Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Learning Minimal Requirements for Compositional Verification

submitted by

Arnaud Fietzke

on
November 30, 2006

Advisor

Sven Schewe

Reactive Systems Group
Department of Computer Science
Saarland University

Reviewers

Prof. Bernd Finkbeiner, PhD
Prof. Dr. Holger Hermanns

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, November 30, 2006

Arnaud Fietzke

Contents

Abstract

Compositional verification is a technique aimed at addressing the state explosion problem associated with model checking. One approach to compositional verification is assume-guarantee reasoning, in which the verification of components of a system allows properties of the whole system to be checked by using assumptions derived from one component in the verification of a second component. Once such intermediate assumptions have been found, they can be used to re-run the verification of the whole system at much lower computational expense. In this context, the size of the intermediate assumptions is of primary importance.

In this thesis we discuss a method for computing minimal intermediate assumptions. The method is based on a recent approach to automatic derivation of intermediate assumptions using the L* algorithm for active learning of regular languages.

1 Introduction

Model checking is a method to verify that the formal description of a system (the model) satisfies a formal specification [4]. The model is usually represented as a state transition system and the model checking process decides whether the model satisfies the required properties by exploring successive states of the system. Especially in the case of systems consisting of several interacting components, the combinatorial blow-up of the state space (the state explosion problem) prevents naive model checking from succeeding in most real-world applications. One focus of current research is thus on reducing the state space that needs to be explored before the verification process can return conclusive results.

An interesting way of addressing the state explosion problem in modular systems is based on the "divide an conquer" paradigm, that is, one decomposes the system into components and applies model checking locally to those components. If the results of this local verification can be efficiently combined to prove the initial system correct, then in principle one avoids any combinatorial explosion since no combinations of states need to be considered.

One such "divide and conquer" approach is known as assume-guarantee reasoning. The key to assume-guarantee reasoning is to consider each component not in isolation, but in conjunction with assumptions about the context of the component [8, 10]. That is, in a system consisting of two components M_1 and M_2, component M_1 is verified under the assumption that context M_2 behaves correctly and symmetrically, M_2 is verified assuming that context M_1 behaves correctly. The challenge of this approach lies in the circularity of the reasoning which must be handled carefully if termination is to be maintained. It is easy to see that this circularity could be broken if a special property A was known, such that component M_1 satisfies the overall specification P under the assumption A, and component M_2 satisifes A under no assumptions.

This style of reasoning is typically performed in an interactive fashion. Developers first check a component under no assumptions about the environment. If model checking returns a counterexample that is impossible for the system

under analysis, they make several attempts at manually defining an assumption that is strong enough to eliminate spurious violations, but that also appropriately reflects the remaining system [7, 12, 8, 10].

In [2], M. Cobleigh et al. present a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion. They follow an iterative approach based on the L* algorithm for learning regular languages. The learning process is based on queries to component M, and on couterexamples obtained by model checking M and its environment, alternately. Each iteration may conclude that the required property is satisfied or violated in the system analyzed. This process is guaranteed to terminate and it converges to an assumption that is necessary and sufficient for the property to hold in the specific system. However, the algorithm often terminates before reaching this point, and returns the first assumption that satisfies the requirements of the verification. Hence it makes sense to ask oneself whether the algorithm can be optimized with respect to certain properties of the returned assumptions.

A property of particular interest is the size of the generated assumptions. Since the assumptions capture those interactions between components which contribute to the non-violation of desired properties of the combined system, it may also be of interest outside of the specific verification context, e.g. the obtained assumption may reveal interesting aspects of the interaction. A smaller assumption corresponds to a less complex behavior, thus finding minimal adequate assumptions will allow for a better understanding of the behavior by humans. Another advantage of small assumptions arises in the context of security guarantees for software. A developer might want to allow its customers to re-run the assume-guarantee verification on delivered programs by supplying an adequate intermediate assumption. A smaller assumptions benefits both the developer who has to transmit it, as well as the user who re-runs the verification. The goal of this thesis is to develop a method that allows to obtain intermediate assumptions of minimal size.

This work is organized as follows: After introducing the L* algorithm for the learning of regular languages, we discuss the incremental framework by Cobleigh et al. for the automatic derivation of intermediate assumptions. Based on this work we then develop a search procedure which allows to compute intermediate assumptions of minimal size. Finally, we develop useful heuristics for improving the performance of our approach.

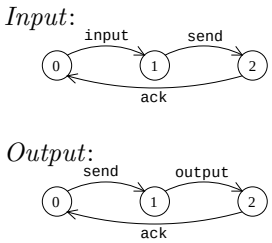

Figure 1: A simple communication channel

2 Definitions

2.1 Labeled transition systems

Both systems and assumptions are represented as Labeled Transition Systems, which allow the modelling of communicating components. Formally, let $\mathcal{A}ct$ be a set of observable actions. Furthermore let π denote a special **error state**, which will trap executions that violate the property under analysis and thus doesn't have any outgoing transitions. A **Labeled Transition System** (LTS) M is then a tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

- Q is a set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions, the **alphabet** of M
- $\delta \subseteq Q \times \alpha M \times Q$ is a transition relation
- $q_0 \in Q$ is a designated initial state

We call $M = \langle Q, \alpha M, \delta, q_0 \rangle$ **deterministic** if for all $(q_1, a, q_1'), (q_2, a, q_2') \in \delta$, $q_1' \neq q_2'$ implies $q_1 \neq q_2$. Furthermore let Π denote the LTS $\langle \{\pi\}, \mathcal{A}ct, \emptyset, \pi \rangle$.

Figure 1 shows two deterministic LTSs which are the components of a simple communication channel. We will use this system as running example throughout this thesis. Unless stated otherwise, the state labeled 0 will always be the initial state of any LTS.

The formal semantics of parallel composition will be given in the next section, but let us quickly describe the behavior of the system. Both components *Input* and *Output* are initially in state 0. *Input* may now perform a transition, labeled with `input`, and thus reach state 1. Think of this as some user issuing a request for some data to be sent over the channel. Now both components can simultaneously take their respective `send` transition. The data has now arrived in the receiving component *Output*, which signals this to its user by taking its `output` transition. Once this has happened, the two components acknowledge that the transmission was successful by taking their respective `ack` transition, again simultaneously.

A **trace** σ of an LTS M is a sequence of observable actions that M can perform starting at its initial state. Formally, $\sigma = a_1 a_2 ... a_n$ is a trace of

$M = \langle Q, \alpha M, \delta, q_0 \rangle$, iff there are states $q_1, ..., q_n \in Q$ such that $\langle q_{i-1}, a_i, q_i \rangle \in \delta$ for $i \in 1, .., n$. The set of all traces of M is called the **language** of M, denoted $\mathcal{L}(M)$. Furthermore, we use λ to denote the **empty word**.

For $\Sigma \subseteq \mathcal{A}ct$, we use $\sigma \upharpoonright \Sigma$ to denote the trace obtained by removing from σ all occurences of actions $a \notin \Sigma$. We call $\sigma \upharpoonright \Sigma$ the Σ-**abstraction** of σ.

