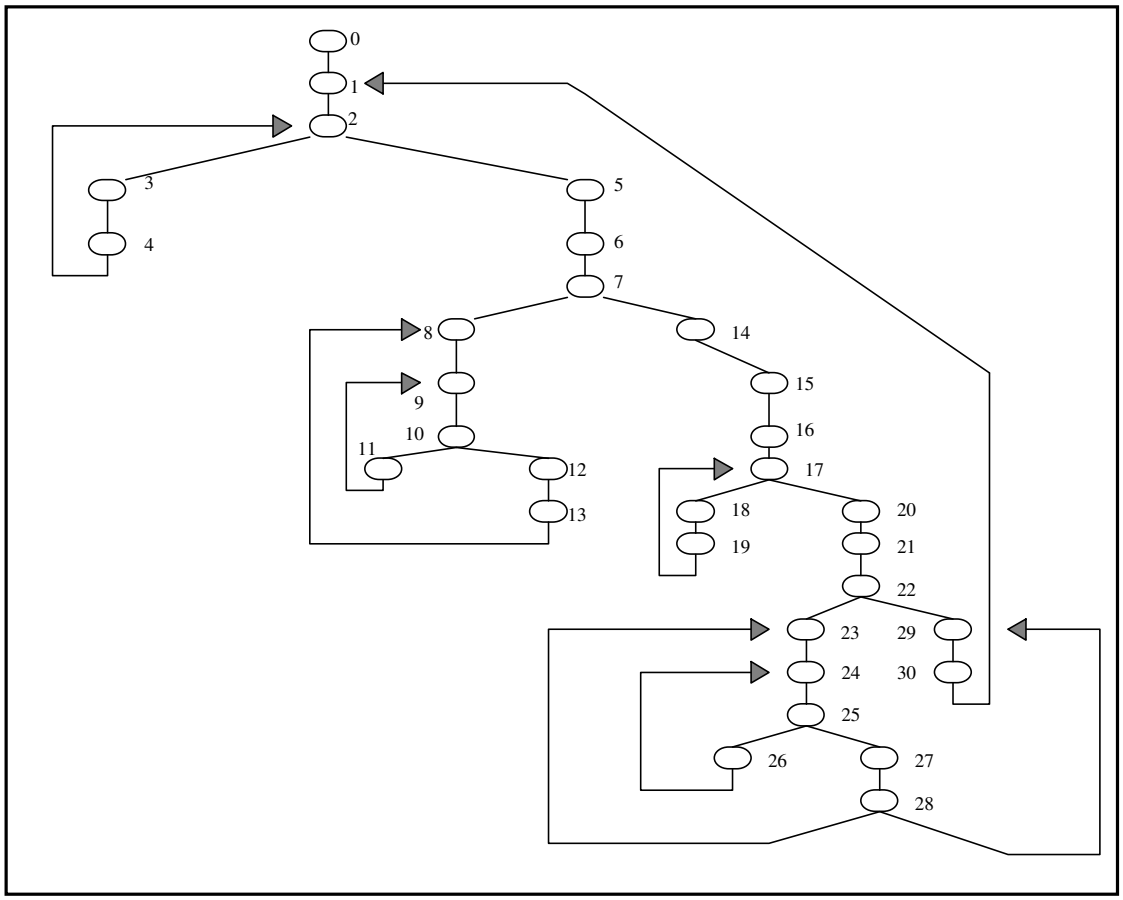

Figure 3. Symbolic graph for the datalink service provider

The guards in states 6 and 21 correspond to the case where the receiver receives a sequence number equal to the expected one (respectively 0 and 1) in which case the data item is delivered to the environment via gate *give*. States 13 and 28 represent the case where the received sequence number is not as expected. The expected sequence number at state 13 is 1 but the receiver receives 0 instead which leads to sending an acknowledgment with the expected sequence number. An acknowledgment with sequence 0 is sent at state 28.

