Interaction Abstraction for Compositional Finite State Systems

Wayne Liu

Department of Electrical and Computer Engineering
University of Waterloo
wbliu@uwaterloo.ca

Abstract. A new algorithm for reducing the state space of compositional finite state systems is introduced. Its goal is similar to compositional minimization algorithms as it tries to preserve only the relevant information for checking properties. It works better than compositional minimization because it reduces components individually and does not need to compose components. Hence it does not suffer from state explosion. Instead, it uses information about interactions with other components, and merges interactions that do not lead to different relevant behaviour. Experiments show that it reduces state spaces dramatically in the cases when only a part of the system's behaviour is of interest

1 Introduction

Model-checking encounters the state-explosion problem. To keep the state space manageable for model-checkers, models of systems should only include features relevant to the property being checked. Holzmann[12] showed it is possible check useful properties using very small models—less than 100 states. Unfortunately, it is often too expensive to manually create a separate model for each property to check. Thus, a single model of a system must be used to verify many different properties of the system.

In these situations, it is desirable to have an algorithm that can abstract away irrelevant features of the model, to create a reduced state space that preserves the property being checked.

A conceptual basis for this kind of abstraction is provided by equivalences of models, such as observation equivalence. Efficient minimization algorithms exist for observation equivalence. Unfortunately, minimization algorithms require the global state space to be generated, which means the reduction requires more effort than checking the model directly.

An approach to alleviate this problem is by compositional minimization, that is composing and minimizing subsets of the system components. Nevertheless, subsystems of components must be composed before they can be reduced, and which may cause state explosion in intermediate state spaces. Another problem is 'spurious behaviour', where a subsystem may exhibit behaviour that is not possible when it is a part of the system as a whole. Interface processes[17,5] have been proposed to reduce spurious behaviour. However, small and effective interface processes are very difficult to find.

Furthermore, removing spurious behaviour still leaves much 'redundant' behaviour in the reduced subsystems. Redundant interactions are interactions with the rest of the system that do not affect the property being checked.

The interaction abstraction algorithm is a new algorithm that has similar goals to compositional minimization. However, it automatically uses information about interactions with the rest of the system to remove redundant information from the model, while keeping information necessary to preserve the property being checked.

2 Abstraction interaction algorithm

2.1 Basic Notation

The algorithm is based on labelled transition systems (LTS)[16].

DEFINITION: An **LTS** is defined as a tuple (Q, A, Δ, q_0) where

- Q is a set of states
- A is a set of labels (or events)
- Δ is a set of transitions, $p_1 \text{—}a\rightarrow q_2$, where $p_1, q_2 \in Q$ and $a \in A$
- $q_0 \in Q$ is the initial state

DEFINITION: The **composition** of two LTSs, $S = (P_1, A_1, \Delta_1, p_{10})$ and $T = (P_2, A_2, \Delta_2, p_{20})$, is the LTS $S \,||\, T$, defined as (Q, A, Δ, q_0) where

- $A = A_1 \cup A_2$,
- $Q = P_1 \times P_2$,
- Δ is the set of transitions of the form $(p_{11}, p_{21}) \text{—}a\rightarrow (p_{12}, p_{22})$ where
  - if $a \notin A_1 \cap A_2$ and $p_{11} \text{—}a\rightarrow p_{12}$, then $(p_{11}, p_{21}) \text{—}a\rightarrow (p_{12}, p_{21}) \in \Delta$
  - if $a \notin A_1 \cap A_2$ and $p_{21} \text{—}a\rightarrow p_{22}$, then $(p_{11}, p_{21}) \text{—}a\rightarrow (p_{11}, p_{22}) \in \Delta$
  - and if $a \in A_1 \cap A_2$ and $p_{11} \text{—}a\rightarrow p_{12}$ and $p_{21} \text{—}a\rightarrow p_{22}$, then $(p_{11}, p_{21}) \text{—}a\rightarrow (p_{12}, p_{22}) \in \Delta$
- the initial state is $q_0 = (p_{10}, p_{20})$

The LTS formalism models abstraction through hiding of labels. Labels that are not of interest in the system (e.g. not mentioned in property to be checked, and not used to interact with other components) can be hidden by renaming them to the spe-

cial label, t. A sequence of 0 or more transitions with hidden labels is written as $p_1 \Longrightarrow p_2$ if exist $p_1 \rightarrow t \rightarrow \ \ldots \ \rightarrow t \rightarrow p_i$, $i \geq 0$. The weak transition relation is defined as $p_1 \Longrightarrow a \Longrightarrow p_2$, if exists $p_1 \Longrightarrow t \Longrightarrow p_3 \rightarrow a \rightarrow p_4 \Longrightarrow t \Longrightarrow p_2$. The weak transition relation enables any number of hidden actions to take place without affecting the observable properties of the system. A system S in which only labels in a set L are visible, and the rest are hidden, is denoted $S\langle L \rangle$.