We extend the restriction operator $\upharpoonright$ to languages by defining

$$\mathcal{L} \upharpoonright \Sigma := \{\sigma \upharpoonright \Sigma \mid \sigma \in \mathcal{L}\}.$$

We call a deterministic LTS that contains no π state a **safety LTS**. A **safety property** is specified as a safety LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. An LTS M **satisfies** P, denoted $M \models P$, iff

$$\forall \sigma \in \mathcal{L}(M) : (\sigma \upharpoonright \alpha P) \in \mathcal{L}(P).$$

When checking a property P, an **error LTS** denoted P_{err} is created, which traps possible violations with the π state. Formally, the error LTS of a property $P = \langle Q, \alpha P, \delta, q_0 \rangle$ is $P_{err} = \langle Q \cup \{\pi\}, \alpha P, \delta', q_0 \rangle$ where

$$\delta' = \delta \cup \{(q, a, \pi) \mid q \in Q \wedge a \in \alpha P \wedge \nexists q' \in Q : (q, a, q') \in \delta\}$$

Note that the error LTS is complete, meaning that each state other than the error state has outgoing transitions for every action in the alphabet. To detect violations of property P by component M, the parallel composition $M \mid\mid P_{err}$ is computed. M violates P iff the π state is reachable in $M \mid\mid P_{err}$ [2].

The *Order* property shown in Figure 2 captures a desired behavior of the communication channel shown in Figure 1. The property comprises states 0 and 1. The dashed arrows illustrate the transitions to the error state π that are added to the property to obtain its error LTS.

In order to check whether a specific string σ allows an LTS M_{err} to reach its error state π, we **simulate** σ on M_{err}. Formally, we do this by constructing a deterministic LTS A_σ that has σ as its only trace, and testing whether π is reachable in $A_\sigma \mid\mid M_{err}$. Given a string $\sigma = a_1 a_2 ... a_n$ over some alphabet Σ, we define the LTS A_σ as follows:

$$A_\sigma = \langle Q, \Sigma, \delta, q_0 \rangle$$

where

- $Q = \{q_0, q_1, ..., q_n\}$
- $\delta = \{(q_i, a_{i+1}, q_{i+1}) \mid 0 \leq i < n\}$

2.2 Semantics of parallel composition

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q_0' \rangle$. We say that M **transits** into M' with action a, denoted $M \xrightarrow{a} M'$, iff $(q_0, a, q_0') \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$ and $\delta = \delta'$ for $q_0' \neq \pi$, or, in the special case where $q_0' = \pi$,

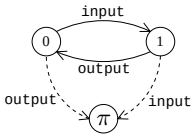

Figure 2: The *Order* property with added error transitions

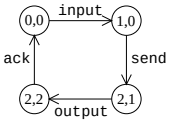

Figure 3: *Input* || *Output*

$M' = \Pi$. The parallel composition operator || is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions. Formally, let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q_0^1 \rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q_0^2 \rangle$ be two LTSs. If $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \mid\mid M_2 = \Pi$. Otherwise, $M_1 \mid\mid M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q_1 \times Q_2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows:

$$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \mid\mid M_2 \xrightarrow{a} M_1' \mid\mid M_2} \qquad \frac{M_1 \xrightarrow{a} M_1',\ M_2 \xrightarrow{a} M_2'}{M_1 \mid\mid M_2 \xrightarrow{a} M_1' \mid\mid M_2'}$$

Note that the symmetric rules are implied by the fact that the operator is commutative. Figure 3 shows the LTS *Input* || *Output*. Each state is labeled by a pair of indices, the first refers to the state of *Input*, the second to the state of *Output*. The initial state is labeled by $0, 0$.

2.3 Deterministic finite automata and LTSs

In the following, we will have to deal with Deterministic Finite-State Automata and convert them into safety LTSs. Let us recall that a **Deterministic Finite-State Automaton** (DFA) M is a five-tuple $\langle Q, \alpha M, \delta, q_0, F \rangle$ where:

- Q is a finite set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions that make up the alphabet of M
- $\delta : Q \times \alpha M \to Q$ is a transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

For a DFA M and a string σ, we use $\delta(q, \sigma)$ to denote the state that M will be in after reading σ starting at state q. A string σ is said to be **accepted** by a DFA $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ if $\delta(q_0, \sigma) \in F$. The **language accepted by** M, denoted $\mathcal{L}(M)$ is the set $\{\sigma \mid \delta(q_0, \sigma) \in F\}$.

If a DFA is minimal and its language is **prefix-closed** (i.e., for every $\sigma \in \mathcal{L}(M)$ and for every prefix σ' of σ, it holds that $\sigma' \in \mathcal{L}(M)$), then the DFA contains exactly one non-accepting state. A DFA with this property is called a **safety DFA**, since its language describes a (regular) safety property. Such a DFA can be transformed into a language-equivalent LTS by removing the non-accepting state and all its incoming transitions. As expected, the resulting LTS will be a safety LTS. Safety properties are among the properties that occur most often in practical verification problems [3].

2.4 Assume-guarantee reasoning

In the assume-guarantee paradigm a **formula** is a triple $\langle A \rangle M \langle P \rangle$, where M is a component, P is a property and A is an assumption about M's environment. The formula is true if, whenever M is part of a system satisfying A, then the system also guarantees P. To check an assume-guarantee formula $\langle A \rangle M \langle P \rangle$, where both A and P are safety LTSs, one computes the composition $A \mid\mid M \mid\mid P_{err}$ and checks if state π is reachable in the composition. If this is the case, then $\langle A \rangle M \langle P \rangle$ is violated by component M; otherwise it is satisfied.

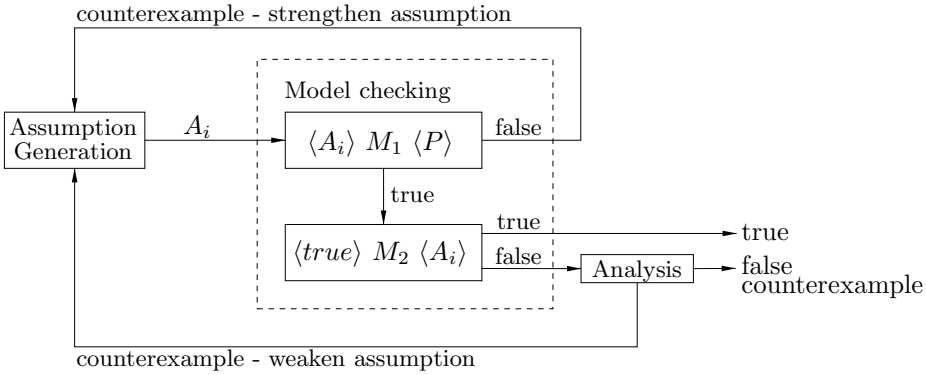

Figure 4: Incremental composition verification

3 Incremental compositional verification

In this section we describe the method presented in [2], which decides whether an assume-guarantee formula of the from $\langle true \rangle\ M_1\ ||\ M_2\ \langle P \rangle$ is true. Here *true* should be be understood as the empty assumption, that is, the assumption that does not disallow any behavior. The method is based on the following observation: if we can find a property A (expressed as a safety LTS) such that component M_1 satisfies property P under the assumption A and it is the case that component M_2 satisfies A without assumptions (i.e., under $\langle true \rangle$), then the system $M_1\ ||\ M_2$ will satisfy P. The following inference rule captures this proof strategy:

$$\frac{\langle A \rangle\ M_1\ \langle P \rangle \qquad \langle true \rangle\ M_2\ \langle A \rangle}{\langle true \rangle\ M_1\ ||\ M_2\ \langle P \rangle}$$

In the context of this rule, the property A is called an **intermediate assumption**. It is clear that the rule places certain requirements on A, e.g. it has to be strong enough (i.e. its language has to be small enough) for M_1 to satisfy P. On the other hand, it should not be too strong, since it needs to be satisfied by M_2. Thus the restrictions it places on M_2 should reflect M_2's behavior.

