Automatic Interface Generation for Compositional Verification

Sandro Spina, Gordon Pace, Frederic Lang

Automatic Interface Generation for Compositional Verification

Sandro Spina, Gordon J. Pace and Frédéric Lang

[1] Department of Computer Science, University of Malta
[2] INRIA Rhône-Alpes, Grenoble, France

Abstract. Compositional verification, the incremental generation and composition of the state graphs of individual processes to produce the global state graph, tries to address the state explosion problem for systems of communicating processes. The main problem with this approach is that intermediate state graphs are sometimes larger than the overall global system. To overcome this problem, interfaces [JL97], and refined interfaces [Lan06], which take into account a system's environment have been developed. The number of states of these interfaces plays a vital role in their applicability in terms of computational complexity, which is proportional to the number of states in the interface. The direct use of complete subcomponents of the global system as interfaces, thus usually fails, and it is up to the system designer to describe smaller interfaces to be used in the reduction. To avoid having to verify the correctness of such manually generated interfaces, we propose automatic techniques to generate correct interfaces. The challenge is to produce interfaces small in size, yet effective for reduction. In this paper, we present techniques to structurally produce language over-approximations of labelled transition systems which can be used as correct interfaces, and combine them with refined interfaces. The techniques are applied to a number of case-studies, analysing the trade-off between interface size and effectiveness.

1 Introduction

Over the past years, the verification of computational systems has taken up considerable interest and support. Techniques in both symbolic and enumerative strategies made many advances in terms of what can be verified. However, the main problem still remains that of the state space explosion arising when composing together components of a system. Many solutions have been proposed and used to counter the problem. Our work focuses on enumerative compositional verification in which the state graph of a system made up of a number of processes, is generated incrementally. In compositional verification, instead of generating the global system, the state graphs of the constituent processes are individually generated, reduced up to some equivalence relation which preserves global system behaviour, and incrementally composed together until the global system is obtained. The problem with this approach is that intermediate state graphs are sometimes larger than the global system, simply because the behaviour of the processes is not constrained by the other processes which have not

yet been composed. The challenge is thus one of constraining the state graph of these intermediate processes without losing anything from their behaviour within the global system.
Interfaces [GSL96,GS91,CK93], provide a means of overcoming this problem. An interface, essentially represents part of the environment of a particular subcomponent of the global system. This environment essentially poses behavioural restrictions imposed on the subcomponent through synchronisation. For instance, in system S, composed of processes P_1 to P_n, processes P_2 to P_n (or abstractions of them) can be seen as interfaces of P_1. Krimm and Mounier [JL97] use interfaces to constrain on-the-fly the state graph of the process being generated and introduce the semi-composition operator which is implemented as the *Projector* tool in the CADP toolkit [GLMS07] which we are using[3]. In their approach, the interface represents the only possible interactions between the process and its environment. Semi-composition is used to reduce the process states and transitions which will anyway not be reachable in the global system given the knowledge in the interface.
Refined interfaces [Lan06] extend these interface techniques to take into account whole families of concurrent processes as an environment. Lang [Lan06] proposes a technique to automatically generate interface processes from a subset of processes that are composed with the process of which we want to produce the state graph. In a number of case studies it has been shown that refined interfaces allow a better reduction of the process state graph we want to generate. The higher the number of processes used to produce a refined interface, the better this interface would describe the environment of a process. However, the higher the number of processes used the higher the number of states the interface will end up with. In both techniques, the computational complexity of the reduction for a given interface is proportional to the number of states in the interface. The direct use of complete processes of the global system as interfaces, may thus fail, and it is is up to the system designer to describe smaller interfaces to be used for the reduction. However, when using an abstraction of the communicating processes, it is important to guarantee that the abstraction is, in fact, correct. Techniques to verify the correctness of an interface have been proposed [JL97], but come at a price. Furthermore, the design of these interfaces requires expert knowledge of the system being verified in order to be effective. In this paper, we propose techniques for the automatic construction of interfaces which are guaranteed to be correct by construction. The techniques are applied to a number of case studies, analysing the trade-off between interface size and effectiveness.

2 Background

2.1 Preliminary Definitions

The behaviour of a process can be modelled as a labeled transition system (LTS), consisting of a set of states and a labeled transition relation between states.

[3] Available from `www.inrialpes.fr/vasy/cadp`

Each transition describes the execution of the process from a current state with a particular instruction (label).
In what follows $\mathcal{A}$ is the global set of labels and 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 $A \cup \{\tau\}$. For a set of labels A, A^* will represent the set of finite sequences on A.