	Guard	Action	Experiment list	Predicate	Successors
State 0	[true]	( )	[]	[true]	{1}
State 1	[]	get	[?BitS@1]	[]	{2}
State 2	[]	send	[!info,!0,!BitS@1]	[]	{3,5}
State 3	[]	i	[]	-	{4}
State 4	[]	i(tout)	[]	[]	{2}
State 5	[]	receive	[!info,!0,!BitS@1]	[]	{6}
State 6	[0 = 0]	give	[!BitS@1]	[]	{7}
State 7	[]	send	[!ack,!inc(0),!empty]	[]	{8,14}
State 8	[]	i	[]	-	{9}
State 9	[]	i(tout)	[]	[]	{10}
State 10	[]	send	[!info,!0,!BitS@1]	[]	{11,12}
State 11	[]	i	[]	-	{9}
State 12	[]	receive	[!info,!0,!BitS@1]	[]	{13}
State 13	[inc(0) =inc(0)]	send	[!ack,!inc(0),!empty]	[]	{8}
State 14	[]	receive	[!ack,!inc(0),!empty]	[]	{15}
State 15	[]	δ	[]	[]	{16}
State 16	[]	get	[?BitS@1]	[]	{17}
State 17	[]	send	[!info,!inc(0),!BitS@1]	[]	{18,20}
State 18	[]	i	[]	-	{19}
State 19	[]	i(tout)	[]	[]	{17}
State 20	[]	receive	[!info,!inc(0),!BitS@1]	[]	{21}
State 21	[inc(0) =inc(0)]	give	[!BitS@1]	[]	{22}
State 22	[]	send	[!ack,!0,!empty]	[]	{23,29}
State 23	[]	i	[]	-	{24}
State 24	[]	i(tout)	[]	[]	{25}
State 25	[]	send	[!info,!inc(0),!BitS@1]	[]	{26,27}
State 26	[]	i	[]	-	{24}
State 27	[]	receive	[!info,!inc(0),!BitS@1]	[]	{28}
State 28	[inc(inc(0))=inc(inc(0))]	send	[!ack,!0,!empty]	[]	{23,29}
State 29	[]	receive	[!ack,!0,!empty]	[]	{30}
State 30	[]	δ	[]	[]	{1}

Verification Phase

The original requirements are translated to a set of temporal logic formulas to be verified. The following are some examples of formulas that can be verified using LMC:

1) Absence of deadlock: *AG(¬deadlock)*

2) ″if the data item *@1* is offered at gate *get* to process transmitter, then it is delivered to the environment at gate *give* in some future path by the receiver″:

$AG((get!@1) \Rightarrow \quad EF(give!@1))$

The use of the symbolic value *@1* makes it possible to express the fact that any particular data item provided to the datalink by the environment will be delivered without having to deal with all the possible values that represent messages.

The user may wish a stronger property to be true, i.e. that the data item is delivered in *all* future paths. Unfortunately this property cannot be true, because the medium can refuse forever to deliver an item. In the specification, this can be seen by the fact that infinite loops are possible, i.e. a timeout can occur every time a data item is sent by the transmitter. This is represented in the model of figure 3 by the sequence of states *2*,*3*,*4* which can be repeated infinitely. Delivery in all future paths could be guaranteed by a fairness assumption [CES86, Fran86]. We have no such assumption in the version of CTL that we have adopted.

3) If we are not interested in the data item delivered, but we simply want to verify that the sequence of actions is correct, then we might check the following property: "if something is offered at gate *get* to process transmitter, then something is delivered to the environment at gate *give* in some future path by the receiver."

$AG((get!*) \Rightarrow \quad EF(give!*))$

4) "No new *receive* operation after a *receive*, until a new *send* is performed."

