Automatic Interface Generation for Enumerative Model Checking

Computer Science Annual Workshop 2006

Sandro Spina
Dept. of Computer Science and A.I.
New Computing Building
University of Malta, Malta
sandro.spina@um.edu.mt

Gordon Pace
Dept. of Computer Science and A.I.
New Computing Building
University of Malta, Malta
gordon.pace@um.edu.mt

ABSTRACT

Explicit state model checking techniques suffer from the state explosion problem [7]. Interfaces [6, 2] can provide a partial solution to this problem by means of compositional state space reduction and can thus be applied when verifying interestingly large examples. Interface generation has till now been largely a manual process, were experts in the system or protocol to be verified describe the interface. This can lead to errors appearing in the verification process unless overheads to check the correctness of the interface are carried out. We address this issue by looking at automatic generation of interfaces, which by the very nature of their construction can be guaranteed to be correct. This report outlines preliminary experiments carried out on automatic techniques for interface generation together with their proofs of correctness.

1. INTRODUCTION

Computer systems (both software and hardware) have over the past few decades been introduced into almost every piece of machinery. Real-time systems such as controllers for avionics, cars and medical equipment have become ubiquitous. Model checking techniques are used to algorithmically verify these finite state systems formally. It is becoming increasingly popular by many hardware/software manufacturers to verify that their systems actually implement the required specifications. This is achieved by verifying if the model of the system satisfies some logical specification. Suppose we want to verify, for example, that every request for service is eventually acknowledged, or that there are no deadlock states in our systems. This sort of verification can be carried out using model checking techniques.

Processes can be described using some formal process calculi such as CCS [8], CSP [3] or LOTOS [4]. Properties are then expressed as temporal logic formulas. Computational Tree Logic (CTL) [1] is one such temporal logic used to express properties of a system in the context of formal verification. It uses atomic propositions as its building blocks to make statements about the states of a system. CTL then combines these propositions into formulas using logical and temporal operators. Referring to the previous example, one would for example want to verify that every computation path after a service request is met, will eventually encounter a service acknowledgment state.

The state space explosion problem occurs with systems composed of a large number of interacting components using data structures which can potentially store many different values. The problem is clearly that of traversing the entire search space which would typically grow exponentially with the addition of new system components. One possible solution is that of decreasing the number of states in the computational graph while still maintaining an equivalent graph. In order to do so one would need to combine equivalent states (thus decreasing states) in the computational graph. We adopt the technique used by Krimm and Mounier in [6], namely *interfaces*.

We start this report with some preliminary definitions. We then focus on the theory behind our method of interface generation. Two interface generators are then explained in some detail. We conclude this report by describing how these implementations work in the verification of a reliable multicast protocol.

2. PRELIMINARY DEFINITIONS

The behaviour of a sequential process can be modeled by a labeled transition system, consisting of a set of states and a labeled transition relation between states. Each transition describes the execution of the process from a current state given a particular instruction (label).

In what follows $\mathcal{A}$ is the global set of labels, τ a particular label representing a hidden or unobservable instruction ($\tau \notin$

$\mathcal{A}$). Given a set of labels A ($A \subseteq \mathcal{A}$) we will write A_τ to denote $A \cup \{\tau\}$ and A^* to represent the set of finite sequences over A.

Definition 1 *A Labeled Transition System (LTS, for short) is a quadruplet $M = (Q, A, T, q_0)$ where Q is a finite set of states, $A \subseteq \mathcal{A}$ is a finite set of actions, $T \subseteq Q \times A_\tau \times Q$ is a transition relation between states in Q and $q_0 \in Q$ is the initial state.*

We write $q \xrightarrow{a} q'$ to denote a transition between states q and q' using $a \in A$, i.e. $(q, a, q') \in T$. We shall also use $q \overset{s}{\Rightarrow} q_n$ (where s is a string in A_τ^*) to indicate that there exist states $q_1 \ldots q_n$ following string s.

We now define the set of possible actions from a state $q \in Q$.

Definition 2 *Given an LTS $M = (Q, A, T, q_0)$ and $q \in Q$, the actions possible from state q is defined as $actions(q) = \{a : A \mid \exists q' \cdot q \xrightarrow{a} q'\}$.*

We now define the language generated by a LTS from a particular state $p \in Q$.