Various notions of equivalence exist for LTS, including the well-known observation equivalence[16]. Observation equivalence is defined using a family of bisimulation relations:

- $R_0 \equiv Q \times Q$,
- $R_{k+1} \equiv \{(p_1,p_2) \mid \forall a \in A \ \forall p_1'(p_1 \Longrightarrow a \Longrightarrow p_1' \Rightarrow \exists p_2'(p_2 \Longrightarrow a \Longrightarrow p_2' \wedge (p_1',p_2') \in R_k) \wedge$
  $\forall p_2'(p_2 \Longrightarrow a \Longrightarrow p_2' \Rightarrow \exists p_1'(p_1 \Longrightarrow a \Longrightarrow p_1' \wedge (p_1',p_2') \in R_k)\}$

The (observation) equivalence relation is defined as $\sim \equiv \bigcup_0^\infty R_k$. Thus, $p_1 \sim p_2$ if $(p_1,p_2) \in R_k$ for all k. Two systems are observation equivalent if their initial states are equivalent.

Given a classification of the states of S, where $[p]$ denotes the class of p, the **quotient** of S, is the LTS $[S] = ([Q], A, [\Delta], [q_0])$ where

- $[Q] = \{[q]\}$ for all $q \in Q$
- $[\Delta] = \{[p_1] \rightarrow a \rightarrow [q_2]\}$ for all $p_1 \rightarrow a \rightarrow q_2 \in \Delta$

2.2 Effect of interactions

To get an intuition for how the algorithm works, consider the composition of two systems, $S \,||\, T$, in Figure 1. The goal is to find reduced versions, $[S]$ and $[T]$, so that $(S \,||\, T)\langle a_1, a_2, a_3 \rangle \sim ([S] \,||\, [T])\langle a_1, a_2, a_3 \rangle$.

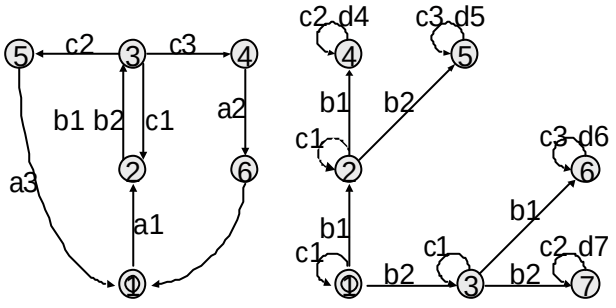

Fig. 1. Example system with two components.

The component on the left can be thought of as a simple model of a telephone, and the component on the right can be thought of as a simple model of a phone directory.

Thus, the telephone can go offhook (a1), then a '1' is dialed (b1), then the number is found to be incomplete (c1), in which case, a '2' is dialed (b2), and it is found to be a complete, and valid number (c3). The telephone then connects to the other phone (a2), and so on. On the other hand, dialing a second '1' would result in an invalid number (c2), and the telephone would give a busy tone (a3).

Suppose we are interested only in the actions offhook, connect, or busy tone (a1, a2, a3). In particular, we are not interested in which numbers are dialed (b1, b2), nor the internal interactions between the components (c1, c2). Intuitively, the directory model can be reduced to just three states: from the initial state, it can move to a state with a valid phone number, or an invalid one. The telephone model can be reduced to move from state 2 directly to 5 or 4.

The idea is to achieve the reduction is to record the effect of the interactions of the components, rather than the actual labels. As a first attempt, we can use this idea directly, and relabel the directory model as in Figure 2. For example, the transition $1 \text{—} b1 \rightarrow 2$ is relabelled by the effect of the interaction on the phone model. The phone model makes the transition $2 \text{—} b1 \rightarrow 3$, so the directory model gets the transition $1 \text{—} 23 \rightarrow 2$.

This relabelling allows the merging of states {4,7}, {2,3} and {5,6}. However, the reduce graph is still unsatisfactory in that the merged state {2,3} is distinguished from state {1}. That means the model tracks how many numbers must be dialed to get a complete number. However, from the point of view of observational equivalence, it does not matter how many internal steps occur between externally visible steps.

The approach to obtain full reduction is to label the model with the transitive closure of the effects of individual interactions. Part of the model labelled with the transitive closure is shown in the left part of Figure 3. The labelling shows the source and destination of the other component, after a sequence of internal interactions. An extra transition between states 1 and 4 has been added.

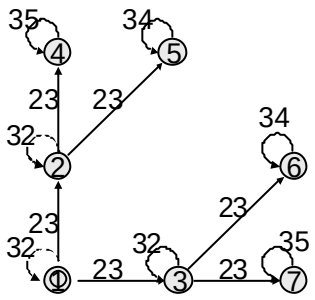

Fig. 2. Component relabelled with effect of interactions.

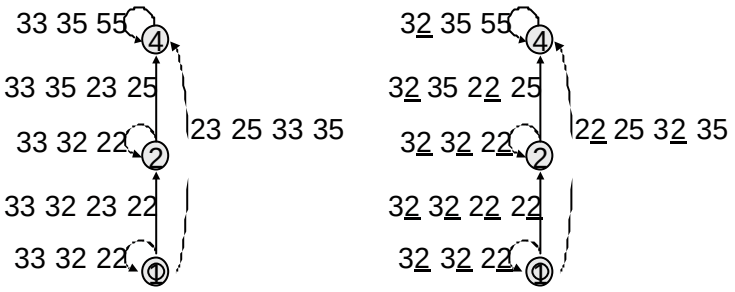

Fig. 3. Part of component relabelled with transitive effect of interactions.