$AG((receive!*!*!*) \Rightarrow \quad AX(A[\neg(receive!*!*!*) \upsilon(send!*!*!*)]))$

5) If the receiver sends an acknowledgment with the sequence number *0* (i.e. it is expecting a message with the sequence number *0*) then one of the two following events should happen eventually:

- The receiver receives an info message with the wrong sequence number (*inc(0)*) in which case it should immediately send again the same acknowledgment with the expected sequence number (*0*).
- The receiver receives an info message with the expected sequence number (*0*).

$$AG(send!ack!0!empty \Rightarrow \quad AF((receive!info!inc(0)!* \wedge \quad AX(send!ack!0!empty)) \vee (receive!info!0!*)))$$

Similarly, we can prove a corresponding property when the receiver is expecting a message with the sequence number *inc(0)*.

6) "Whenever the transmitter sends a message of type info, it won't send another message of type info until the receiver receives the first message."

$AG((send!info!*!*) \Rightarrow \quad AX(A[\neg(send!info!*!*) \upsilon \quad (receive!info!*!*))]))$

This property obviously violates the protocol requirements because process transmitter can send again the same message of type info in case of timeout. It will be refused by the model checker.

8. A Transport Service Handler

This example was first introduced in [BB87]. It consists of a simple transport service handler. The service consists of three phases of conversation between two entities *A* and *B*:

- Connection establishment.
- Data transfer.
- Connection release.

Informal Description of the Service

The service is specified in terms of possible sequences of interactions (Transport Service Primitives) between the transport service provider and two session layer entities, at the transport service access points. The LOTOS specification describes only the interactions with respect to the service primitives. The parameters exchanged during the interactions are not considered. The service primitive sequences are expressed by time sequence diagrams (Figure 4):

The LOTOS Specification

As in the previous example, we give here the main behaviour of the service only:

Handler[*ConReq,ConInd,ConRes,ConCnf, DatReq,DatInd,DisReq,DisInd*]

where process *Handler* is defined as follows:

Connection_phase[*ConReq,ConInd,ConRes,ConCnf,DisReq,DisInd*]
>>
(*Data_phase*[*DatReq,DatInd*]
[>
Termination_phase[*DisReq,DisInd*])
>>
Handler[*ConReq,ConInd,ConRes,ConCnf,DatReq,DatInd,DisReq,DisInd*]

Data_phase is only enabled when process *Connection_phase* terminates successfully. On the other hand, process *Termination_phase* can disrupt process *Data_phase* at any time before it terminates. Once process *Termination_phase* exits, it enables recursively the whole service again by calling process *Handler*.

The Symbolic Graph

Because of the fact that the specification is in "basic LOTOS", the graph generated here does not contain symbolic values. In other words, the exchange of parameters between the entities is not specified. Each state contains only a service primitive or an internal action.