Finding an appropriate intermediate assumption is a non-trivial task, thus we will present an iterative method which converges to an appropriate intermediate assumption if one exists, and returns a counterexample to the assume-guarantee formula otherwise.

At each iteration i, a conjecture A_i is computed based on some knowledge about the system and on the results of the previous iteration. Model checking can then be used as described in Section 2.4 to check the two premises of the compositional rule.

$\langle A_i \rangle$ M_1 $\langle P \rangle$ is modelchecked first. If the result is false, model checking returns a counterexample, i.e. a trace of $A_i \mid\mid M_1 \mid\mid P_{err}$ ending in the π state. In this case, we know that the assumption was too weak, since it did not restrict the environment enough for P to be satisfied. The assumption therefore needs to be strenghened, which corresponds to removing behaviors from it. This is done using the counterexample. In the context of the next assumption A_{i+1}, component M should not exhibit the violating behavior reflected by this counterexample.

Once we find that the first premise of the compositional rule is satisfied by A_i, we also need to check the second one. This is done analogously to the previous step, i.e. $\langle true \rangle$ M_2 $\langle A_i \rangle$ is checked. If this check is successful, then by the compositional rule P holds in $M_1 \mid\mid M_2$. On the other hand, if model checking finds a counterexample, two situations are possible: either the assumption A_i was too strong, since it did not allow certain behaviors of M_2; or P is violated in $M_1 \mid\mid M_2$.

To distinguish between these two cases, the counterexample needs to be analyzed. The analysis concludes either that the counterexample does not represent a violation of P by $M_1 \mid\mid M_2$, in which case we proceed as before: the assumption A_i was too strong, so the counterexample is used to weaken the assumption in such a way that this particular counterexample is no longer allowed by A_{i+1}. Otherwise the analysis returns another trace which is a counterexample to $M_1 \mid\mid M_2$ satisfying P. In this case the whole algorithm terminates with that counterexample as output.

Finally, if both $\langle A \rangle$ M_1 $\langle P \rangle$ and $\langle true \rangle$ M_2 $\langle A \rangle$ are shown to hold, the algorithm has proven that $M_1 \mid\mid M_2$ satisfies P. Figure 4 gives a schematic view of the incremental framework.

3.1 The L* algorithm

We now take a closer look at the algorithms employed by the iterative method described above. The successive intermediate assumptions are computed by a learning algorithm introduced by Angluin [1] and later improved by Rivest and Schapire [11], the **L* algorithm**. It represents a general method for learning an unknown regular language based on queries and counterexamples.

Let U be an unknown regular language over some alphabet Σ. The L* algorithm interacts with a so called **Teacher** and an **Oracle**, to which it can send two types of queries.

- A **membership query** consists in asking the Teacher whether a string $\sigma \in \Sigma^*$ is in U. The Teacher answers $true$ or $false$.
- A **conjecture** is DFA C which is a candidate for the unknown language, the goal being $\mathcal{L}(C) = U$. The Oracle answers $true$ if this is the case;

		E	
		λ	b
S	λ	1	1
	a	0	0
$S \cdot \Sigma$	a	0	0
	b	1	1
	aa	0	0
	ab	0	0

Figure 5: An observation table for $\Sigma = \{a, b\}$, $S = \{\lambda, a\}$, $E = \{\lambda, b\}$ and $U = b^*$

otherwise it returns a counterexample, which is a string in the symmetric difference of $\mathcal{L}(C)$ and U.

The learning algorithm (which we call the **learner** in the following) will then start by making a number of membership queries to gain some partial knowledge about the unknown language. When certain conditions are met, which allow the algorithm to build a candidate DFA, it submits this as a conjecture to the Oracle. Typically the first conjectures won't match the target language, so the learner will use the counterexamples obtained from the Oracle to refine its hypothesis. If the target language is regular, this process is guaranteed to terminate with the learner producing a DFA C with $\mathcal{L}(C) = U$.

At any point during this process, the information the learner has about U can be represented as a partial mapping $\gamma : \Sigma^* \to \{0, 1\}$, where $dom(\gamma)$ is the set of strings for which membership queries have been performed.

In order to facilitate the transformation of the partial mapping into a DFA, the learner represents γ as an **observation table** $\mathcal{T} = (S, E, T)$. The observation table consists of two sets $S, E \subseteq \Sigma^*$, the *prefixes* and *suffixes*, and a function $T : S \cup (S \cdot \Sigma) \times E \to \{0, 1\}$, where $\cdot$ denotes string concatenation.

We can think of $\mathcal{T}$ as a table with rows labeled by the strings in $S \cup (S \cdot \Sigma)$, and columns labeled by the strings in E. The entry in row s, column e is then $T(s, e) = \gamma(se)$.

For a given observation table $\mathcal{T} = (S, E, T)$, we denote by $row_{\mathcal{T}}(s)$ the finite function f from E to $\{0, 1\}$ defined by $f = T(s, e)$, for $s \in S \cup (S \cdot \Sigma)$. We shall omit the subscript and write $row(s)$ whenever $\mathcal{T}$ is clear from the context.

Figure 5 shows an observation table for $\Sigma = \{a, b\}$, $S = \{\lambda, a\}$, $E = \{\lambda, b\}$ and $U = b^*$.

Let us recall **Nerode's right congruence**. Given a language $\mathcal{L} \subseteq \Sigma^*$, two strings $u, v \in \Sigma^*$ are said to be equivalent, written $u \equiv_{\mathcal{L}} v$, if for all $w \in \Sigma^*$, it holds that $uw \in \mathcal{L}$ iff $vw \in \mathcal{L}$. That is, two strings are equivalent if there does not exist a suffix that distinguishes between them.

In order to build an automaton M such that $\mathcal{L}(M) = U$, the learner would have to know $\equiv_U$. Since the learner has only partial information about U, it needs to approximate $\equiv_U$ by an equivalence relation over $S \cup (S \cdot \Sigma)$, using

1. $S \leftarrow \{\lambda\}$
2. $E \leftarrow \{\lambda\}$
3. **loop**
4. update T using queries
5. **while** T is not closed **do**
6. add sa to S to make S closed
7. update T using queries
8. construct candidate DFA C from T
9. submit C to Oracle
10. **if** C is correct **then**
11. **return** C
12. **else**
13. add $e \in \Sigma^*$ that witnesses the counterexample to E

Figure 6: The L* Algorithm

the current partial knowledge about U. Any closed observation table implicitly represents an equivalence relation $\simeq_{\mathcal{T}}$, defined by:

$$\forall s, s' \in S \cup (S \cdot \Sigma) : s \simeq_{\mathcal{T}} s' \Leftrightarrow row(s) = row(s')$$

That is, the suffixes in E are considered as potentially distinguishing experiments.