Definition 1 (LTS). A Labeled Transition System (LTS, for short) is a quadruple $M = (Q, A, T, q_0)$ 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 will use the notation $q \xrightarrow{a}_T q'$ to mean $(q, a, q') \in T$, and $q \overset{a_1 a_2 \ldots a_n}{\Rightarrow}_T q'$ to mean that there exist states $q_1, q_2 \ldots q_n$ with $q_1 = q$ and $q_n = q'$ such that for all i, $q_i \xrightarrow{a_{i+1}}_T q_{i+1}$. We abuse the notation to allow a^* to appear in the string, indicating any number of repetitions of a. We will also leave out T when it is clear from the context. Given a state q, we will use the notation incoming(q) to denote the set of immediate actions which can be performed just before q (incoming(q) $= \{a \mid \exists q' \cdot q' \xrightarrow{a} q\}$), and similarly outgoing(q) to indicate the set of immediate outgoing actions from q. We now define the language generated by a LTS from a particular state $p \in Q$.

Definition 2 (Language generated by an LTS). Given an LTS M, such that $M = (Q, A, T, q_0)$ and $q \in Q$, the (observable) language starting from q in M, written $\mathcal{L}(M)$ is defined as follows $\{\ w \mid \exists q' \cdot w = a_1 a_2 \ldots a_n \wedge q_0 \overset{\tau^* a_1 \tau^* \ldots a_n \tau^*}{\Rightarrow} q'\}$.

Two LTSs can be composed together with the parallel composition operator synchronising on a set of labels common to both LTSs.

Definition 3 (Parallel Composition, Hiding). Let $S_i = (Q_i, A_i, T_i, q_{0_i})$ ($i = 1, 2$) be two LTSs, and $G \subseteq \mathcal{A}$. The *parallel composition* of S_1 and S_2 over G, written $S_1 \parallel_G S_2$, models the concurrent execution of S_1 and S_2 with forced synchronization on G, and is defined as the LTS $(Q, A_1 \cup A_2, T, (q_{01}, q_{02}))$, where Q and T are the smallest sets satisfying both $(q_{01}, q_{02}) \in Q$ and the following rules:

$$\frac{(q_1, q_2) \in Q,\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2',\ a \in A}{(q_1', q_2') \in Q,\ (q_1, q_2) \xrightarrow{a}_T (q_1', q_2')}$$

$$\frac{(q_1, q_2) \in Q,\ q_1 \xrightarrow{a}_{T_1} q_1',\ a \notin A}{(q_1', q_2) \in Q,\ (q_1, q_2) \xrightarrow{a}_T (q_1', q_2)} \qquad \frac{(q_1, q_2) \in Q,\ q_2 \xrightarrow{a}_{T_1} q_2',\ a \notin A}{(q_1, q_2') \in Q,\ (q_1, q_2) \xrightarrow{a}_T (q_1, q_2')}$$

Note that, by construction, the states belonging to Q are reachable. A state q_1 of S_1 (respectively q_2 of S_2) is said *reachable* in $S_1 \parallel_G S_2$ if there is a state (q_1, q_2) in $S_1 \parallel_G S_2$. Similarly, a transition $q_1 \xrightarrow{a} q_1'$ of S_1 (respectively $q_2 \xrightarrow{a} q_2'$ of S_2) is said *reachable* in $S_1 \parallel_G S_2$ if there is a transition $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ in $S_1 \parallel_G S_2$. The system `hide` A `in` S_1, corresponds to the LTS $(Q_1, A_1 \setminus A, T_1', q_{01})$, where T_1' is defined as follows:

$$\frac{q \xrightarrow{a}_{T_1} q',\ a \in A}{q \xrightarrow{\tau}_{T_1'} q'} \qquad \frac{q \xrightarrow{a}_{T_1} q',\ a \notin A}{q \xrightarrow{a}_{T_1'} q'}$$

Consider for instance modelling a communication protocol with an LTS T representing a transmitter, and LTS R representing a receiver's behaviour. $T \parallel_G R$ is the LTS generated by concurrently executing LTSs T and R synchronising on the labels in G. Any transitions not in G are performed independently by the two LTSs.

2.2 Compositional Verification Using Interfaces

The semi-composition operator takes an LTS M_1, and reduces its behaviour up to another LTS M_2 communicating on an alphabet G. The effect is that of removing sequences of actions which M_2 would never allow M_1 to engage in. Formally, the definition of semi-composition is the following [Lan06]:

Definition 4 (Semi-Composition). Let $M_i = (Q_i, A_i, T_i, q_{0i})$ $(i = 1,\ 2)$ be two LTSs, $G \subseteq \mathcal{A}$, and $(Q', A', T', q_0') = M_1 \parallel_G M_2$. The *semi-composition of* M_1 *and* M_2, written $M_1 \between_G M_2$, is the LTS (Q, A_1, T, q_{01}), where $Q = \{q_1 \mid (q_1, q_2) \in Q'\}$ and $T = T_1 \cap \{(q_1, a, q_1') \mid (q_1, q_2) \xrightarrow{a}_{T'} (q_1', q_2')\}$. G is called the *synchronization set* and the pair (G, M_2) is called the *interface*. We say that an action $a \in A_1$ is *controlled* by the interface (G, M_2) if $a \in G$.