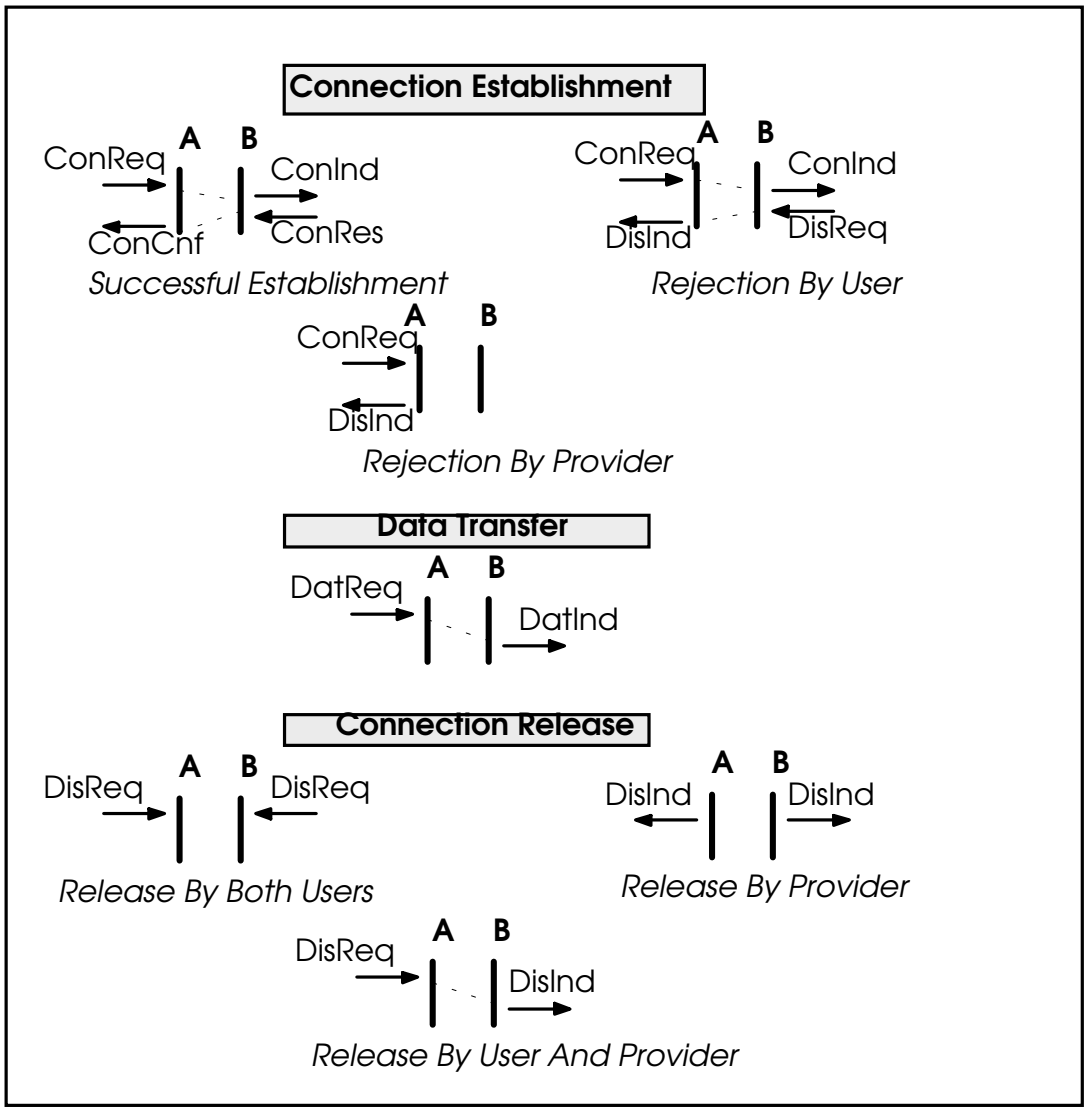

Figure 4. Transport Service Primitives

Verification Phase

Starting from the informal requirements, many interesting properties can be derived. We give here some examples of such formulas that can be evaluated by LMC. It is to be noticed that value offers are not expressed in the formulas, simply because the specification does not consider the parameters exchanged during the interactions between the two session entities.

1) Absence of deadlock: $AG(\neg deadlock)$

2) "Every request for connection *ConReq* can be granted by receiving a connection confirmation *ConCnf* or rejected, in which case, the user receives a disconnection indication *DisInd*."

$AG(ConReq \Rightarrow \quad AF(ConCnf \vee DisInd))$

3) "If a connection indication *ConInd* is received, the entity should be able to respond *ConRes*, or reject it by requesting a disconnection *DisReq*."

$AG(ConInd \Rightarrow AF(ConRes \vee DisReq))$

4) "If a connection indication *ConInd* is received, the entity should always respond by sending a *ConRes*."

$AG(ConInd \Rightarrow AF(ConRes))$

This property obviously violates the service requirements. An entity should be able to reject a connection request. This property is evaluated to false by our model checker.

5) "The user can send a connection request *ConReq* and then he can possibly request data *DatReq* without receiving a confirmation *ConCnf* to his request *ConReq*."