Unfortunately, the model still cannot be further reduced, as state 1 has a transition —23→ that leads to state 2, while state 2 does not have a transition —23→ that leads to an equivalent state. The problem is the phone model's states 2 and 3 are distinct. But they do not need to be, as they do not result in different external behaviour (as internal interactions are not observable). If states 2 and 3 of the phone model can be merged (labelled as '2'), then the portion of the directory model becomes the right side of Figure 3, where it can be seen that states 2 and 1 can be merged.

The idea is to track whether interactions cause the other model to move to equivalent states, rather than just the same states. Unfortunately, which states can be considered equivalent in the other model also depends on which states can be considered equivalent in this model, and vice versa. Thus, the equivalence reduction needs to be computed iteratively.

At the end, the interaction labels in the two reduced models must be matched in order to allow the models to compose.

2.3 Algorithm

The steps of the interaction abstraction algorithm for one component are as follows:

1. Calculate transitive effect of interactions:
   - Store tuple (p, q, p', q') iff whenever the state (p,q) is reachable, there is a transition $(p,q) \Longrightarrow t \Rightarrow (p',q')$ in $(S \mid\mid T)\langle L\rangle$
2. For a given classification $[T]_0$ of T, relabel S with assumed equivalent effects, to obtain S_1:
   - Remove all transitions with labels not in $L \cup \{t\}$
   - Add a transition $p \text{—} q[q']_0 \rightarrow p'$ for each tuple (p, q, p', q')
3. Classify S_1 to obtain $[S]_1$:
   - Set $[p_1]_1=[p_2]_1$ iff $p_1 \sim p_2$ in S_1

The iteration for two components is as follows:

4. Repeat Steps 2 and 3 until $[S]_k$ and $[T]_k$ are the same as $[S]_{k-1}$ and $[T]_{k-1}$:
   - Update $[T]_k$ using $[S]_{k-1}$ if changed, and vice versa
5. Finally, label interactions in $[S]=[S]_k$ and $[T]=[T]_k$:
   - Remove transitions with labels not in L
   - For each tuple (p, q, p', q') add transitions $[p]\text{—}[p][q][p'][q']\rightarrow[p']$ in $[S]$, and $[q]\text{—}[p][q][p'][q']\rightarrow[q']$ in $[T]$

After Step 5, $[S]$ and $[T]$ can be composed using the "interaction labels".

2.4 Proof of correctness

We want to prove that, at the end of the algorithm, $(S \,||\, T)\langle L\rangle \sim ([S] \,||\, [T])\langle L\rangle$ by proving that for all (p,q) reachable, $(p,q) \sim ([p],[q])$. And since the initial state (p_0,q_0) is reachable, then the two compositions are equivalent. First, we prove some properties of the algorithm.

LEMMA 1:

If $[p_1]_k=[p_2]_k$ then all of the following are true:

a) for all transitions $p_1 \Longrightarrow t \Rightarrow p_1'$ there exists a transition such that $p_2 \Longrightarrow t \Rightarrow p_2'$ where $[p_2']_k=[p_1']_k$
b) for all transitions $p_2 \Longrightarrow t \Rightarrow p_2'$ there exists a transition such that $p_1 \Longrightarrow t \Rightarrow p_1'$ where $[p_2']_k=[p_1']_k$
c) for all tuples (p_1, q, p_1', q_1'), there exists a tuple (p_2, q, p_2', q_2') where $[p_2']_k=[p_1']_k$ and $[q_2']_k=[q_1']_k$
d) for all tuples (p_2, q, p_2', q_2'), there exists a tuple (p_1, q, p_1', q_1') where $[p_1']_k=[p_2']_k$ and $[q_1']_k=[q_2']_k$

PROOF:

$[p_1]_k=[p_2]_k$
$\Rightarrow p_1 \sim p_2$ in S_k, by definition of $[S]_k$
$\Rightarrow$ for all transitions $p_1 \Longrightarrow a \Rightarrow p_1'$ there exists a transition $p_2 \Longrightarrow a \Rightarrow p_2'$ where $p_2' \sim p_1'$ in S_k, by corollary to the definition of $\sim$
$\Rightarrow$ for all transitions $p_1 \Longrightarrow t \Rightarrow p_1'$ there exists a transition $p_2 \Longrightarrow t \Rightarrow p_2'$ where $[q_2']_{k-1}=[q_1']_{k-1}$, by definition of $[T]_{k-1}$, by which follows cases (a) and (b).

For cases (c) and (d), if there exists a tuple (p_1, q, p_1', q_1')
$\Rightarrow$ there exists a transition $p_1 \text{—} q[q_1']_{k-1} \rightarrow p_1'$
$\Rightarrow$ there exists a transition $p_2 \Longrightarrow q[q_1']_{k-1} \Rightarrow p_2'$ where $[p_2']_{k-1}=[p_1']_{k-1}$ since $p_1 \sim p_2$
And since tuples contain transitive effects (Step 1), there exists a tuple (p_2, q, p_2', q_2') where $[q_2']_{k-1}=[q_1']_{k-1}$.