From this definition it is clear that the resultant LTS is a sub-LTS of S_1. This guarantees that semi-composition never increases the number of states of its first operand. We also know from [JL97] that one can replace the expression $M_1 \parallel_G M_2$ with $(M_1 \between_G M_2) \parallel_G M_2$ without loosing any temporal properties of the system $M_1 \parallel_G M_2$. Semi-composition also guarantees that $M_1 \between_G M_2$ is branching bisimilar to $M_1 \between_G M_2'$ if $\mathcal{L}$(hide $(\mathcal{A} \setminus G)$ in M_2) = $\mathcal{L}$(hide $(\mathcal{A} \setminus G)$ in M_2'). This means that one can first hide uncontrolled actions and then minimize the LTS modulo a relation preserving observable traces (for instance branching or safety equivalence reductions). Minimization of the interface is very important since this reduces the cost of semi-composition, the complexity of which is the same as parallel composition.
Furthermore, these results can be extended to work with language inclusions:

Conjecture 1 *Given* LTS*s* M_1 *and* M_1'*, such that* $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_1')$*, then for any* LTS M_2*, and action list* G*,* $(M_2 \between_G M_1') \parallel_G M_1$ *is branching bisimilar to* $M_2 \parallel_G M_1$*;*

The proof of this conjecture is not given for the time being, but will be included in future work.

2.3 Refined Interfaces

In our work we make use of refined interface generation to produce interfaces which take into account a subset of the processes neighbouring the process whose state graph we want to generate. Generation of refined interfaces is based on the network of LTSs concurrent model in which the composition hierarchy is completely flattened. The network of LTSs model is more general than the parallel composition operator defined in the previous section, and the parallel composition, renaming, hiding and cutting operators from many process algebras can be translated into networks of LTSs [Lan05]. We first define *vectors*, from which networks of LTSs are composed.

Definition 5 (Vectors). A *vector* of *length* n over a set S is an element of S^n, written $\boldsymbol{t}$ or $(t_1, \ldots, t_n)$. For $1 \leq i \leq n$, $\boldsymbol{t}[i]$ denotes the ith element t_i of $\boldsymbol{t}$, and $\boldsymbol{t}[i \leftarrow t'_i]$ represents a copy of $\boldsymbol{t}$ where $\boldsymbol{t}[i]$ is replaced by t'_i. Given $t \in S$, we write t^n the vector of length n such that $(\forall 1 \leq i \leq n)\ t^n[i] = t$. Given $I \subseteq \{1, 2 \ldots n\}$, the *projection* $\boldsymbol{t}_{\downarrow I}$ is defined by: $\boldsymbol{t}_{\downarrow I} = (\boldsymbol{t}[k_1], \ldots, \boldsymbol{t}[k_m])$ where $\{k_i \mid 1 \leq i \leq m\} = I$ and $(\forall i < j)\ k_i < k_j$.

Definition 6 (Network of LTSs). Let $\bullet \notin \mathcal{A}_\tau$ be a special symbol denoting that a particular LTS has no role in a given synchronization. A *synchronization rule* is a pair $(\boldsymbol{t}, a)$, where $\boldsymbol{t}$ is a vector over $\mathcal{A}_\tau \cup \{\bullet\}$ (called a *synchronization vector*) and $a \in \mathcal{A}_\tau$. The components $\boldsymbol{t}$ and a are called respectively the left- and right-hand sides of the synchronization rule. A *network of* LTS*s* (or simply *network*) N of *dimension* $n > 0$ is a pair $(\boldsymbol{S}, V)$ where $\boldsymbol{S}$ is a vector of LTSs of length n and V is a set of synchronization rules, whose left-hand sides are all of length n. Each left-hand side $\boldsymbol{t}$ expresses a synchronization constraint on $\boldsymbol{S}$, all components $\boldsymbol{S}[i]$ where $\boldsymbol{t}[i] \neq \bullet$ having to take a transition labeled respectively $\boldsymbol{t}[i]$ altogether so that a transition labeled with the corresponding right-hand side a be generated in the product. More formally, let $\boldsymbol{S}[i] = (Q_i, A_i, T_i, q_{0_i})$, $(1 \leq i \leq n)$. To $N = (\boldsymbol{S}, V)$ corresponds an LTS $(Q, A, T, \boldsymbol{q_0})$, written $sem(N)$ or $sem(\boldsymbol{S}, V)$, such that $A = \{a \mid (\boldsymbol{t}, a) \in V\}$, $\boldsymbol{q_0} = (q_{0_1}, \ldots, q_{0_n})$, and Q and T are the smallest sets satisfying both $\boldsymbol{q_0} \in Q$ and:

$$\frac{\boldsymbol{q} \in Q,\ (\boldsymbol{t}, a) \in V,\ (\forall 1 \leq i \leq n)\ (\boldsymbol{t}[i] = \bullet \wedge \boldsymbol{q'}[i] = \boldsymbol{q}[i]) \vee \boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{q'}[i]}{\boldsymbol{q'} \in Q,\ (\boldsymbol{q}, a, \boldsymbol{q'}) \in T}$$

Note that, by construction, the states that belong to Q are reachable. Synchronization rules must obey the following *admissibility* properties, which forbid cutting, synchronizations and renaming of τ transitions and therefore ensure that safety equivalence and stronger relations (e.g., observational, branching, and strong equivalences) are congruences for networks of LTS [Lan05]:

$$((\exists 1 \leq i \leq n)\ \tau \text{ is reachable in } \boldsymbol{S}[i]) \implies (\exists (\boldsymbol{t}, \tau) \in V)\ \boldsymbol{t}[i] = \tau$$
$$(\forall (\boldsymbol{t}, a) \in V)\ ((\exists 1 \leq i \leq n)\ \boldsymbol{t}[i] = \tau) \implies (a = \tau \wedge (\forall 1 \leq j \leq n \backslash \{i\})\ \boldsymbol{t}[j] = \bullet)$$

The refined interface generation technique as defined and described in [Lan06] is used to generate interfaces from a network of LTSs. In our work we use the

EXP.OPEN 2.0 tool [Lan05] which allows for the description of concurrent systems as a composition of LTSs, using either synchronisation vectors, or standard parallel composition, hiding, renaming and cut operators from several process algebras. The following syntax describes the generation of a refined interface $M_{1..n}$, from the synchronisation vectors of LTSs M_1 to M_n.

$$\begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_n \end{bmatrix} \Rightarrow M_{1..n}$$

Clearly, the higher the number of processes used to generate a refined interface, the higher the number of states this interface will end up with. This means that one can end up with a very good description of the environment of a process which however cannot be used because of its size. The ideal scenario is a trade off between the size of the interface and its effectiveness in representing the environment of a process.

3 LTS Reductions

As discussed in section 2, one can constrain the generation of an LTS state graph by using other LTSs in the network as interfaces, even if, in most cases the problem is intractible due to the large size of the base LTSs. However, conjecture 1 gives us the option to use language over-approximations to gain access to smaller automata. In this section, we propose a number of LTS reduction techniques especially designed to work for effective interface generation. Rather than design each technique independantly of each other, we provide an infrastructure to reason about a class of reduction techniques, enabling us to present a number of solutions guaranteed to be correct.
We start this section by presenting the concept of an LTS structural reduction which guarantees language inclusion.

Definition 7 (LTS Reduction). *We say that M_2 is a structural reduction of M_1 with respect to a total function $eq \in Q_1 \rightarrow Q_2$, written $M_2 \sqsubseteq_{eq} M_1$, if the following conditions hold: (i) $A_1 = A_2$; (ii) $Q_2 \subseteq Q_1$; (iii) $q \xrightarrow{a}_1 q'$, implies that, $eq(q) \xrightarrow{a}_2 eq(q')$; and (iv) $q_{02} = eq(q_{01})$.*

We simply say that an LTS M_2 is a reduction of another LTS M_1, written $M_2 \sqsubseteq M_1$, if there exists a total function eq such that $M_2 \sqsubseteq_{eq} M_1$. Also, we say that M_2 is the reduction of M_1 with respect to eq, if M_2 is the (unique) solution satisfying $M_1 \sqsubseteq_{eq} M_2$.

Proposition 1. *$\sqsubseteq$ is a reflexive, transitive relation over* LTS*s.*

The property of structural reduction we will be using most is that it guarantees language inclusion:

Lemma 1. Given two LTSs $M_i = (Q_i, A_i, T_i, q_{0i})$, with $i = 1, 2$, related with a total function eq, then $q \stackrel{s}{\Rightarrow}_1 q'$ implies that $eq(q) \stackrel{s}{\Rightarrow}_2 eq(q')$

Proof: The proof follows directly by induction over string s. □

Language inclusion follows directly from the previous lemma and the fact that $eq(q_{02}) = eq(q_{01})$:

Theorem 1. *Given two* LTS*s* $M_i = (Q_i, A_i, T_i, q_{0i})$*, with* $i = 1, 2$*, if* $M_2 \sqsubseteq M_1$*, then* $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$*.*