Definition 3 *Given an LTS $M = (Q, A, T, q_0)$ and $q \in Q$, the (observable) language starting from q in M is defined as follows:*

$$\mathcal{L}_\mathcal{M}(q) = \{\sigma \mid \sigma = a_1 a_2 \ldots a_n \wedge \exists q_1, \ldots, q_n \cdot q \overset{\tau^* a_1}{\Rightarrow} q_1 \ldots \overset{\tau^* a_n \tau^*}{\Rightarrow} q_n\}$$

The (observable) language generated by LTS M is defined as the language starting from the initial state of M: $\mathcal{L}_M(q_0)$.

3. REFINEMENTS OF LTSS

In this section we introduce the binary operator $\sqsubseteq$, that compares two LTSes.

Definition 4 *We say that M_2* is refined by *M_1 ($M_2 \sqsubseteq M_1$) if for some* total *function* eq $\in Q_1 \rightarrow Q_2$ *the following holds:*

i) $A_1 = A_2$

ii) $Q_2 \subseteq Q_1$

iii) $q \xrightarrow{a}_1 q'$*, implies that,* $eq(q) \xrightarrow{a}_2 eq(q')$

iv) $q_0 = eq(q_0)$

Lemma 1 *Given two LTS $M_i = (Q_i, A_i, T_i, q_{0i})$, where $i \in \{1, 2\}$, related with a function eq, then $q \overset{s}{\Rightarrow}_1 q'$ implies that $eq(q) \overset{s}{\Rightarrow}_2 eq(q')$*

Proof: We prove this lemma by string induction over s.

The base case, taking s to be the empty string is trivially true.

For the inductive case, we start by assuming that: $\forall q, q' \cdot q \overset{t}{\Rightarrow}_1 q'$ implies that $eq(q) \overset{t}{\Rightarrow}_2 eq(q')$.

We now need to prove that:
$$\forall q, q' \cdot q \overset{at}{\Rightarrow}_1 q' \text{ implies that } eq(q) \overset{at}{\Rightarrow}_2 eq(q')$$

But, if $q \overset{at}{\Rightarrow}_1 q'$ then $\exists q'' \cdot q \xrightarrow{a}_1 q'' \wedge q'' \overset{t}{\Rightarrow}_1 q'$.

By the inductive hypothesis, it follows that $\exists q'' \cdot q \xrightarrow{a}_1 q'' \wedge eq(q'') \overset{t}{\Rightarrow}_2 eq(q')$.

Since we know that every transition in M_1 is mirrored in M_2 on equivalent states, then we know that $\exists q'' \cdot eq(q) \xrightarrow{a}_2 eq(q'') \wedge eq(q'') \overset{t}{\Rightarrow}_2 eq(q')$.

Hence, we conclude that:
$$eq(q) \overset{at}{\Rightarrow}_2 eq(q')$$

The result follows by string induction. $\square$

Theorem 1 *Given two LTSs $M_i = (Q_i, A_i, T_i, q_{0i})$, with $i \in \{1, 2\}$, if M_2* is refined by *M_1 ($M_2 \sqsubseteq M_1$), then the language generated by M_1, $\mathcal{L}(M_1)$, is a subset of the language generated by M_2, $\mathcal{L}(M_2)$.*

Proof: To require to show that if $s \in \mathcal{L}(M_1)$ then $s \in \mathcal{L}(M_2)$.

If $s \in \mathcal{L}(M_1)$, then, by definition of $\mathcal{L}(M)$:
$$\exists q_1 : Q_1 \cdot q_{01} \overset{s}{\Rightarrow}_1 q_1$$

By applying lemma 1, we can conclude that $eq(q_{01}) \overset{s}{\Rightarrow}_2 eq(q_1)$. But since, we know that $q_{02} = eq(q_{01})$, it follows that $s \in \mathcal{L}(M_2)$. $\square$

4. INTERFACES

Interfaces [6] exploit the use of a compositional approach for state space generation. Essentially an interface represents the envorinment of a sub-expression E' in E. This interface, usually a LTS, represents the set of authorised execution sequences that can be performed by E' within the context of E. Using a projector operator one can generate a restricted LTS of E' such that useless execution sequences are cut off according to the corresponding interface.