$AG(ConReq \Rightarrow E[\neg ConCnf \upsilon DatReq])$

This property mysteriously seems to violate the requirements of the service. However, it is satisfied. The property intends to verify whether it is possible for a user to send a *DatReq* without receiving a *ConCnf* to his request for connection *ConReq*. In fact, after sending a connection request, the user may receive a disconnection indication, and then another user may wish to establish connection with him. In that case, he can forward a connection response *ConRes* and then once the connection is established, he can send data requests. This is represented in the graph in the path 2, 20, 23, 24, 25, 5, 6 (figure 5). It can be seen easily that state 2 represents a *ConReq* and state 6 is labelled with *DatReq* and none of the intermediate states in the specified path represents a *ConCnf*.

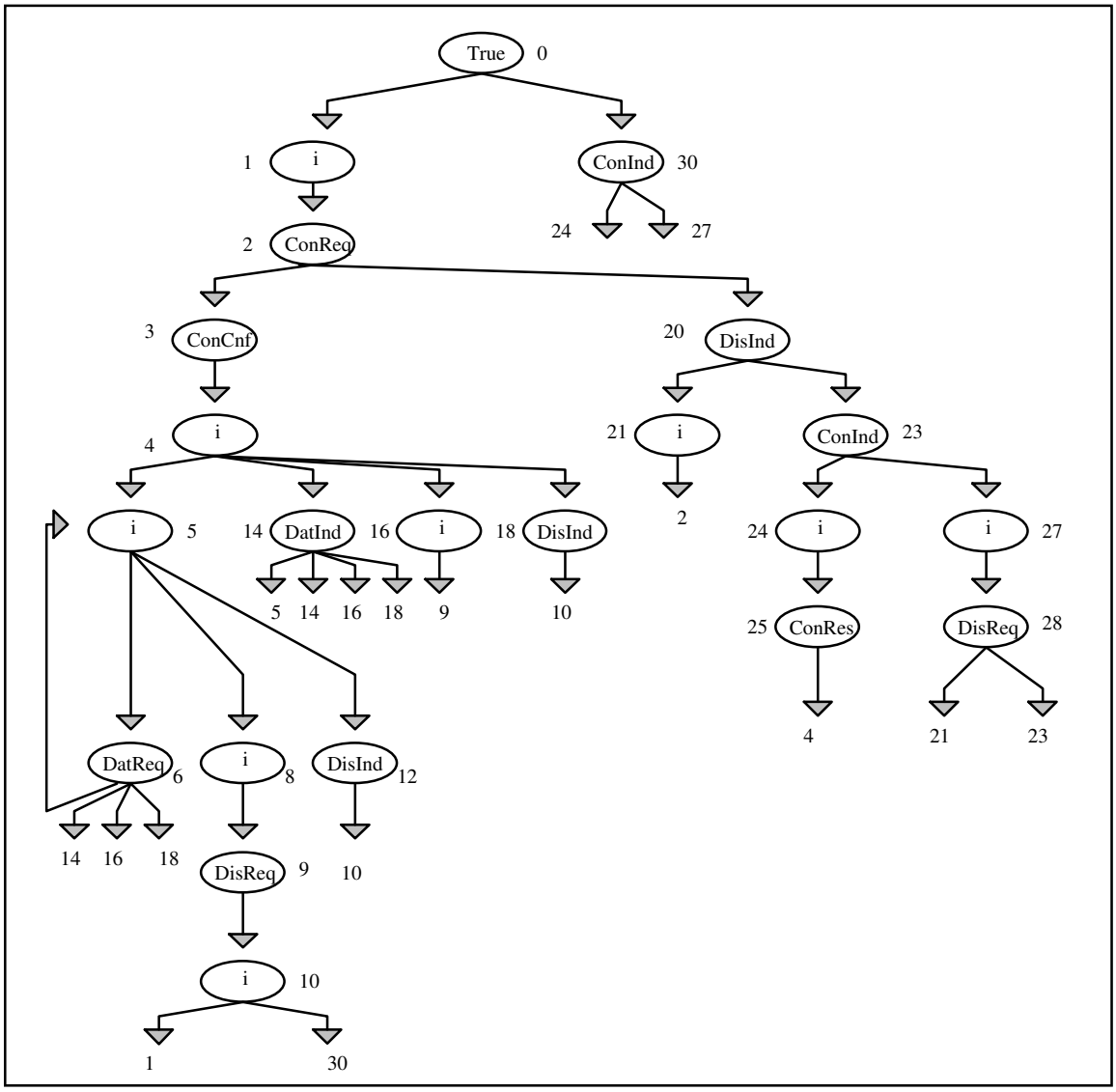

Figure 5. Symbolic graph for the Transport Service Handler

9. Conclusion

We have presented a validation environment for LOTOS based on the integration of three tools, the interpreter ISLA, the symbolic expander SELA and the model checker LMC. The useful features of LMC are:

- The possibility of expressing deadlock explicitly using CTL formulas.
- The use of symbolic values in order to denote values to be obtained from the environment.
- The possibility of using "don't care" values for experiments.

LMC is a complete and self-contained implementation of CTL. It was programmed in Prolog by the first author.

LMC can be used whenever a finite symbolic behaviour graph can be obtained from the given specification. In the case of truncated graphs, incomplete model checking is also allowed in our system.

Effective use of a model checker presupposes the existence of a diagnostic system, capable of giving the user useful information on why some properties were not satisfied. We are investigating new ways to provide automated diagnostics without the intervention of the user, based on sequences extracted from the model. Previous work has been done on this subject and can be found in [RAS90]. Also, validation methods based on the generation of the complete reachability graphs have their limitations, mainly due to the limitations in memory size. Optimized methods of generating and storing large behaviour graphs will have to be used in order to extend the applicability of the environment to larger and more practical examples.