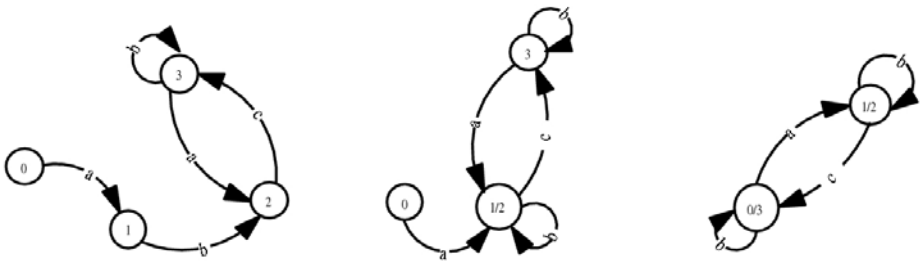

Fig. 1: LTS progressively approximated from left to right (S, S' and S'' respectively)

Consider a small system, for the purposes of illustration, whose behaviour is modelled by the first LTS S as depicted in figure 1. From this LTS we know that the system is capable of first performing a, after which it can only perform b and then sequences of $c\ b^\star a$. This LTS has four states. S' is the LTS generated after performing a reduction on S, where $eq(0) = 0$, $eq(1) = eq(2) = 1/2$ and $eq(3) = 3$. This guarantees that $\mathcal{L}(S) \subseteq \mathcal{L}(S')$. In S' we still know that the system is initially only capable of performing a. However we've now lost the information which said that *only* b can be performed from state 1. In S' after the initial a the system can now produce either a sequence of b or c. Finally S'' is produced from S' using function eq, with $eq(0) = eq(3) = 0/3$ and $eq(1/2) = 1/2$. Thanks to theorem 1 we know that $\mathcal{L}(S') \subseteq \mathcal{L}(S'')$. In this third language over approximation we lose further information from the behaviour of S. From the initial state, the system can now perform either a or b. However we still know that from the second state following an a transition, we can perform only either a sequence of b or just one c. From this simple example, one can easily see that applying the functional composition of the two eq functions applied, one gets that $S \sqsubseteq S''$.

We are currently exploring the definition of different structural reduction techniques for the generation of interfaces. In the following sections, we describe a number of such techniques, which are then used in the case studies in section 4.

3.1 Chaos

The most straightforward structural reduction technique is that of keeping a number of states of the original system (including the initial state), and coalescing the remaining states into one which can behave chaotically, by emulating any of the remaining states:

Definition 8. *Given an* Lts M*, and a subset of its states* Q_{in}*,* $CH_M[Q_{in}]$ *is defined to be the reduction of* M *with respect to function* eq*, defined as follows* ($\chi \notin Q$)*:*

$$eq(q) = \begin{cases} q & \text{if } q \in Q_{in} \\ \chi & \text{otherwise} \end{cases}$$

Typically, when implemented, this reduction is applied by exploring the state-space of M progressively (in a breadth-first manner), for a fixed depth, beyond which everything else is collapsed together:

Definition 9. *Given an* Lts M*, we define* $CH_M[n]$ *to be* $CH_M[Q_{in}]$*, where* Q_{in} *is the set of states reachable in no more than* n *steps from the initial state.*

3.2 Partition Based on Action Capability Similarity

The main idea behind this reduction technique is that of creating a state partition which groups together states that exhibit a similar local behaviour. States can be compared by checking that they have the same outgoing transitions.

Definition 10. *Given an* Lts M*, we define* $TR_M[out]$ *to be the reduction of* M *with respect to function* $eq = outgoing$*.*

In this manner, for each set of possible outgoing actions, we coalesce all states with that particular capability into one state, and abstract transitions accordingly. Similarly, $TR_M[inout]$ looks at both incoming and outgoing actions:

Definition 11. *Given an* Lts M*, we define* $TR_M[inout]$ *to be the reduction of* M *with respect to function* $eq(q) = (incoming(q), outgoing(q))$*.*

A weaker version of comparing outgoing actions, is to identify a state with a set of outgoing actions with another state with a superset of those actions. Let the maximal outgoing transition sets of an Lts to be the sets of action capabilities of states for which no state can perform a superset of:

$$\{A \mid (\exists q \mid A = \text{outgoing}(q)) \wedge (\neg\exists q \mid A \subset \text{outgoing}(q))\}$$

Definition 12. *Given an* Lts M*, with maximal outgoing transition sets* S*, we define* $TR^{\subseteq}_M[out]$ *to be the reduction of* M *with respect to some function* eq *satisfying* $eq(q) \in \{A \mid A \in S \wedge outgoing(q) \subseteq A\}$*.*

Clearly, in an implementation, eq has to be fixed. The most straightforward implementation, when performing the analysis on-the-fly, is to use the implicit enumeration induced when traversing the state space, and take the first solution to the constraint. Note that the computational complexity of creating the state partition is linear in the number of states of the interface we want to reduce if a hash map is used to store state partitions.