In [6] a new projection operator is defined. This is the *semi-composition* operator which ensures that :

1. it restricts the behaviour of E' according to its environment

2. it preserves the behaviour of the initial expression when a sub-expression E' is replaced by its corresponing reduced expression.

3. it can be computed on-the-fly, i.e. can be obtained without generating the LTS of E' first.

The definition of the semi-composition operator is given in [6]. $M_1 \rrbracket_G M_2$ denotes the LTS resulting from the semi-composition of M_1 by M_2. M_2 is the interface with which M_1 is semi-composed. One should note that if M_2 is manually generated by an expert of the system or protocol being verified then semi-composition is probably going to be much more effective in reducing the states of S_1. Our work explores the possibility of *automatically* creating effective interfaces. We can then guarantee their correctness by construction.

If M_1 is composed with M_2 (not necessarily locally) and M_2' is an LTS such that $\mathcal{L}(M_2) \subseteq \mathcal{L}(M_2')$, then we can reduce M_1 to $M_1 \rrbracket_G M_2'$ without altering the overall behaviour of the overall system. We would clearly be altering the behaviour of M_1 but this is exactly what we want in terms of state reduction. Since the complexity of the semi-composition operator increases as the the number of states of the interface increases, sometimes being impossible to calculate due to the size of the interface, our aim is to balance these requirements — taking an LTS M we want to produce a smaller LTS M' satisfying $\mathcal{L}(M) \subseteq \mathcal{L}(M')$. Clearly, many solutions exist satisfying this loose requirement. In this paper we present two initial experiments in this direction.

5. AUTOMATIC GENERATION OF INTERFACES

We have so far implemented two interface generators. This section describes these algorithms. In what follows we refer to the original LTS as M_1 and its reduced LTS, the interface, as M_2

5.1 Chaos State Partition

The first interface implementation is the chaos state partition interface. The main idea is that we keep a number of states from M_1, collapsing the rest into a *chaos state*. So for example if we have M_1 with 20 states and we want to generate an interface taking in consideration only the first 10 states (traversing the LTS in breadth-first order, starting from the inital state), its reduced version M_2 would have 10+1 states. The extra state is the chaos state (we call χ) in which all the other states are grouped. Figure 1 illustrates this reduction.

Let $\bar{Q}$ be a subset of states of M_1 which will be kept in M_2. M_2 is the LTS resulting from the reduction of M_1. $\bar{Q}$ = {0,1,2,3,4,5,6,7,8,9} in the example given here.

Definition 5 *Given an LTS* $M = (Q, A, T, q_0)$ *and* $\bar{Q} \subseteq Q$, *we define the reduction of* M *to states* $\bar{Q}$ *to be* $M \rhd \bar{Q} = (Q', A', T', q_0')$ *such that* $Q' = \bar{Q} \cup \{chaos\}$, $A' = A$, $q_0' = \text{chaos}_\chi^{\bar{Q}}(q_0)$ *and* $T' = \{(\text{chaos}_\chi^{\bar{Q}}(q), a, \text{chaos}_\chi^{\bar{Q}}(q')) \mid (q, a, q') \in T_1\} \cup \{(\chi, a, \chi) \mid a \in A\}$, *where* chaos *is defined as follows:*

$$\text{chaos}_\chi^Q(q) = \begin{cases} q & \text{if } q \in Q \\ \chi & \text{otherwise} \end{cases}$$

This construction yields an LTS M' which is refined by the original LTS M.

Proposition 1 $M \rhd \bar{Q} \sqsubseteq M$

This can be shown by taking $eq(q)$ to be $chaos_\chi^{\bar{Q}}(q)$.

By theorem 1 we are guaranteed that $M \rhd \bar{Q}$ accepts a superset of the language accepted by M, and hence can be used as a replacement interface.

The Chaos Partition algorithm has been implemented using the CADP toolkit [5] traversing the LTS in breadth-first order, starting from the inital state.