While the use of symbolic values makes it possible to contain the size of graphs, it also implies that unfeasible execution paths may be present in a graph. Mechanisms for dealing with such paths could also be part of the diagnostic system.

A promising new method recently developed in our LOTOS research group is the Goal Oriented Execution [HHLS92]. This method helps finding to find execution paths with certain characteristics, for example, execution paths that start with one particular action and end with another one. It can find paths without having to generate the complete graph of the specification. The application of this method to model checking deserves to be studied in depth.

Acknowledgment

This research was supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the government of Canada, the Tunisian government, and the Canadian International Development Agency. We thank the colleagues of the LOTOS group of the University of Ottawa for useful discussions, and especially Pierre Ashkar and Jean-Luc La Salle, the authors of SELA. We are indebted to Ana Rosa Cavalli for insight on temporal logic.

Bibliography

[Ash92] P. Ashkar. Symbolic Execution of LOTOS Specifications. Master's Thesis, University of Ottawa, 1992.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The Temporal Logic of Branching Time. In *Proc. 8th Ann. Symp. on Principles of Programming Languages* (1981) 164-176; Journal version, *Acta Informatica*, (1983) 20:207-226.

[BB87] B. Bolognesi, E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, (1987) 14: 25-59.

[CE81] E.M. Clarke, E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. Workshop on*

Logics of Programs, *Lecture Notes in Computer Science*, (Springer, Berlin, 1981) 131:52-71.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM TOPLAS*, 8(2):244-263, April 1986.

[DFGR91] R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. *Proc. of 3rd Workshop on Computer Aided Verification*, (Aalborg, Denmark, July 1991).

[dSV90] R. de Simone, D. Vergamini: aboard AUTO, I.N.R.I.A. Technical Report 111 (1990).

[EC81] E.A. Emerson, E.M. Clarke. Characterizing Correctness Properties of Parallel Programs as Fixpoints. In *Proc. 7th Internat. Coll. on Automata, Languages, and Programming, Lecture Notes in Computer Science*, (Springer, Berlin, 1981) 85:169-181.