3.3 Partition Based on Label Similarity

Finally, we present a third reduction technique, similar to action capabilities, but comparing actions up to an equivalence relation, which is typically weaker than equality.

Definition 13. *Given an* LTS M*, and an equivalence relation on actions* $\approx$*, we define* $LS^{\approx}_{M}[out]$ *to be the reduction of* M *with respect to function eq defined as follows:*

$$eq(q) = \{a \mid \exists a' \cdot a' \in outgoing(q) \wedge a' \approx a\}$$

Note that $TR_M[out]$ is simply a special case of this, with $TR_M[out] = LS^{=}_{M}[out]$. Typically, in a system, labels are strings — for example, when translating a language such as LOTOS into an LTS, one gets labels consisting of a string describing (amongst other things) the gate over which the communication takes place. For this reason, partitioning labels based on equality of prefixes of labels provides a straightforward equivalence relation. For example, consider the LTS with the following eight labels:

$$\{AAA,\ AAB,\ ABA,\ BAA,\ ABB,\ BAB,\ BBA,\ BBB\}$$

Matching the first two symbols of the label strings, we can create a label partition of four classes, partitioning the labels as follows:

$$\{\ \{AAA,\ AAB\},\ \{ABA,\ ABB\},\ \{BAA,\ BAB\},\ \{BBA,\ BBB\}\ \}$$

Definition 14. *Given an* LTS M*, we define* $PREFIX^{n}_{M}[out]$ *to be* $LS^{\approx}_{M}[out]$*, where* $\approx$ *is defined as follows:*

$$l_1 \approx l_2 = first_n(l_1) = first_n(l_2)$$

Note that $\text{first}_n(s)$ gives the first n symbols in string s.
Since we usually would rather put a limit on the number of classes in the partition, we define $PF^{n}_{M}[out]$ to be $\text{PREFIX}^{i}_{M}[out]$, maximising i, such that $\approx$, the partition used contains no more than n label classes.

4 Case Studies

4.1 Using LTS Reductions with Refined Interfaces

LTS reduction techniques are well suited to be combined with refined interface generation, since the size of a refined interface sometimes makes it impossible to use in practice. Since in our LTS language over-approximations we guarantee that our interfaces include the traces of the environment, we can replace $M_1 \parallel_G M_2$ with $M_1 \parallel_G M_2'$, where $M_2 \sqsubseteq M_2'$.
Consider for instance, the composition expression $E_1 = (M_1 \parallel_{G1} M_2) \parallel_{G2} (M_3 \parallel_{G3} M_4))$. If the number of states in M_1, M_2 and M_3 is small enough, we can generate a refined interface for M_4 from their LTS. If this interface is not small enough such that it can be used with the semi-composition operator,

we can reduce the interface and use the resulting over-approximated interface in order to constrain the generation of M_4.

$$\begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix} \Rightarrow M_{123} \Rightarrow \textit{reduction of } M_{123}$$

In general, the higher the number of LTSs involved in the creation of a refined interface the better that interface would be at representing environment constraints. However, sometimes, the generation of a refined interface from the state graphs of M_1, M_2 and M_3 might itself not be possible due to the individual sizes of the constituent LTSs meaning that we would either produce a refined interface from two of the environments, or generate a full refined interface over reductions on the individual environments:

$$\begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix} \Rightarrow \begin{bmatrix} \textit{reduction of } M_1 \\ \textit{reduction of } M_2 \\ \textit{reduction of } M_3 \end{bmatrix} \Rightarrow \textit{reduction of} M_{123}$$

After generating the refined interface from the reduced LTSs, one can, depending on the size of M_{123}, apply another reduction technique on M_{123}. Clearly the more language over-approximations applied to an interface the more generic the interface becomes. In our experiments we make use of this procedure to verify the effectiveness of our LTS reduction techniques in the generation of an ODP trader.