Algorithm 1 Calculate $M_2 = M_1 \rhd \bar{Q}$

```
Require: m ≤ n
  n ← number of states in $M_1$
  m ← depth in breath first order of last state in $M_1$ to keep before chaos
  for $i = 0$ to $n$ do
    if i < m then
      copy state $Q_i$ from $M_1$ to $M_2$
      copy outgoing tranisitions of state $Q_i$ from $M_1$ to $M_2$
    else
      join state $Q_i$ to the chaos state in $M_2$
      copy outgoing transitions of $Q_i$ in $M_1$ to $M_2$
    end if
  end for
```

5.2 Node Behaviour State Partition

Our second implementation abandons the idea of creating a chaos state and instead moves in the direction of creating a set partition which groups together states in M_1 which can perform exactly the same set of strings of length n. With, for example, the length being 1, and there two states in M_1 which can only perform actions a and b then these two states are grouped together in one state in M_2. We currently cater only for length 1 but will deal with the general case in future work. Figure 2 illustrates the same LTS shown earlier on but this time reduced with the Node Behaviour state partition.

The state partitions created with this reduction are the following:

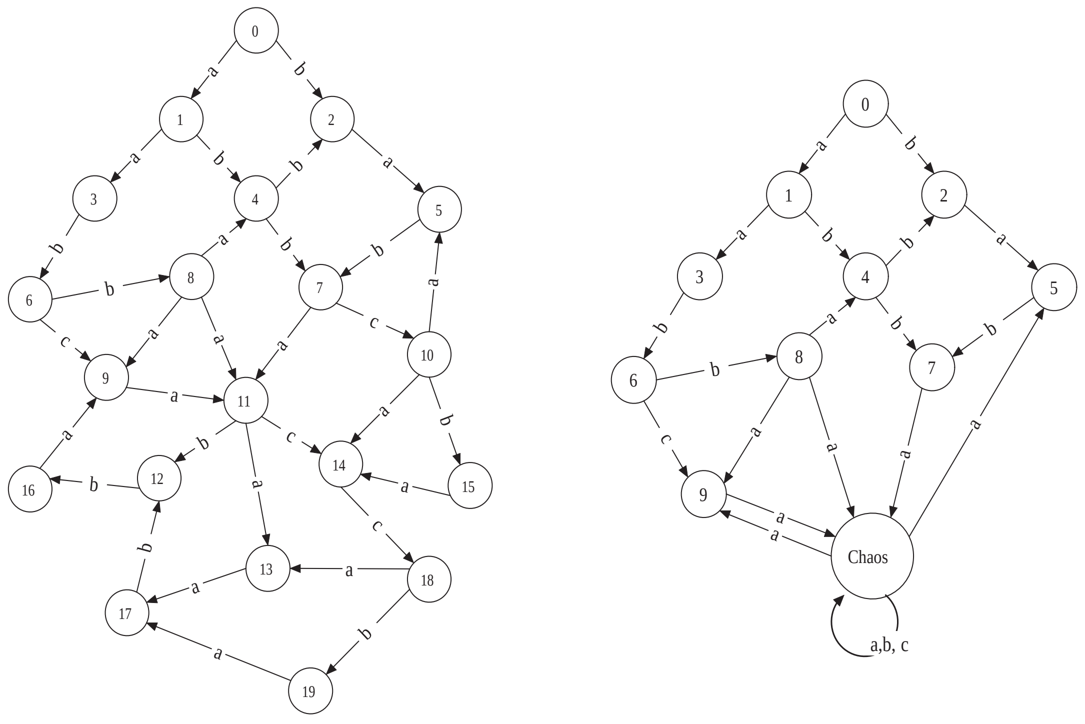

Figure 1: State reduction of a 20 state LTS using the chaos state partition

State Actions	State Partition
a,b	{0,1,10,18}
a	{2,8,9,13,15,16,19}
b	{3,4,5,12,17}
b,c	{6}
a,c	{7}
a,b,c	{11}
c	{14}

The partitions created will form the new states in M_2.

Definition 6 *Given an LTS* $M = (Q, A, T, q_0)$, *the reduction of* M *looking at trails of length 1 is defined to be* $M^{[1]} = (Q', A', T', q'_0)$ *where* $Q' = 2^A$, $A' = A$, $q'_0 = actions(q_0)$ *and* $T' = \{actions(q), a, actions(q')) \mid (q, a, q') \in T\}$.

This construction yields an LTS M' which is refined by the original LTS M.