Initially, the algorithm sets S and E to be $\{\lambda\}$. The mapping γ is then updated by making membership queries until there is a value associated with every string in $(S \cup (S \cdot \Sigma)) \cdot E$. It then checks whether the observation table is **closed**, i.e., whether

$$\forall t \in S \cdot \Sigma,\ \exists s \in S : row(t) = row(s)$$

The observation table in Figure 5 is closed. In the original algorithm by Angluin [1], it was also required that the observation table be **consistent**, i.e., whenever $s_1, s_2 \in S$ such that $row(s_1) = row(s_2)$, then $\forall a \in \Sigma : row(s_1 a) = row(s_2 a)$. However, in the algorithm of Rivest and Schapire [11], any observation table produced during the learning process satisfies the invariant that whenever $s_1, s_2 \in S$ such that $s_1 \neq s_2$, then $row(s_1) \neq row(s_2)$, hence consistency holds trivially and doesn't need to be checked explicitely.

If the observation table $\mathcal{T}$ is not closed, then there exists some $t \in S \cdot \Sigma$ for which there is no $s \in S$ with $row(t) = row(s)$. In this case t is added to S and $\mathcal{T}$ is again updated by making membership queries.

Once $\mathcal{T}$ is closed, a candidate DFA $C = \langle Q, \alpha M, \delta, q_0, F\rangle$ is constructed as follows:

- $Q = \{row(s) \mid s \in S\}$
- $q_0 = row(\lambda)$
- $F = \{row(s) \mid s \in S \text{ and } \gamma(s) = 1\}$
- $\delta(row(s), a) = row(sa)$

The candidate C is presented as a conjecture to the Oracle. If the conjecture is correct, i.e. $\mathcal{L}(C) = U$, the L* algorithm returns C as correct, otherwise it receives a counterexample $c \in \Sigma^*$ from the Oracle. The counterexample c is analyzed by L* to find a suffix e of c that witnesses a difference between $\mathcal{L}(C)$ and U. This suffix must be such that adding it to E will cause the candidate to reflect this difference.

3.1.1 Distinguishing suffixes of counterexamples

Inuitively, a difference between $\mathcal{L}(C)$ and U means the current approximation $\simeq_{\mathcal{T}}$ of $\equiv_U$ is too coarse, i.e. there exist strings s and t that are equivalent under $\simeq_{\mathcal{T}}$ but not under $\equiv_U$. Hence the suffix e extracted from the counterexample c must be such that it distiguishes s and t, at least.

The following approach is due to Rivest and Schapire [11]. For some $s, s' \in S$ and $a \in \Sigma$ for which $row(s) = row(s'a)$, the suffix e should distinguish s and $s'a$, i.e. it should hold that $\gamma(se) \neq \gamma(s'ae)$. Let M_U be a minimal automaton with $\mathcal{L}(M_U) = U$ and $C = \langle Q, \alpha M, \delta, q_0, F\rangle$ the current candidate DFA.

Since $Q = \{row(s) \mid s \in S\}$ and because of the invariant mentioned earlier, for each $q \in Q$, $s \in S$ is uniquely determined by $q = row(s)$. Hence each state q can be associated with a unique string s. From the construction of the DFA C, it follows that s belongs to the equivalence class represented by q.

For $0 \leq i \leq |c|$, let p_i, r_i be such that $c = p_i r_i$ and $|p_i| = i$. Let s_i be the unique string determined by $row(s_i) = \delta(q_0, p_i)$, i.e. the state reached in C after the first i symbols of c have been read. Since c is a counterexample, we know that $\gamma(c) = 1$ iff $c \notin U$ and $\gamma(c) = 0$ iff $c \in U$. Now let

$$\alpha_i = \begin{cases} 1 & \text{if} \quad s_i r_i \in U \\ 0 & \text{if} \quad \text{otherwise} \end{cases}$$

Note that α_i can be determined by submitting $s_i r_i$ to the Teacher as a membership query.

Let us now assume without loss of generality that $c \in \mathcal{L}(C) - U$. Since $c = s_0 r_0 \notin U$, it follows that $\alpha_0 = 0$. On the other hand, since $c \in \mathcal{L}(C)$, it holds that $s_{|c|}$ is an accepting state of C, hence $\gamma(s_{|c|}) = 1$. But then by construction of γ, we have that $s_{|c|} \in U$. It follows that $\alpha_{|c|} = 1$ (note that $r_{|c|}$ is the empty string). Using binary search, it is possible to find some i such that $\alpha_i \neq \alpha_{i+1}$.

We can now show that r_{i+1} is the desired suffix e: let a be the first symbol of r_i. By definition of s_i, we have $s_{i+1} = \delta(s_i, a)$, hence $row(s_{i+1}) = row(s_i a)$. But if we add r_{i+1} to E and update γ, we get

$$T(s_i a, r_{i+1}) = \gamma(s_i a r_{i+1}) = \alpha_i \neq \alpha_{i+1} = \gamma(s_{i+1} r_{i+1}) = T(s_{i+1}, r_{i+1})$$

3.2 L* for assume-guarantee reasoning

In the context of incremental compositional verification as presented in Section 3, the L* algorithm is used to learn an intermediate assumption that can be used in the compositional rule.

An assumption with which the compositional rule (see Section 3) is guaranteed to work is the **weakest assumption** A_w, which restricts the environment of M_1 no more and no less than necessary for P to be satisfied. Assumption A_w describes exactly those traces over $\Sigma = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ which, when simulated on $M_1 \mid\mid P_{err}$ cannot lead to state π.

It is easy to see that for any environment component M_E, the formula $\langle true \rangle\ M_1 \mid\mid M_E\ \langle P \rangle$ holds iff $\langle true \rangle\ M_E\ \langle A_w \rangle$ holds. In the learning framework by Cobleigh et al. [2], L* learns the traces of A_w. The process returns as soon as both Oracles return *true*, which is often before the weakest assumption A_w is computed. We now look at how the Teacher and the Oracle are implemented in this framework.

Teacher. To answer a membership query for $\sigma = \langle a_1, a_2, ..., a_n \rangle$ in Σ^*, the Teacher simulates the query on $M_1 \mid\mid P$. That is, the Teacher checks if there is a trace σ' of $M_1 \mid\mid P$ leading to the error state, such that $\sigma' \restriction \Sigma = \sigma$. If such a trace exists, the Teacher returns $false$; otherwise, the answer to the membership query is *true*. In other words, the Teacher checks whether $\sigma \in \mathcal{L}(A_w)$.

Oracle. Due to the fact that the language $\mathcal{L}(A_w)$ that is being learned is prefix-closed, all conjectures returned by L* are also prefix-closed. Hence they can be transformed into safety LTSs as described in Section 2.3, which constitute the intermediate assumptions A_i.

The conjecture A_i must satisfy both premises of the compositional rule, hence the Oracle actually consists of two steps, which we call **Oracle 1** and **Oracle 2**. In the first step, or Oracle 1, $\langle A_i \rangle\ M_1\ \langle P \rangle$ is checked. If it succeeds, the second step, or Oracle 2, checks $\langle true \rangle\ M_2\ \langle A_i \rangle$.

Counterexample analysis needs to distinguish whether a counterexample c returned by Oracle 2 only means that A_i was too strong, or whether it shows that P is violated in $M_1 \mid\mid M_2$. To do this, $A_{c \restriction \Sigma}$ is computed and $\langle A_{c \restriction \Sigma} \rangle\ M_1\ \langle P \rangle$ is checked. If true, it means that M_1 does not violate P in the context of c, so $c \restriction \Sigma$ is returned to the learner as a counterexample for conjecture A_i.
If model checking fails with some counterexample c', then P is violated in $M_1 \mid\mid M_2$. A counterexample to $\langle true \rangle\ M_1 \mid\mid M_2\ \langle P \rangle$ is then generated by combining c and c'.