■

LEMMA 2:

a) If there exists a transition $(p,q) \Longrightarrow t \Rightarrow (p',q')$ in $(S \,||\, T)\langle L\rangle$ then there exists a transition $([p],[q]) \Longrightarrow t \Rightarrow ([p'],[q'])$ in $([S] \,||\, [T])\langle L\rangle$

b) If there exists a transition $p \Longrightarrow a \Rightarrow p'$ in S, where $a \notin L$, then there exists a transition $[p] \Longrightarrow a \Rightarrow [p']$ in $[S]$

PROOF:

Proof of (a):

$(p,q) \Longrightarrow t \Rightarrow (p',q')$ in $(S \,||\, T)\langle L\rangle$

$\Rightarrow$ exist tuples (p, q, p', q') by Step 1

$\Rightarrow$ exist transitions $[p] \text{—} [p][q][p'][q'] \rightarrow [p']$ in $[S]$, and $[q] \text{—} [p][q][p'][q'] \rightarrow [q']$ in $[T]$ by Step 5

$\Rightarrow ([p],[q]) \Longrightarrow t \Rightarrow ([p'],[q'])$ in $([S] \,||\, [T])\langle L\rangle$ by composition.

Proof of (b): True since the quotient keeps transitions of elements of a class, and the algorithm does not relabel transitions with label a in L.

■

LEMMA 3 (inverse of Lemma 2):

a) If there exists a transition $([p],[q]) \Longrightarrow t \Rightarrow ([p'],[q'])$ in $([S] \,||\, [T])\langle L\rangle$ and (p,q) is reachable, then there exists a transition $(p,q) \Longrightarrow t \Rightarrow (p_1',q_1')$ in $(S \,||\, T)\langle L\rangle$ where $[p_1']=[p']$, $[q_1']=[q']$.

b) If there exists a transition $[p] \Longrightarrow a \Rightarrow [p']$ in $[S]$, where $a \notin L$, then there exists a transition $p \Longrightarrow a \Rightarrow p_1'$ in S where $[p_1']=[p']$.

PROOF:

Proof of (a):

$([p],[q]) \Longrightarrow t \Rightarrow ([p'],[q'])$

Since the only interactions are through the interaction labels, then by composition and hiding,

$\Rightarrow$ exist transitions $[p] \Longrightarrow [p][q][p'][q'] \Rightarrow [p']$ in $[S]$ and $[q] \Longrightarrow [p][q][p'][q'] \Rightarrow [q']$ in $[T]$

$\Rightarrow$ exist transitions $[p] \Longrightarrow t \Rightarrow [p_2] \text{—} [p][q][p'][q'] \rightarrow [p_2'] \Longrightarrow t \Rightarrow [p']$ in $[S]$, and $[p_2]=[p]$, $[p_2']=[p']$ (by Step 5)

Since $[S]_k=[S]_{k-1}$ and $[T]_k=[T]_{k-1}$,

$\Rightarrow$ exist transitions $p_1 \text{—} q_1[q']_k \rightarrow p_1'$ in S_k where $[p_2]_k=[p_1]_k=[p]_k$, $[p_2']_k=[p']_k$

$\Rightarrow$ exist tuples (p_1, q_1, p_1', q_1'), where $[q_1]_k=[q]_k$, $[p_1']_k=[p']_k$

$\Rightarrow$ exist tuples (p, q_1, p_1', q_1'), where $[q_1]_k=[q]_k$ (By Lemma 1(c) and (d), since $[p_1]_k=[p]_k$)

$\Rightarrow$ exist tuples (p, q, p_1', q_1') (By Lemma 1 applied to $[T]_k$)

$\Rightarrow$ if (p,q) is reachable, then there exists a transition $(p,q) \Longrightarrow t \Rightarrow (p_1',q_1')$ in $(S \,||\, T)\langle L\rangle$ where $[q_1']=[q']$ and $[p_1']=[p']$

Proof of (b): True by definition of quotient and by Lemma 1(a) and (b).

■

THEOREM: If (p,q) reachable in $(S \,||\, T)<L>$, then $(p,q) \sim ([p],[q])$ in $([S] \,||\, [T])<L>$.
PROOF:

The proof is a simple application of the definitions. It only uses the properties of the algorithm given in Lemmas 2 and 3.

Obviously for all (p,q) reachable, $((p,q), ([p],[q]))$ in R_0.

Assume for all (p,q) reachable, $((p,q), ([p],[q]))$ in R_{k-1}.

$(p,q) \Longrightarrow a \Rightarrow (p',q')$ where a in L
$\Rightarrow (p,q) \Longrightarrow t \Rightarrow (p_1',p_1') \rightarrow a \rightarrow (p_2',q_2') \Longrightarrow t \Rightarrow (p',q')$ by definition of $\Longrightarrow a \Rightarrow$
$\Rightarrow ([p],[q]) \Longrightarrow t \Rightarrow ([p_1'],[p_1']) \Longrightarrow a \Rightarrow ([p_2'],[q_2']) \Longrightarrow t \Rightarrow ([p'],[q'])$ by lemma 1 (a) and (b)
$\Rightarrow ([p],[q]) \Longrightarrow a \Rightarrow ([p'],[q'])$ by definition of $\Longrightarrow a \Rightarrow$
And by induction hypothesis, $((p',q'), ([p'],[q']))$ in R_{k-1}

For the other direction:

$([p],[q]) \Longrightarrow a \Rightarrow ([p'],[q'])$
$\Rightarrow ([p],[q]) \Longrightarrow t \Rightarrow ([p_1'],[p_1']) \rightarrow a \rightarrow ([p_2'],[q_2']) \Longrightarrow t \Rightarrow ([p'],[q'])$ by definition of $\Longrightarrow a \Rightarrow$
$\Rightarrow (p,q) \Longrightarrow t \Rightarrow (p_3',q_3') \Longrightarrow a \Rightarrow (p_4',q_4') \Longrightarrow t \Rightarrow (p_5',q_5')$ where $[p_3']=[p_1']$, $[q_3']=[p_1']$, $[p_4']=[p_2']$, $[q_4']=[q_2']$, $[p_5']=[p']$, and $[q_5']=[q']$ by lemma 2 (a) and (b)
$\Rightarrow (p,q) \Longrightarrow a \Rightarrow (p_5',q_5')$ where $[p_5']=[p']$, and $[q_5']=[q']$
And by induction hypothesis, $((p_5',q_5'), ([p'],[q']))$ in R_{k-1}.
Thus, $((p,q), ([p],[q]))$ in R_k for all k

■

Next, it is necessary to show the algorithm always terminates, which can be done using the following lemma.

LEMMA: Let the classification $[T]_k$ be a refinement of the classification $[T]_{k-1}$. Then $[S]_{k+1}$ computed using $[T]_k$ is a refinement of $[S]_k$ computed using the labelling $[T]_{k-1}$.
PROOF:

Suppose $[p_1]_k=[p_2]_k$. We want to show that $[p_1]_{k+1}=[p_2]_{k+1}$, that is, if $p_1 \sim p_2$ in S_k, then $p_1 \sim p_2$ in S_{k+1}.

Since $[T]_k$ is a refinement of $[T]_{k-1}$, if labels $q_1[q_1']_{k-1} = q_1[q_2']_{k-1}$, then labels $q_1[q_1']_k = q_1[q_2']_k$. Thus, if two transitions in S_k have the same labels using the labelling $\Longrightarrow q_1[q_1']_{k-1} \Rightarrow$, the transitions in S_{k+1} will still have the same labels using the labelling $\Longrightarrow q_1[q_2]_k \Rightarrow$.

Supose $p_1 \sim p_2$ in S_k
$\Leftrightarrow$ for any transition $p_1 \Longrightarrow a_1 \Rightarrow p_1'$, there is a corresponding transition $p_2 \Longrightarrow a_1 \Rightarrow p_2'$, and $p_1' \sim p_2'$ in S_k
Since the labels a_1 of the two transitions $p_1 \Longrightarrow a_1 \Rightarrow p_1'$ and $p_2 \Longrightarrow a_1 \Rightarrow p_2'$ are guaranteed to be the same in S_{k+1} as in S_k,
$\Rightarrow$ for any transition $p_1 \Longrightarrow a_1 \Rightarrow p_1'$, there is a corresponding transition $p_2 \Longrightarrow a_1 \Rightarrow p_2'$, and $p_1' \sim p_2'$ in S_{k+1}
$\Leftrightarrow p_1' \sim p_2'$ in S_{k+1}

■

The lemma shows that the algorithm is monotonic, that is, each iteration computes a refinement of the classification of the previous iteration. Since the number of refinements is finite, the number of iterations is finite and the algorithm must terminate.

2.5 Multiple components, and multi-way interactions

It has been shown how to compute the reduction for two components. For multiple components with 2-way interactions, the interactions between each pair of components are collected and labelled separately.

For Step 1, we can simply store $S_iS_j{:}(p_i, p_j, p_i', p_j')$ if there is a transition $(p_i, p_j) \Longrightarrow{t}\Rightarrow (p_i', p_j')$ in $(S_i \,||\, S_j)\langle L\rangle$. This satisfies the condition that $S_iS_j{:}(p_i, p_j, p_i', p_j')$ is stored iff whenever the state $(\ldots p_i,\ldots, p_j,\ldots)$ is reachable, there is a transition $(\ldots p_i,\ldots, p_j,\ldots) \Longrightarrow{t}\Rightarrow (\ldots p_i',\ldots, p_j',\ldots)$ in $(S_1 \,||\, \ldots \,||\, S_n)\langle L\rangle$ (only the i and j components change). For Step 2, the labels are added as $p_i \text{—} p_j[p_j']_0 \rightarrow p_i$ for all tuples $S_iS_j{:}(p_i, p_j, p_i', p_j')$. For the iteration Step 4, update all $[S_i]_k$ if any $[S_j]_{k-1}$ with which it interacts has changed. For Step 5, for all tuples $S_iS_j{:}(p_i, p_j, p_i', p_j')$, add transitions $[p_i] \text{—} S_iS_j{:}[p_i][p_j][p_i'][p_j'] \rightarrow [p_i']$ in $[S_i]$ and $[p_i] \text{—} S_iS_j{:}[p_i][p_j][p_i'][p_j'] \rightarrow [p_i']$ in $[S_j]$.