Proposition 2 $M^{[1]} \sqsubseteq M$

With depth $= 1$ the possible states of M_2 is equal to the powerset of A. Hence function eq is defined as follows:

$$eq(q) = actions(q)$$

Interface generation using the Node Behaviour partition algorithm has also been implemented within the CADP toolkit [5]. As explained in algorithm 2 we first go through all the states in M_1 and group them in state partitions according to their outgoing transitions. These state partitions become the new states in M_2. We then create the transitions between the new state partitions to reflect the transitions present in M_1. The number of states and number of transitions in M_2 is clearly always smaller or equal to that in M_1.

6. CASE STUDIES OF COMPOSITIONAL VERIFICATION

In this section we describe some experiments carried out with the automatically generated interfaces. In order to come up with a good comparison of how these interfaces work we make use of an example listed as demo of the CADP toolkit which uses the *projector* operator discussed earlier.

We first give a very brief introduction to the CADP toolkit, then describe the rel/REL multicast protocol. We illustrate the results achieved when the sub-expressions of the protocol are restricted with our automatically generated interfaces.

6.1 Construction and Analysis of Distributed Processes (CADP)

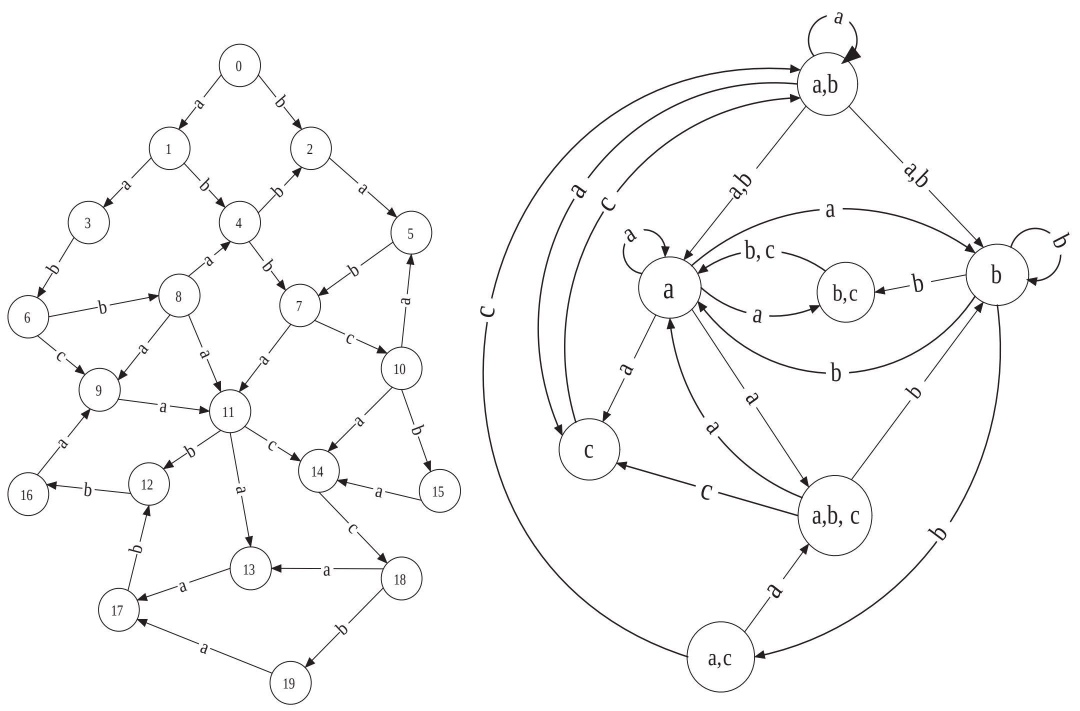

Figure 2: State reduction of a 20 state LTS when n=1

Algorithm 2 Calculate $M_2 = M_1^{[1]}$

```
n ← number of states in $M_1$
for $i = 0$ to $n$ do
  if state partition already exists which maps the behaviour of $Q_i$ then
    add $Q_i$ from $M_1$ to this state partition
  else
    create new state partition
    add $Q_i$ from $M_1$ to the newly created state partition
  end if
end for
X,Y ← set of state partitions in $M_2$
for all partitions $X_j$ in $X$ do
  for all partitions $Y_k$ in $Y$ do
    if there are states in $X_j$ and $Y_k$ such that these states were connected in $M_1$ then
      connect with the same transition as in $M_1$ partitions $X_j$ and $Y_k$
    end if
  end for
end for
```