αM_1			=	$\{a, b, e, f\}$							
αM_2			=	$\{a, b, c, d\}$							
Σ =	$\alpha M_1 \cap \alpha M_2$		=	$\{a, b\}$							
c			=	abb	c	b		d	a		dd b
c'			=	abb		b	ee		a	ef		b
combination of c and c':				abb	c	b	ee	d	a	ef	dd b

Figure 7: Combination of counterexamples $c \in \alpha M_2$ and $c' \in \alpha M_1$

Combination of counterexamples works similarly to parallel composition of LTSs, in the sense that actions common to αM_1 and αM_2 are "synchronized" and the remaining actions of c and c' are interleaved. Figure 7 shows an example of combination, where interleaving inserts non-synchronized substrings of c' before non-synchronized substrings of c.

3.3 Example

Given the components *Input* and *Output* as shown in Figure 1 and the property *Order* shown in Figure 2, we will check $\langle true \rangle$ *Input* || *Output* $\langle Order \rangle$ by using the approach presented so far. The alphabet of the assumptions that will be used in the compositional rule is

$$\Sigma = (\alpha Input \cup \alpha Order) \cap \alpha Output = \{\texttt{ack}, \texttt{output}, \texttt{send}\}.$$

The first closed observation table $\mathcal{T}_1$ obtained by L*, along with its derived assumption A_1 is shown if Figure 8. A_1 is submitted to Oracle 1, which checks $\langle A_1 \rangle$ *Input* $\langle Order \rangle$. $\sigma = \{\texttt{input}, \texttt{send}, \texttt{ack}, \texttt{input}\}$ is a trace in A_1 || *Input* || $Order_{err}$ that leads to state π, hence Oracle 1 returns false, with counterexample $c = \sigma \restriction \Sigma$. From c, L* extracts the suffix `ack`, adds it to the set E of suffixes and updates the observation table. This table $\mathcal{T}_2$ along with its assumption A_2 is shown in Figure 9. This time, the assumption passes both Oracle 1 and Oracle 2, so L* terminates with the result that $\langle true \rangle$ *Input* || *Output* $\langle Order \rangle$ holds.

This example did not involve weakening of the assumption, since A_2 was sufficient for the compositional proof. This isw not always the case. For example, let us substitute *Output* by *Output'* shown in Figure 10, which allows multiple `send` actions to occur before producing `output`. The process will be identical to the previous one, until Oracle 2 is invoked for conjecture A_2. Oracle 2 returns that $\langle true \rangle$ *Output* $\langle A_2 \rangle$ is false, with counterexample $c = \langle \texttt{send}, \texttt{send}, \texttt{output} \rangle$. Counterexample analysis determines that in the context of this trace, *Input* does not violate *Order*. The trace is returned to the L* algorithm, which extracts the suffix `output` and uses it to weaken its assumption. The process involves two more iterations, during which assumptions A_3 and A_4 (Figure 11) are produced. Using assumption A_4, both Oracles report true, so L* terminates with the result that $\langle true \rangle$ *Input* || *Output'* $\langle Order \rangle$ holds.

$\mathcal{T}_1$	λ
λ	1
output	0
ack	1
output	0
send	1
output,ack	0
output,output	0
output,send	0

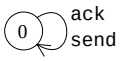

Figure 8: $\mathcal{T}_1$ and A_1

$\mathcal{T}_2$	λ	ack
λ	1	1
output	0	0
send	1	0
ack	1	1
output	0	0
send	1	0
output,ack	0	0
output,output	0	0
output,send	0	0
send,ack	0	0
send,output	1	1
send,send	1	1

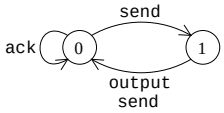

Figure 9: $\mathcal{T}_2$ and A_2

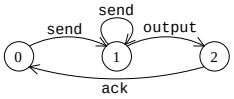

Figure 10: *Output'*

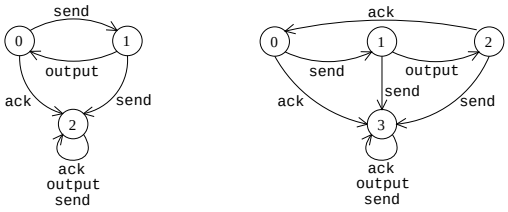

Figure 11: A_3 and A_4

4 Computing minimal assumptions

In the framework introduced so far, queries to the Teacher test for membership of a string in the language of the weakest assumption A_w (as defined in Section 3.2, but the procedure terminates as soon as the conditions of both Oracles are met. Remember that Oracle 1 removes words not in $\mathcal{L}(A_w)$ to be removed from the learned language, while Oracle 2 causes words from $\mathcal{L}(M_2)$ to be added. When the procedure terminates, the language of the intermediate assumptions it returns is thus a language U which satisfies $\mathcal{L}_2 \subseteq U \subseteq \mathcal{L}_1$ for two languages

$$\mathcal{L}_2 = \mathcal{L}(A_w)$$
$$\mathcal{L}_1 = \mathcal{L}(M_2) \upharpoonright \Sigma$$

In order to find a minimal intermediate assumption, we thus have to find a language that respects the above inclusion relations, while having a minimal number of equivalence classes and thus a minimal representation as an automaton. Obtaining smaller intermediate assumptions is interesting for several reasons:

- A smaller assumption means less complex behavior, hence such an assumption is easier for a human to understand. This is interesting, since the intermediate assumptions capture the interaction between components, which might be complex themselves.
- A party producing security-critical or otherwise essential software might want to allow its users to convince themselves that the software is correct by allowing them to re-run the verification. In order to save them the work of running the iterative scheme presented in the last sections, the producer might deliver the precomputed intermediate assumptions along with the actual components. The smaller the assumptions are, the easier it is to transmit them.
- Finally, assuming the same scenario as before, the user applies the compositional rule with the delivered intermediate assumption. Since checking of the premises is done by model checking of a parallel composition which has the assumption as one of its components, the computational cost of this check is influenced by the size of the assumptions.

4.1 Three-valued observation tables

Considering the inclusion relations mentioned above, the natural idea to minimize the intermediate assumptions is to take these inclusions into account already during the membership queries. We will thus relax the partial mapping γ by extending its range to include a third value $*$, which can be read as "don't know". Since we are trying to learn a language for which we only know a lower and an upper bound (w.r.t language inclusion), we will mark those strings which lie in the upper language but not in the lower language as "don't know" since they may or may not be in the language of the minimal automaton we want to

compute. Strings which lie outside the upper language cannot be part of the target language, and strings which are in the lower language have to be in the target language.

Upon receiving a membership query $\sigma = a_1, a_2, ..., a_n$ in Σ^*, the modified Teacher simulates the query on $M_1 \mid\mid P_{err}$, which corresponds to checking whether $\sigma \in \mathcal{L}_1$. If the result is negative, the Teacher returns 0. Otherwise, it simulates σ on M_{2err} to decide whether $\sigma \in \mathcal{L}_2$. If this is true, the Teacher returns 1, otherwise $*$.