4.2 Open Distributed Processing Trader

In this section we describe the experiments carried out on the generation of an Open Distributed Processing (ODP) trader. ODP is an ISO standard (International Standard 10746, ISO — Information Processing Systems, Geneve, 1995) whose purpose is to serve as a reference model for distributed processing. The ODP framework which has been modelled in the LOTOS process algebra consists of one trader (implementing ODP) which communicates with a number of objects. These objects can either be clients, servers or both of particular services. The trader LTS consists of roughly one million states when generated on its own without environment constraints. In the experiments described here we make use of three objects to generate the refined interface for the trader. Table 1 illustrates the results achieved while incrementally increasing the complexity of the objects (by allowing them to offer and request more services).
The interfaces are generated using the EXP.OPEN 2.0 (with the -interface option) tool available with the CADP toolkit. The full description and specification of the case study can be found on the CADP website[4]. The first trader state graph is generated through the normal process of first creating a refined interface out of the objects and then using this interface in semi-composition with the generation of the trader (*Trader* in table 1). This should clearly give the best

[4] Demo 37 at http://www.inrialpes.fr/vasy/cadp/demos.html

Table 1: ODP with three objects and one trader

Obj1 ($TR[out]$)	Obj2 ($TR[out]$)	Obj3 ($TR[out]$)	Interface (Safety R)	$Trader$ St / Tr	Interface ($TR^{\subseteq}[out]$)	$Trader_{TR}$ St / Tr
256 (122)	64 (50)	64 (50)	901K (17K)	1024 / 13K	141K (7106)	1024 / 13K
256 (122)	128 (83)	96 (74)	2.6M (48K)	2048 / 30K	242K (1088)	2048 / 41K
128 (84)	128 (54)	384 (193)	3M (96K)	2048 / 20K	553K (1079)	2048 / 36K
384 (187)	256 (123)	192 (107)	15M (∞)	∞	2.4M (11676)	8192 / 165K

possible reduction since the interface is giving the exact picture of the trader environment, but is also the most expensive. We then test how reduction TR perfoms, by first applying $TR[out]$ to the LTSs of the objects. A refined interface is then generated out of these over-approximated objects. This refined interface is further reduced by applying $TR^{\subseteq}[out]$, and is then used in semi-composition with the generation of the trader LTS ($Trader_{TR}$ in table 1).
The final ODP experiment documented here is composed of three objects with a complexity (measured in number of states) of 384,256 and 192 states respectively. If no reductions are applied to these objects, a refined interface of 15M states is produced. A state graph of this size cannot be used as an interface. On the other hand, we are able to produce a refined interface of 2.5M states when the objects are over-approximated (using reduction $TR[out]$) prior to the generation of the refined interface. $TR^{\subseteq}[out]$ is then applied on this interface to produce a final interface of 11,676 states. This interface manages to contrain the generation of the Trader state graph to 8,192 states. This shows that (in this particular case) as the complexity of the objects increases, without applying LTS reduction techniques, we would not be able to produce a refined interface out of three objects.

4.3 Directory Based Cache Coherency Protocol

The second case study describes a standard cache coherency protocol for a multiprocessor architecture. The system consists of five agent processes composed in parallel with a directory process. The protocol specification guarantees the coherency of the cache maintained on the directory across the five agents that are concurrently writing to it. Table 2 reports on the time and memory usage results achieved when generating this directory based cache coherency protocol. The LOTOS specification is available online[5].
The standalone directory state graph consists of one million states and 40 million transitions. Its reduction modulo strong bisimulation produces a LTS of 2,862 states and 1,132,544 transitions. In this experiment we first generate a refined interface from the five agents which are executing in parallel with the remote directory. This interface consists of a LTS of 1.8 millions states and 14 million transitions. When reduced up to branching equivalence we get a directory interface of 2,560 states and 40,576 transitions.

[5] Demo 28 at `http://www.inrialpes.fr/vasy/cadp/demos.html`

Table 2: Directory-based cache coherency protocol with five Agents and one Directory

Generation Technique	Interface Size	Directory Size	Time	Memory
	(States/Trans)	(States/Trans)		
Safety Reduction	48 / 1008	49 / 278	5min 24sec	82Mb
CH[1]	1 / 14	50 / 350	1 sec	1Mb
TR[out]	56 / 1355	49 / 278	2 sec	1Mb
$PF^8[out]$	36 / 612	49 / 292	2min 10sec	41Mb

Since interfaces can be reduced up to safety without changing their behaviour, we apply different interface generation techniques and compare them with respect to time and memory consumptions against the standard safety equivalence reduction. The reduction of the full automaton has 48 states, but takes 5 minutes 24 seconds to generate. The most significant result is that achieved with $TR[out]$, where the interface is generated and reduced in only two seconds, with only 8 additional states. It is also important to note that with this reduced interface, we generate exactly the same directory state graph as with the full interface. We use CH[1] in order to check how much of the reduction is due to the absence of labels (which are otherwise present in the state graph we want to generate) in the interface. In this particular case we notice that most of the directory reduction is induced simply by labels which are absent in the interface. When applied with semi-composition we get a 50 state, 350 transitions directory. The main purpose of this case-study, however, is to show that different reduction techniques produce different results when applied with semi-composition. In this particular case, the effectiveness of the interface is measured on the number of transitions which are blocked in the generation of the directory. Here $PF^8[out]$ produces 292 transitions which is slightly less than the chaos interface which produces 350.