The CADP toolkit [5] consists of a set of verification tools developed by the VASY research group[1]. The toolkit is especially useful in the design of communication protocols and distributed systems. Processes are specified in LOTOS [4] which is essentially a language for specifying CCS and CSP like processes. CADP contains amongst other tools, compilers for translating LOTOS descriptions into C code and LTSs which is then used for simulation, verification and testing purposes. The implementation of our interfaces was carried out using the CADP tools for explicit state manipulation, namely the BCG. Binary Coded Graphs (BCG) is a package which uses highly optimised representation of LTSs together with various other tools for the manipulation of these LTSs. Further information on CADP can be obtained by visiting the VASY website.

The *projector* operator within CADP implements the semi-composition operator defined in [6]. This operator is used to constrain processes within their environment. We make use this semi-composition operator to contrain processes with our automatically generated interfaces before actually applying the parallel composition on them.

6.2 The rel/REL Multicast Protocol

The rel/REL protocol [9] provides a reliable atomic multicast service between stations connected by a network. The

[1]http://www.inrialpes.fr/vasy/cadp/

LOTOS specification of the multicast protocol can be found in [6]. The rel/REL multicast service should ensure that a message sent by a transmitter is appropriately broadcasted to all receivers taking in consideration the fact that these stations may suffer from failures (i.e., they can crash).

In this paper we do not give details of how the rel/REL protocol works. We are only interested in the service provided by the protocol which need to be verified. The first property is atomicity, meaning that either all receivers get the message, or none of them will. This property is verified using temporal logic. The second property is that of causality meaning that the order of messages is preserved. This property is verified using comparison modulo safety equivalences. By using our automatically generated interfaces we would like to observe a decrease in the number of states in the intermediate LTSs composing the whole protocol.

The example considered here is composed of one transmitting station (Trans) and three receiver stations (Rec). Their parallel composition denoted by expression E (without using interfaces) is as follows:

$$((Rec2||_A Rec3)||_B Rec1)||_C Trans$$

$$\begin{array}{rcl} \text{where A} & = & \{R23, R32\} \\ \text{B} & = & \{R12, R13, R21, R31\} \\ \text{C} & = & \{RT1, RT2, RT3\} \end{array}$$

In [6] manually generated interfaces $\mathcal{I}$ are used to restrict the receiver nodes. The transmitter LTS is then used to further restrict the composed receiver stations. Figure 6.2 shows how the projector operator is used when generating the LTS of the whole system. Note that the symbol $-||$ in figure 6.2 refers to the projector opertor.

In order to calculate the decrease in number of states imposed by our interfaces we need to split up expression E in various sub-expressions. [6] splits expression E listed above in the following intermediate LTSs each of which corresponding to a generation step. Here we add $S0$ as a further intermediate step to analyse.

$$\begin{array}{lcl} S1_i & = & sem(Rec_i)]\!| \mathcal{I}_i \\ S0 & = & S1_2 ||_{\{R23,R32\}} S1_3 \\ S2 & = & S0]\!|_{\{RT2,RT3\}} sem(Tx) \\ S3 & = & (S1_1 ||_{\{R12,R21,R13,R31\}} S2) \ ]\!|_{\{RT_n\}} \ sem(Tx) \\ sem(E) & = & S3 ||_{\{RT_n\}} sem(Tx) \end{array}$$

The following table illustrates the various reductions obtained when using our interfaces. Each row indicates the number of states of the subexpression together with the number of states remaining when the subexpression is reduced up to strong bisimulation [8].