Lemma 1 is changed to

a) for all tuples $(p_{i1}, p_j, p_{i1}', p_{j1}')$, there exists a tuple $(p_{i2}, p_j, p_{i2}', p_{j2}')$ where $[p_{i2}']_k = [p_{i1}']_k$ and $[p_{j2}']_k = [p_{j1}']_k$

b) for all tuples $(p_{i2}, p_j, p_{i2}', p_{j2}')$, there exists a tuple $(p_{i1}, p_j, p_{i1}', p_{j1}')$ where $[p_{i1}']_k = [p_{i2}']_k$ and $[p_{j1}']_k = [p_{j2}']_k$

Lemma 2 is changed to

a) If there exists a transition $(\ldots p_i,\ldots, p_j,\ldots) \Longrightarrow{t}\Rightarrow (\ldots p_i',\ldots, p_j',\ldots)$ in $(S_1 \,||\, \ldots \,||\, S_n)\langle L\rangle$ then there exists a transition $(\ldots[p_i],\ldots, [p_j],\ldots) \Longrightarrow{t}\Rightarrow (\ldots[p_i'],\ldots, [p_j'],\ldots)$ in $([S_1] \,||\, \ldots \,||\, [S_n])\langle L\rangle$

b) If there exists a transition $p_i \Longrightarrow{a}\Rightarrow p_i'$ in S_i, where $a \notin L$, then there exists a transition $[p_i] \Longrightarrow{a}\Rightarrow [p_i']$ in $[S_i]$

Similarly, change Lemma 3 and all the proofs. That is, p, q is replaced with $\ldots p_i,\ldots, p_j,\ldots$ and $S \,||\, T$ is replaced with $S_1 \,||\, \ldots \,||\, S_n$. The proof of the theorem is changed so that a t-transition $(p_1,\ldots, p_n) \Longrightarrow{t}\Rightarrow (p_1',\ldots, p_n')$ in $(S_1 \,||\, \ldots \,||\, S_n)\langle L\rangle$ must be broken down into a sequence of constituent t-transitions with pair-wise interactions, such as $(\ldots p_i,\ldots, p_j,\ldots) \Longrightarrow{t}\Rightarrow (\ldots p_i',\ldots, p_j',\ldots)$. Then, the lemmas are applied to each constituent t-transition.

For multi-way interactions, interactions for each subset of interacting components is collected and labelled separately. For example, for a 3-way interaction, the stored vectors are $S_1S_2S_3(p_1, p_2, p_3, p_1', p_2', p_3')$ if there is a transition $(p_1, p_2, p_3) \Longrightarrow{t}\Rightarrow (p_1', p_2', p_3')$ in $(S_1 \,||\, S_2 \,||\, S_3 \,||\, \ldots \,||\, S_n)\langle L\rangle$. The conditions, the other steps, and the proof proceed similarly.

2.6 Algorithm complexity

A bound for the algorithm complexity can be obtained by adding up the cost of basic operations.

Since interactions with each pair of components are collected separately, the stored interactions (for 2-way interactions) is $S_iS_j(p_i, p_j, p_i', p_j')$. Thus, the maximum number of interactions of S_i is at worst nm^4 for a system of n components, all with m states. For multi-way interactions, the number of interactions is at worst nm^{2k} if there are at most k-way interactions.

Minimization of each component by observational equivalence can be performed in $O(ne)$ time, where e is the number of transitions of the relabelled components. The number of transitions is the number of interactions plus the number of visible transitions. At worst, this is lm^2+nm^4, where l is the number of externally visible labels. Assuming l is unrelated to m and n, then the number of transitions is $O(nm^4)$. (Typically, l should be small.) Thus, the minimization has complexity $O(n^2m^4)$. (For k-way interactions, the minimization has complexity $O(n^2m^{2k})$).

During one iteration, at most n minimizations is required. The number of iterations is at most the sum of the states of the components, nm, since the size of one reduced component must increase or else the algorithm terminates. Thus, the number of minimizations is at most $O(n^2m)$, and the overall complexity is $O(n^4m^5)$. (For k-way interactions, complexity is $O(n^4m^{2k+1})$).

2.7 Algorithm notes

Collecting interactions with each pair of components separately means reductions can be computed without composing the rest of the system. However, the disadvantage is that interactions with different components are being distinguished from each other, thus lessening the amount of reduction possible. For example, interactions of S_i with two different components would still be distinguished even if the interactions do not change the states of either of the other components. In particular, a system with all labels hidden would not reduce to a set of 1-state abstractions!

An optimization can be made in the number of edges added to components. In many cases, the same edges in S (labelled $p_1[p_2]$) may cause many different edges in T (labelled $q_1[q_2]$). But, for the purpose of reducing the components, many of the labels $q_1[q_2]$ are redundant. Two labels $q_1[q_2]$ are redundant if they always appear together in all transitions labelled $p_1[p_2]$, since they can never be used to distinguish any states in S.

Formally, it is safe to merge the labels a and b if, whenever there is a transition labelled with $p_1 \Longrightarrow p_2$, then there is also a transition labelled with $p_1 \Longrightarrow p_2$, and vice versa. Experiments show this optimization greatly reduces the number of transitions, and significantly speeds up the minimization of components.

2.8 Scalability of algorithm

In theory, the abstraction algorithm avoids state explosion by avoiding composition of components and abstracting each component individually. Nevertheless, the theoretical complexity of $O(n^4m^5)$ looks quite daunting for practical use. However, the actual situation is much better in experiments.

The major factor in the cost is the set of interactions between two components. This set has a theoretical size complexity of $O(m^4)$. This level of complexity can occur in practice with components that are basically data structures. It is important to realize, however, that this complexity simply results from interactions between two components. Any type of model-checking that takes into account interactions between components must face at least this level of complexity.