We have now increased the accuracy of our observation tables by distinguishing between strings which lie in the lower language and those which do not. We still have to eventually transform those observation tables into automata, and we want these automata to be minimal.

If we want to apply the construction described in Section 3.1, we have to get rid of the "don't know" entries. Since a "don't know" entry means that the corresponding string may or may not be in the target language, it seams reasonable to instantiate the $*$ entries by 0 or 1. Determining how any such instantiation affects the size of the eventual automaton is non-trivial.

Hence our first approach consists in exploring all possible instantiations of all "don't know" entries by both 0 and 1. This will yield an exponential (in the number of $*$ entries) number of new observation tables at each step. We call such observation tables **instances** of the observation table they are derived from.

4.2 Wellformedness

We call an observation table $\mathcal{T} = (S, E, T)$ **wellformed**, if it defines a unique partial mapping γ. Note that different combinations of row and column labels can designate the same string in $(S \cup (S \cdot \Sigma)) \cdot E$. This is the case whenever $s, s' \in S \cup (S \cdot \Sigma)$, $e, e' \in E$ such that $s \neq s'$, $e \neq e'$ but $se = s'e'$. For the observation table to be wellformed, it must hold that the entries at se and $s'e'$ be identical, i.e. $T(s, e) = T(s', e')$. When instantiating an observation table, the wellformedness requirement thus constrains all $*$ entries representing the same string to take on the same value.

If $\mathcal{T}$ is wellformed, there is a unique function $\gamma : (S \cup (S \cdot \Sigma)) \cdot E \rightarrow \{0, 1\}$ with $T(s, e) = \gamma(se)$. In this case, we will use γ and T interchangeably. If an observation table contains $*$ entries, γ will actually be a partial function, remaining undefined on the strings which label $*$ entries in the table.

4.3 Prefix-closedness

Since the language we are learning is prefix-closed, we want to exclude from the search those observation tables which do not represent prefix-closed languages.

from the partial mapping γ represented by the observation table T. In practice, this can be checked using two functions has_prefix and is_prefix

	λ	b
λ	1	1
a	0	1
	...	

Figure 12: An observation table that is not prefix-closed

which get a set S of strings over Σ as first argument and a string of Σ^* as second argument and are defined as follows: .

- has_prefix is a function such that, for any $s \in \Sigma^*$, $has\_prefix(S, s) = 1$ if there exists $s' \in S$ and some string $e \in \Sigma^*$ (possibly empty) such that $s = s'e$, and 0 otherwise;
- is_prefix is a function such that, for any $s \in \Sigma^*$, $is\_prefix(S, s) = 1$ if there exists $s' \in S$ and some string $e \in \Sigma^*$ (possibly empty) such that $s' = se$, and 0 otherwise.

Now let

$$S_{accepted} = \{\ s \in (S \cup (S \cdot \Sigma)) \cdot E \ \mid\ \gamma(s) = 1\ \}$$

be the set of all strings accepted by the current mapping γ, and similarly let

$$S_{rejected} = \{\ s \in (S \cup (S \cdot \Sigma)) \cdot E \ \mid\ \gamma(s) = 0\ \}$$

be the set of strings rejected by γ.

We call the observation table T **prefix-closed**, iff there exists no string $s \in (S \cup (S \cdot \Sigma)) \cdot E$ such that both

$$has\_prefix(S_{rejected}, s) = 1$$

and

$$is\_prefix(S_{accepted}, s) = 1$$

hold.

Figure 12 shows part of an observation table that is not prefix closed, since here both $has\_prefix(S_{rejected}, a) = 1$ and $is\_prefix(S_{accepted}, a) = 1$ hold.

4.4 Closedness

Before we can build a candidate DFA out of a prefix-closed instance, we still need the instance to be closed. Remember that an observation table $\mathcal{T} = (S, E, T)$ is closed if

$$\forall t \in S \cdot \Sigma,\ \exists s \in S : row(t) = row(s)$$

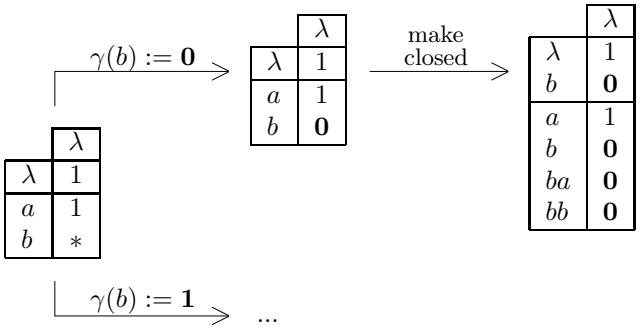

Figure 13: Instantiation and update without membership query

If the instance is not closed, new prefixes must be added, as before. Now however, we cannot simply add a prefix to the set S and update the observation table by making membership queries to the modified Teacher. If we add a prefix $s \in \Sigma^*$ where $row(s)$ has been modified by the instantiation, then by adding s to S and re-querying $row(s)$, we introduce $*$ entries again.

More generally, when updating an observation table that is the result of an instantiation, we need to take into account the information introduced by the instantiation. This can be achieved in the following way: before updating an observation table $\mathcal{T}$, record the partial mapping γ represented by $\mathcal{T}$ (assuming that $\mathcal{T}$ is wellformed). Note that if $\mathcal{T}$ is an instance of $\mathcal{T}'$, the partial mapping represented by $\mathcal{T}$ contains more information than the partial mapping represented by $\mathcal{T}'$. When $\mathcal{T}$ is updated and a string σ needs to be tested for membership, use γ before submitting any membership query to the modified Teacher. If σ has a prefix that is mapped to 0 by γ, then σ must be rejected. Similarly, if some string mapped to 1 by γ has σ as a prefix, then σ must be accepted. In practice, testing this can again be achieved using tries.

Figure 13 illustrates this idea with a simple observation table over $\Sigma = \{a, b\}$ that initially contains a * entry. The bold entries are deduced from the instantiation $\gamma(b) := 0$ without membership queries. Both ba and bb have b as a prefix, hence they must be rejected.

In general however, when updating an observation table, the partial mapping must be truly extended, in the sense that the membership of some strings cannot be determined without making a membership query.

4.4.1 Partial closing

Consider again the notion of closedness of an observation table. Since $*$ entries may be instantiated to either 0 or 1, our current definition of closedness does not make much sense for non-instantiated tables. On the other hand, it is

clear that if a non-instantiated table $\mathcal{T}$ contains some $t \in S \cdot \Sigma$ such that there is no $s \in S$ with $row(s) = row(t)$, then no instance of $\mathcal{T}$ will contain such an s either. Futhermore, if $row(t)$ does not contain any *, then making any instance of $\mathcal{T}$ closed will involve adding t to S. Hence a lot of work can be avoided by treating such cases before instantiating the observation table. But the absence of $*$ in a row alone is not a sufficient condition for adding the corresponding prefix to S. In order to maintain the consistency invariant described in Section 3.1, we must also make sure that no row in the upper part of the table can be instantiated to the row we are adding. Otherwise, the invariant $s_1 \neq s_2 \Rightarrow row(s_1) \neq row(s_2)$ could be violated in that particular instance. Hence, we will say that an observation table is partially closed whenever

$$\forall t \in S \cdot \Sigma, \text{ if } * \notin row(t) \text{ and } \nexists s \in S : inst(row(t), row(t)), \text{ then } \exists s' \in S : row(t) = row(s').$$

holds, where $inst$ is defined by

$$inst(r, r') \Leftrightarrow \forall e \in E : r(e) = r'(e) \vee r'(e) = *$$

$\mathcal{T}$	λ	b
λ	1	1
a	1	0
b	x_1	x_2
a	1	0
b	x_1	x_2
aa	0	0
ab	0	0
ba	x_3	x_4
bb	x_2	0

$\mathcal{T}'$	λ	b
λ	1	1
a	1	0
b	**0**	**0**
a	1	0
b	**0**	**0**
aa	0	0
ab	0	0
ba	**0**	**0**
bb	**0**	0

Figure 14: An observation with table constraints ($\mathcal{T}$), and the instance obtained from it by setting $\gamma(b) := 0$ ($\mathcal{T}'$).