[EM85] B. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications*. Springer-Verlag, 1985.

[EW92] H. Eertink, D. Wolz. Symbolic Execution of LOTOS Specifications. In M. Diaz and R. Groz, (ed.), *Fifth International Conference on Formal Description Techniques FORTE'92*, (Lannion, France, October 1992).

[FGMRS91] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, et J. Sifakis. Une boîte à outils pour la verification de programmes LOTOS. In O. Rafiq (ed.), *CFTP'91, Ingenierie des protocoles*, (Rennes, 1991) 479-500.

[Fran86] N. Francez. *Fairness*. Springer-Verlag, New York, 1986.

[Gar89a] H. Garavel. Compilation et verification de programmes LOTOS. These de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[GHL88] R. Guillemot, M. Haj-Hussein, and L.Logrippo. Executing Large LOTOS Specifications. In S. Aggarwal and K. Sabnani, (eds.), *Protocol Specification, Testing, and Verification VIII*, (North-Holland, 1988) 399-410.

[Ghr92] B. Ghribi. A Model Checker for LOTOS. Master's Thesis, University of Ottawa, 1992. Available by anonymous ftp on lotos.csi.uottawa.ca.

[GS90] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R.L. Probert, and H. Ural, (eds.), *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification* , (North-Holland, June 1990) 359- 376.

[HH88] M. Haj-Hussein. ISLA: An Interactive System for LOTOS Applications.Master's Thesis, University of Ottawa, 1989.

[HHLS92] M. Haj-Hussein, L. Logrippo, J. Sincennes. Goal-Oriented Execution for LOTOS. To appear in: M. Diaz and R. Groz, (ed.), *Fifth International Conference on Formal Description Techniques FORTE'92*, (Lanion, France, October 1992).

[Hoa85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[ISO8807] ISO, IS 8807. *Information Processing Systems - Open Systems Interconnection* - LOTOS : A formal Description Technique based on the Temporal Ordering of Observational behaviour, May 1989.

[JKP89] B. Jonsson, A. H. Khan, and J. Parrow. Implementing a Model Checking Algorithm by Adapting Existing Automated Tools. *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science, (1989) 407:179-188.

[Karj92] G. Karjoth, C. Binding, and J.Gustafson. LOEWE: A LOTOS Engineering Workbench. To appear in Computer Networks and ISDN Systems.

[Lamp80] L. Lamport. Sometimes is Sometimes "Not Never"- on the Temporal Logic of Programs. In *Proc. 7th Ann. ACM Symp. on Principles of Programming Languages*, (1980) 174-185.

[LFH92] L. Logrippo, M. Faci, M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems* 23(5) 1992 325-342.

[Mil80] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, (Springer-Verlag, 1980) No. 92.

[QFP 88] J. Quemada, S. Pavón, and A. Fernández. Transforming LOTOS Specifications with LOLA: The Parametrized Expansion.In K. J. Turner, (ed.), *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88*, (North-Holland, Amsterdam, September 1988) 45-54.

[QS83] J. P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness. *Acta Informatica*,19:195-220, 1983.

[Ras90] A. Rasse. CLEO: Diagnostic des Erreurs en XESAR. These de Doctorat, Institut National Polytechnique de Grenoble, June 1990.

[RKKL85] P. Rety, C. Kirchner, H. Kirchner, and P. Lescanne. Narrower: A new Algorithm for Unification and its Application to Logic Programming. In Dijon, (ed.), *Proceedings of the First International Conference on Rewriting Techniques and Applications*, 141-157, 1985.

[Rod88] C. Rodriguez. Specification et Validation de Systemes en XESAR. These de Doctorat, Institut National Polytechnique de Grenoble, May 1988.

[vE89] P. van Eijk. The Design of a simulator tool. In P. van Eijk et al., (eds.), *The Formal Description Technique LOTOS*. (North-Holland, Amsterdam, 1989) 351-390.