Let us assume that individual components are small, so that the tools built are able to handle compositions of two components in a reasonable (i.e. constant) time. This assumption eliminates the powers of m. Also at the most four iterations of reductions were required for each component in experiments. That is much better than the worst case of $O(n^2)$ minimizations of each component. Further, assume that components only interact with a limited (i.e. bounded) number of other components. This eliminates one power of n.

Thus, the actual observed complexity under these assumptions is proportional to $O(n)$ and the effort depends linearly on the number of components.

3 Implementation and results

The algorithm has been implemented in a prototype tool as part of the Component Interaction Testing project. One of the goals of the project is to generate test cases from formal design models of software. The other parts of the project include

- the ObjectState formal object-oriented modelling language with features similar to UML for Real-Time[15]
- formal interaction coverage criteria for generating test requirements from models of component (Event-flow[14])
- tools to translate design models and test requirements to the LTS formalism (or Promela[10]), and generate test cases, exploiting the abstraction algorithm.

The tool uses the Caesar-Aldebaran Development Package (CADP)[6]. The CADP toolbox provides facilities to generate LTS files, compute compositions and minimizations. It does not have facilities to compute hook compositions.

The performance of the algorithm was tested using a model of a private branch exchange (PBX) software. The design model is 1000 lines of ObjectState code, while the implementation of the PBX is 16 000 lines of C code.

The following model-checking tools and algorithms were compared:

Component subset	PROD	Alde-baran	SPIN	Exhibi-tor	Increment test gen	Interact abstract
REQ +CH	Memory out	Time out	554	790	365	4113
REQ +CH +DB	-	-	Not found	7644	534	6158
REQ +CH +DB +CM	-	-	-	Memory out	677	6311
REQ +2×CH +DB +CM	-	-	-	-	Time out	8436
REQ +3×CH +DB +CM	-	-	-	-	-	10431
REQ +2×CH +DB +CM +LS	-	-	-	-	-	Memory out

Table 1 Times (seconds) for generating paths by each tool

- SPIN[10], using depth-first, partial-order reduction, supertrace
- Exhibitor, part of CADP, using simple on-the-fly breadth-first search
- Araprod[19] using on-the-fly breadth-first search with partial-order reduction
- Aldebaran, part of CADP, using minimal model generation[3], BDDs
- new analyzer using incremental test generation (observational minimization)
- new analyzer with incremental test generation, and interaction abstraction.

The tools were chosen because they implement advanced and successful model-checking algorithms. All the tools are freely obtainable for research purposes.

3.1 Test results

Beginning with only one component and the test requirement, the tools were given more components to analyze until they exceeded available memory or failed to give an answer in a reasonable amount of time (i.e. 24 hours).

The test results are shown in Table 1. The components of the PBX model as listed as CH (call handler), DB (database), CM (call manager), LS (line scan), and REQ (the test requirement).

The interaction abstraction algorithm allows much larger models to be analyzed than possible with the other tools. Even for the simplest case of finding a path for a single component, the Aldebaran and Araprod tools failed. SPIN failed for the interaction of two components. The simple breadth-first search in Exhibitor performed better than the more complex algorithms, but it eventually failed to compute a path for three components. With incremental test generation (observational minimization), an

additional component can be analyzed. Adding interaction abstraction, five components can be analyzed. However, it also fails when the sixth component was included.

The reason the interaction abstraction failed was the large number of interactions with the sixth component (LS). Recall that there are worst $O(m^4)$ interactions, and this seems to occur in this case. This number of interactions overwhelms the available memory.

The sizes of the models and the impressive reductions achieved by interaction abstraction are shown in Table 2. Each table shows a subset of components that was analyzed. Note that the number of states of a component can be different in different subsets because the component may have to interact with different numbers of other components, and hence require more states. Also the test generation procedure is incremental, and makes several passes. Thus, the actual number of states for each pass is usually much smaller than the maximum shown.

Note that the implementation currently has an error in it that causes it to over-

	REQ	CH1
RAW	7	85956
MIN	5	12790
ABS	5	315

	REQ	DB	CH1
RAW	7	291	85956
MIN	5	4	12390
ABS	5	2	383

	REQ	DB	CM	CH1
RAW	7	291	146	85956
MIN	5	4	125	12790
ABS	5	2	37	392

	REQ	DB	CM	CH1	CH2
RAW	7	579	390	85956	85956
MIN	5	7	223	12790	12790
ABS	5	2	18	63	156

	REQ	DB	CM	CH1	CH2	CH3
RAW	7	579	390	85956	85956	85956
MIN	5	7	223	12790	12790	12790
ABS	5	2	172	102	174	81

	REQ	DB	CM	LS	CH1	CH2
RAW	7	579	390	41371	85956	85956
MIN	5	7	223	1665	12790	12790
ABS	5	2	166	861	326	408

Table 2: number of states of components, after observation minimization. after interaction abstraction (maximum over all test generation passes)

reduce components in some cases, and hence generate incorrect test cases. However, all the test cases generated in the examp les shown are correct. Thus, it is likely that the reductions are correct for these examples.