Let us now consider a small example which will relate the notions of well-formedness and partial closing. Figure 14 shows an observation table $\mathcal{T}$ in which the "don't know" entries have been named x_1 to x_4. The variable x_1 represents the value of $\gamma(b)$, and x_2 stands for $\gamma(bb)$. x_3 and x_4 denote $\gamma(ba)$ and $\gamma(bab)$, respectively. Because of the wellformedness requirement, x_1 and x_2 appear several times in the table, meaning that any instance derived from this observation table must instantiate the entries $T(b, b)$ and $T(bb, \lambda)$ identically. Note that the table is already partially closed: there is no row $0, 0$ in the upper part of the table, however we can not add aa as a new prefix to S, because $inst(row(aa), row(b))$ holds. Indeed, by setting both x_1 and x_2 to 0, we would obtain the desired row. So let us consider what happens if we start instantiating

the variables. Setting x_1 to 0 is already sufficient to remove all "don't know" entries from the table: since all of bb, ba and bab have b as prefix, it follows that the only prefix-closed observation table derivable from $\mathcal{T}$ is $\mathcal{T}'$ shown (Figure 14). Note that $\mathcal{T}'$ is a closed observation table.

4.5 Treatment of suffixes

In the L* algorithm as described in Section 3.1, for each counterexample to the current conjecture, one suffix was extracted and added to the set E of suffixes. It turns out that in the case of three-valued observation tables, doing this naively can result in inconsistent instances.

Figure 15 shows an observation table and its derived conjecture (this observation table arose during experimentation with the Peterson mutual exclusion protocol, see Section 6.2). The alphabet is $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Note that the states are labeled by their corresponding prefixes in the observation table. The state labeled by λ is the initial state.

Oracle 1 determined that the language of the conjecture was too large, and returned the string 5.3.7.2.0 as a counterexample. The extracted suffix was 3.7.2.0. The observation table obtained after adding this suffix, applying partial closing and instantiating all $*$ entries by 0 is shown in Figure 16, along with the conjecture built from it (the entries resulting from instantiation are underlined).

This new conjecture indeed represents a finer equivalence relation $\simeq$, since the string 5 now has an equivalence class of its own, while it previously was equivalent to the empty string λ. However, the string 5.3.7.2.0 is still in the language of the new conjecture. The reason lies in the way we have instantiated the * entries in the new column 3.7.2.0. According to this instantiation, $row(5.3) = row(5)$, hence $5.3.7.2.0 \simeq 5.7.2.0$. By $row(5.7) = row(7)$, it follows that $5.7.2.0 \simeq 7.2.0$. Hence we obtain $5.3.7.2.0 \simeq 7.2.0$, but $T(7.2.0, \lambda) = 1$, so the string 5.3.7.2.0 is accepted. It is clear that the observation table can not be consistent, since $T(5, 3.7.2.0) = 0$, meaning that the string 5.3.7.2.0 should not be in the language of the conjecture.

But this inconsistency does not imply that the observation table is not well-formed, in the sense defined in above. This is because the inconsistency does not arise from contradictory information about the membership of strings from $(S \cup (S \cdot \Sigma)) \cdot E$, i.e. strings "directly" represented in the observation table, but rather from contradictions between this membership information and the equivalence relation $\simeq$ implicitely represented by the observation table.

Instead of testing each instance of an observation table for such contradictions, we will instead make sure that no instance can ever contain such contradictions. We can achieve this in the following way: whenever we get a counterexample to our current conjecture, we add not only the extracted suffix itself, but also ensure that the set E is always suffix-closed. Then the wellformedness requirement discussed in Section 4.2 will take care of maintaining consistency.

To see why this is the case, suppose we again have the table from Figure 15 as starting point. Oracle 1 again returns the string 5.3.7.2.0 as a counterexample, and we extract the suffix 3.7.2.0. Now we add to E the suffixes: 3.7.2.0, 7.2.0 and

2.0 (note that 0 was already in E). By membership queries we again obtain that $T(5, 3.7.2.0) = 0$. But now, by wellformedness, we also have $T(5.3, 7.2.0) = 0$. Assuming we have chosen an instantiation for the remaining * entries and made the resulting instance closed, there must now (by closedness) be a row labeled by some $s_1 \in S$ such that $row(5.3) = row(s_1)$. Now consider the entry $T(s_1.7, 2.0)$. By wellformedness, it must again be 0, and by closedness, there exists some $s_2 \in S$ such that $row(s_1.7) = row(s_2)$. Again by wellformedness, we have $T(s_2.2, 0) = 0$ and finally $T(s_3, \lambda) = 0$ for some $s_3 \in S$ with $row(s_2.2) = row(s_3)$. Hence the string 5.3.7.2.0 is equivalent to some rejected string, and hence rejected.

	λ	2	0
λ	1	0	0
1	0	0	0
7	1	1	0
7, 2	1	0	1
0	0	0	0
1	0	0	0
2	0	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	0	0	0
7	1	1	0
1, 0	0	0	0
1, 1	0	0	0
1, 2	0	0	0
1, 3	0	0	0
1, 4	0	0	0
1, 5	0	0	0
1, 6	0	0	0
1, 7	0	0	0
7, 0	0	0	0
7, 1	0	0	0
7, 2	1	0	1
7, 3	1	1	0
7, 4	1	1	0
7, 5	1	1	0
7, 6	0	0	0
7, 7	0	0	0
7, 2, 0	1	0	0
7, 2, 1	0	0	0
7, 2, 2	0	0	0
7, 2, 3	1	0	1
7, 2, 4	1	0	1
7, 2, 5	1	0	0
7, 2, 6	0	0	0
7, 2, 7	0	0	0

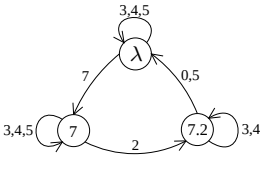

Figure 15: Observation table and corresponding conjecture LTS

4.6 Search procedure

Extending observation tables by adding prefixes or suffixes will in general introduce new $*$ entries, which again requires instantiation before any candidate DFA can be built.