The first column shows the reductions when manually generated interfaces are used. The reductions obtained with this

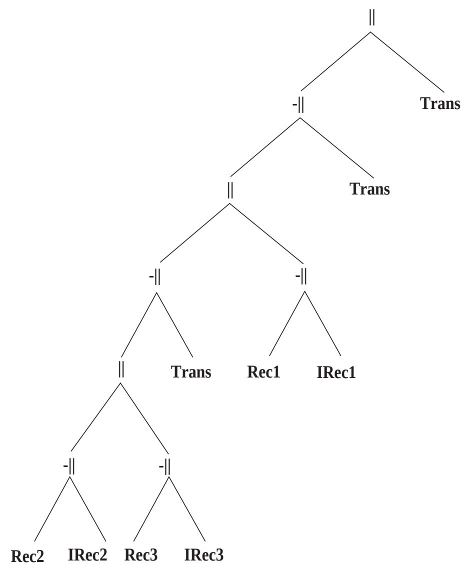

Figure 3: Computational tree of the rel/REL protocol using interfaces

method are clearly the best possible. The second column shows the sizes of the intermediate LTSs when no interfaces are used. We could only generated $S0$ in this case since the composition of $S2$ with $S1_i$ could not be generated due to the exponential growth of the composed LTSs. The third and fourth column illustrate the reductions obtained when *BFReduction* (Chaos State Partition implementation) and *TEReduction* (Node Behaviour n=1) were used.

The first row is indicative of the reductions obtained by the interfaces. When using a manually generated interface (user written) the receiver node is reduced to 1086 states. The original size of the receive node LTS is 2882 as indicated in the second column. Both BFReduction and TEReduction exploit the Transmitter LTS which is eventually composed with the Receiver nodes. An interface is generated out of this LTS and immediately applied to the Receiver node.

Both interfaces obtain a reduction in state space of the receiver node. When the receiver LTS is projected with the interface created by the BFReduction algorithm the receiver node is reduced to 2106 states. With the TEReduction algorithm we only manage to reduce the receiver LTS to 2700. It is interesting to note however that when both LTSs are reduced up to strong bisimulation both interface reduced receiver nodes get much smaller. Further work needs to be carried out in order to understand why this is the case.

The composition of two receiver nodes yields an LTS of 3,789,678 states when no interfaces are used. When using our interfaces we manage to get a reduction to 1,621,640 states with the BFReduction interface and a reduction to

Expression	Manual Interfaces	No Interfaces	BFReduction	TEReduction
$S1_2$	1086/369	2882/127	2106/149	2700/127
$S0$	229767/31020	3789768/15139	1621640/20939	3313694/15139
$S2$	149366/30171	na	1621640/20939	3313694/15139

3,313,694 states with the TEReduction interface. The composition if the receiver nodes on which the manual interface was used goes up only to 229,676 states. This clearly shows the difference a manually generated interface has on the composition of these LTS.

With these interfaces we can only generate up to $S2$. Without interfaces not even $S2$ could be generated. We are currently investigating some of the results obtained with these interfaces, whilst trying to improve on their implementations. The two implementations which we curently have serve only as proof of concept for our work. We clearly need to come up with more 'intelligent' interface generators.

7. FUTURE RESEARCH

This paper discusses only our initial attempts at interface generation. Through these interface we want to reduce the intermediate state space needed for the verification of systems. So far we have implemented two interface generators which only give us some indications on the way forward. The current interfaces have some pitfalls which we shall be addressing shortly. For example, TEReduction with a depth of 1, groups together in one state all those states which are able to perform only the internal action τ. This can be partially avoided simply by increasing the depth to some other value greater than 1.

Further experiments also need to be carried out in order to better asses the behaviour of the current interfaces. We shall be working on the verification of various other protocols using interfaces. Ultimately we would like to obtain a general purpose interface generator which is able to effectively reduce the number of intermediate states necessary.

8. REFERENCES

[1] E.M.Clarke, O.Grumberg, and D.A.Peled. *Model Checking*. The MIT Press, Cambridge, Massachusettes, 1999.

[2] Frederic Lang. Refined interfaces for compositional verification. In E. Brinksma, editor, *International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2006*, pages 239–258, Enschede, The Netherlands, 2006. Springer Verlag, LNCS 1217.

[3] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[4] ISO/IEC. LOTOS. A formal description technique based on the temporal ordering of observational behaviour. international standard 8807, international organisation for standardization - information processing systems. *Open Systems Interconnection*, 1989.

[5] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[6] J.P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 239–258, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.

[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[8] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[9] P D Ezhilchelvan and S K Shrivastava. rel/rel: a family of reliable multicast protocols for distributed systems. *Distributed Systems Engineering*, 1(5):323–331, 1994.