4 Related work

There are many state space reduction algorithms, and they target different kinds of redundancy in the state space representation. Many redundancies exist as some type of shared state space structure, such as symmetry[13], partial-order equivalence[9], hierarchical state machines[1], shared state representation (using BDDs[4], state compression[8]), and so on. Interaction abstraction, on the other hand, exploits redundancy in model interactions that do not lead to different relevant behaviour. For example, models of systems may deal with many aspects, but only one aspect is of interest at a time.

Interaction abstraction can be used as a preprocessing step for other reduction techniques. After abstraction, other techniques can be applied, including compositional minimization, on-the-fly search, or partial-order reduction. This approach exploits the greatest amount of redundancy in models.

The most closely related algorithms are compositional minimization algorithms. Interaction abstraction differs compositional minimization in that it does not need to compose components of a subsystem, but reduces each component by itself using information about interactions with other components. Thus it avoids the state-explosion problem. Also, it takes into account context of components (its interactions with the rest of the system). Unlike methods using interface processes[5][17], it is completely automatic, and multiple contexts are taken into account without needing to compose the contexts. In addition, interaction abstraction merges redundant interactions, and only preserves behaviour that is relevant to the property being checked, allowing for greater reduction.

While compositional minimization typically tries to construct a single minimal global model on which many different properties can be evaluated, interaction abstraction is much more effective when few behaviours are observable. Therefore, it is more effective to create a specific abstraction for each property to be checked.

5 Conclusions

A new algorithm has been presented to reduce the state space of a model for model checking by abstracting component interactions that are not relevant to the property being checked. It is proved that the algorithm preserves behaviour of interest. Ab-

straction is performed without composing components, thus avoiding the state space explosion problem.

The algorithm is most useful when models of systems may deal with many aspects, but only one aspect is of interest at a time. It is not useful for properties that depend on all behaviours of a system, such as absence of deadlock.

The complexity of the algorithm is $O(n^4m^5)$ for a system with n components that have m states maximum. (For systems where k components interact simultaenously, complexity is $O(n^4m^{2k+1})$). However, for systems that are made up of many small, loosely-coupled components, and each component communicates with a limited number of other components, the algorithm performs well.

Interaction abstraction can be used in conjunction with other reduction techniques, such as compositional minimization, on-the-fly search, or partial-order reduction. By combining different methods, the greatest amount of redundancy in models is exploited.

Experiments show the algorithm is very effective in reducing state spaces, and allowing much larger models to be analyzed than previously possible.

Research is needed to achieve greater reduction. Especially, the algorithm should handle multiple components better. Rather than distinguishing interactions with different components, it is desirable to only consider the final result of interactions with all other components. The problem is how to achieve this without an explosion in the number of interactions that must be stored. In addition, the algorithm should be extended to preserve coarser equivalence relations, such as safety equivalence[2] or trace equivalence. Finally, the algorithm should be incorporated into industrial-strength tools in order to be used for practical applications.

References

1. R. Alur and M. Yannakakis. “Model checking of hierarchical state machines”. Sixth ACM Symposium on the Foundations of Software Engineering, pp. 175-188, 1998
2. A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez and J. Sifakis. Safety for Branching Time Semantics. 18th ICALP, Springer-Verlag, July 1991.
3 A. Bouajjani, J.C. Fernandez, N. Halbwachs, C. Ratel, and P. Raymond. “Minimal state graph generation”. Science of Computer Programming, 18(3), June 1992.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. “Symbolic Model Checking: 10^{20} states and beyond”. Technical Report, Carnegie Mellon University, 1989.
5. S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability Analysis. ACM Transactions on Software Engineering and Methodology. October 1996.
6. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP: A Protocol Validation and Verification Toolbox. Proceed-

ings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA), pages 437-440, August 1996

7. Jean-Claude Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. Science of Computer Programming, volume 13, number 2-3, pages 219-236, May 1990.
8. J-Ch. Grégoire, "State space compression in SPIN with GETSs", Second SPIN Workshop, August 1996.
9. P. Godefroid and P. Wolper. "A partial approach to model checking". In Proc. 6th Annual Symposium on Logic in Computer Science, pages 406-415, July 1991.
10. G. J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall 1991.
11. G.J. Holzmann. The engineering of a model checker: the Gnu i-protocol case study revisited. Proc. of the 6th Spin Workshop LNCS, Vol. LNCS 1680, Springer Verlag, Toulouse France, Sept. 1999.
12. G.J. Holzmann. Designing executable abstractions. Proc. Formal Methods in Software Practice, ACM Press, Clearwater Beach Florida USA, March 1998.
13. K. Jensen. Coloured Petri Nets. Volume 2, Analysis Methods. Monographs in Theoretical Computer Science 575, Springer-Verlag 1992, pp.192-202.
14. Wayne Liu and Paul Dasiewicz. Selecting System Test Cases for Object-oriented Programs Using Event-Flow. Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '97)
15. Andrew Lyons. UML for Real-Time Overview. RATIONAL Software Corporation Whitepaper, April 1998, Available at http://www.rational.com/.
16. R.Milner (1980), A calculus of communication systems, LNCS 92, Springer-Verlag
17. B. Steffen, S. Graf, G. Lüttgen "Compositional Minimization of Finite State Systems". International Journal on Formal Aspects of Computing, Vol. 8, pp. 607-616, 1996.
18. A. Valmari. The State Explosion Problem. Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, pp. 429-528
19. Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995.