4.4 Experiments on the VLTS Benchmark Suite

Finally, we report on two benchmarks which have been set up to measure the effectiveness of LTS reduction techniques. We make use of the VLTS (Very Large Transition Systems) benchmarks[6] state graphs which have been obtained from various case studies about the modelling of communication protocols and concurrent systems. Many of these case studies correspond to real life, industrial systems.
In both benchmarks, the original VLTS graphs are used to create their own interfaces. For both, the interface is created by randomly removing some of the transitions from the original VLTS graph and miniminizing the graph modulo branching equivalence. In tables 3 and 4, the first column indicates the technique used to generate the interface which is then used (using the PROJECTOR tool) in order to constrain the generation of the VLTS graph. The second column

[6] http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html

Table 3: vasy_164_1619.bcg VLTS state graph with 992 states

Generation Technique	Interface Size	% Reduction	Projected Lts	% Reduction
Safety Reduction	377	0	375 / 924	62
$PF^2[out]$	318	16	375 / 924	62
$TR[inout]$	338	11	641 / 338	35
CH[1]	1	100	992 / 3456	0

Table 4: vasy_564_13561.bcg VLTS state graph with 3577 states

Generation Technique	Interface Size	% Reduction	Projected Lts	% Reduction
Safety Reduction	2792	0	664 / 2809	81
$TR[inout]$	2274	19	903 / 4073	75
$TR[out]$	266	91	903 / 4076	75
$PF^10[out]$	1732	38	664 / 2809	81
$PF^6[out]$	1460	48	664 / 2809	81
$PF^4[out]$	1267	55	664 / 2809	81
$PF^2[out]$	853	70	666 / 2811	81
CH[1]	1	100	2683 / 12140	0

shows the interface size in number of states, while the third column indicates the percentage reduction in number of states of the interface. The fourth and fifth columns show, respectively, the number of states (and transitions) of the projected VLTS state graph and the percentage state reduction of the projected Lts. In the first benchmark, $PF^2[out]$ generates the same projected VLTS state graph with a decrease of 16% in the size of the interface. In the second benchmark, using $PF^4[out]$ we achieve, with an interface that is 55% smaller, the same projected VLTS graph. Moreover, with $TR[out]$, which is 91% smaller than the original interface, we achieve a 75% state reduction of the original VLTS graph.

4.5 Discussion

The results achieved so far indicate that we can use Lts language over approximations in order to produce interfaces which are effective in constraining the generation of Ltss. In the ODP case study, we achieve interface effectiveness which is very close to the one achieved when the most specific environment is used. The two benchmarks which we present here, indicate that there is no one particular language over-approximation technique that performs best in every experiment. This was to be expected however, since what we are doing is effectively applying different heuristics in the generation of interfaces.

5 Conclusion and Future Perspectives

This paper describes how Lts language over-approximations can be used to alleviate the problem of state space explosion in compositional verification. We

have shown how an ODP trader can be generated by semi-composition with a more generic interface with fewer states. Using a reduced interface, we are able to achieve the same number of states for the ODP trader as was achieved with the original (full) interface. As the complexity of the objects composed with the ODP trader increases, the use of LTS reductions which produce language over-approximations makes it possible to generate reasonably sized and effective interfaces.

There are three main directions which, we feel, need to be further explored. The first one it that of coming up with more effective heuristics for LTS reduction. The challenge here is that of being able to design heuristics, which are on the one hand effective in describing the environment as specific as possible and on the other are not computationally intensive to produce. The best solution would be that of generating interfaces on-the-fly. The second direction is that of coming up with alternative ways of combining LTS language over-approximations with refined interfaces. Finally we plan to investigate how the static analysis of the specification of a particular composition of processes can help in the generation of constrains to include as part of the interface.

References

[CK93] Shing-Chi Cheung and Jeff Kramer. Enhancing compositional reachability analysis with context constraints. In *Foundations of Software Engineering*, pages 115–125, 1993.

[GLMS07] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV 2007 (Berlin, Germany)*, volume 4590, pages 158–163, jul 2007.

[GS91] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 186–196, London, UK, 1991. Springer-Verlag.

[GSL96] Susanne Graf, Bernhard Steffen, and Gerald Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[JL97] 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.

[Lan05] Frédéric Lang. Exp.open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM 2005 (Eindhoven, The Netherlands)*, November 2005. Full version available as INRIA Research Report RR-5673.

[Lan06] Frédéric Lang. Refined interfaces for compositional verification. In *International Conference on Formal Techniques for Networked and Distributed Systems FORTE 2006*, 2006.