Finding a conjecture of minimal size that passes both Oracles thus constitutes a search problem in a search space of observation tables. If we consider wellformed, closed instances to be successors of observation tables, we can apply classic search algorithms. We can conveniently define the size $|\mathcal{T}|$ of an observation table $\mathcal{T} = (S, E, T)$ as $|S|$. This way, the size of any closed instance corresponds to the number of states of its derived conjecture. If observation tables are "expanded", i.e. instantiated, in strictly increasing order of size (instantiation leaves the size of an observation table unchanged, whereas the

	λ	2	0	3, 7, 2, 0
λ	1	0	0	1
1	0	0	0	0
7	1	1	0	0
7, 2	1	0	1	0
5	1	0	0	0
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	1	0	0	1
4	1	0	0	1
5	1	0	0	0
6	0	0	0	0
7	1	1	0	0
1, 0	0	0	0	0
1, 1	0	0	0	0
1, 2	0	0	0	0
1, 3	0	0	0	0
1, 4	0	0	0	0
1, 5	0	0	0	0
1, 6	0	0	0	0
1, 7	0	0	0	0
7, 0	0	0	0	0
7, 1	0	0	0	0
7, 2	1	0	1	0
7, 3	1	1	0	0
7, 4	1	1	0	0
7, 5	1	1	0	0
7, 6	0	0	0	0
7, 7	0	0	0	0
7, 2, 0	1	0	0	0
7, 2, 1	0	0	0	0
7, 2, 2	0	0	0	0
7, 2, 3	1	0	1	0
7, 2, 4	1	0	1	0
7, 2, 5	1	0	0	0
7, 2, 6	0	0	0	0
7, 2, 7	0	0	0	0
5, 0	0	0	0	0
5, 1	0	0	0	0
5, 2	0	0	0	0
5, 3	1	0	0	0
5, 4	1	0	0	1
5, 5	1	0	0	0
5, 6	0	0	0	0
5, 7	1	1	0	0

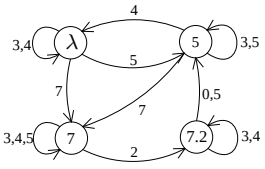

Figure 16: Observation table from Figure 15 after adding suffix 3.7.2.0, and corresponding conjecture LTS

addition of prefixes and suffixes causes it to become strictly greater), then any optimal search strategy will find a minimal intermediate assumption, if there is one.

4.6.1 Breadth-first search

One such optimal search strategy is breadth-first search, where uninstantiated, partially-closed observation tables are kept in a FIFO queue.

Figure 17 shows a breadth-first search procedure as a pair of mutually recursive procedures bfs_pop and $bfs\_instantiate$. Before bfs_pop is called, the FIFO queue q contains only the initial observation table with $S = E = \{\lambda\}$. The set $instances(\mathcal{T})$ at line 2 of bfs_pop is the set of all wellformed, prefix-closed instances of $\mathcal{T}$. It appears in the formal description of the search procedure for convenience, but in the actual implementation, the incremental method described in Section 5.4 is used to compute those instances.

procedure bfs_pop()

1. $T \leftarrow q$.pop()
2. bfs_instantiate$(T, 0, |\text{instances}(\mathcal{T})|)$

procedure bfs_instantiate$(\mathcal{T}, i, n)$

1. **if** $i = n$ **then**
2. bfs_pop()
3. $\mathcal{T}_i \leftarrow \text{instance}(\mathcal{T}, i) = (T_i, S_i, E_i)$
4. make_closed$(\mathcal{T}_i)$
5. **if** T_i contains * **then**
6. q.push$(\mathcal{T}_i)$
7. bfs_instantiate$(\mathcal{T}, i+1, n)$
8. construct candidate DFA C from $\mathcal{T}_i$
9. **if** *Oracle1*(C) fails with c **then**
10. extend E_i
11. partial_close$(\mathcal{T}_\rangle)$
12. q.push$(\mathcal{T}_i)$
13. bfs_instantiate$(\mathcal{T}, i+1, n)$
14. **else if** *Oracle2*(C) fails with c' **then**
15. **if** c' witnesses violation of P **then**
16. **return** Violation + ctrex
17. **else**
18. extend E_i
19. partial_close$(\mathcal{T}_\rangle)$
20. q.push$(\mathcal{T}_i)$
21. bfs_instantiate$(\mathcal{T}, i+1, n)$
22. **else**
23. **return** C

Figure 17: The *bfs_pop* and *bfs_instantiate* procedures

4.6.2 Iterative-deepening depth-first search

The main problem with breadth-first search as presented above, is that the queue has to hold a (potentially) exponentially growing number of observation tables. This makes the approach unpractical even for examples of modest size (see Section 6.2). Iterative-deepening depth-first search combines the space-efficieny of depth-first search with the optimality of breadth-first search. It proceeds by running a depth-limited depth-first search repeatedly, each time increasing the depth limit by one, but still has the same asymptotic complexity as breadth-first search. Only the path from the root (the initial observation table) to the current instance has to be kept in memory. This path is represented by a stack data structure. Figure 18 shows the iterative-deepening depth-first search procedure. Procedure *dls* applies depth-limited depth-first search, procedure *iddfs* calls *dls* with increasing depth limits. The initial table at line 3 of *iddfs* is again the non-instantiated observation table with $S = E = \{\lambda\}$.

procedure iddfs()

1. $maxd \leftarrow 0$
2. **while true do**
3. $path \leftarrow [initial\ table]$
4. dls$(maxd, 0)$
5. $maxd \leftarrow maxd + 1$

procedure dls$(maxd, d)$

1. **if** $path = empty$ **then**
2. **return** "no result"
3. **if** $d \geq maxd$ **then**
4. $path$.remove_top()
5. dls$(maxd, d-1, path)$
6. $\mathcal{T}_i \leftarrow path$.pop()
7. **if** $i \geq |$instances(parent$(\mathcal{T}_i))|$ **then**
8. $path$.remove_top()
9. dls$(maxd, d-1, path)$
10. construct candidate DFA C from $\mathcal{T}_i$
11. **if** *Oracle1*(C) fails with c **then**
12. extend E_i
13. partial_close$(\mathcal{T}_\rangle)$
14. $path$.push$(\mathcal{T}_i)$
15. dls$(maxd, d+1, path)$
16. **else if** *Oracle2*(C) fails with c' **then**
17. **if** c' witnesses violation of P **then**
18. **return** Violation + ctrex
19. **else**
20. extend E_i
21. partial_close$(\mathcal{T}_\rangle)$
22. $path$.push$(\mathcal{T}_i)$
23. dls$(maxd, d+1, path)$
24. **else**
25. **return** C

Figure 18: The *iddfs* and *dls* procedures

5 Reducing the search space

As we started implementing the algorithm described above, it became evident that even with the reduction in search space induced by requirements of well-formedness, prefix-closedness and partial closing, the number of instances that need to be explored can be very large (see Section 6.2 for experimental results).

One way of getting closer to a practically usable method is to introduce heuristics which reduces the search space, at the price of losing optimality, since they may cause the search procedure to terminate with a non-minimal intermediate assumption. Chosing which heuristics to use will hence involve a trade-off between speed and optimality, which may depend on the concrete problem.

An approach which potentially maintains optimality but lets the user decide when a result is satisfactory is discussed in Section 5.3.

5.1 Independent counterexamples

Assume that $C = \langle Q, \alpha M, \delta, q_0, F \rangle$ is a conjecture DFA built from some instance T_i of an observation table $\mathcal{T}$ that contains $*$ entries. If a counterexample c to C, returned by one of the two Oracles, can be shown to be independent of the specific instantiation, then c is guaranteed to be also a counterexample to any other instance T_j of $\mathcal{T}$. Hence if c was also a minimal counterexample to any instance of $\mathcal{T}$, the procedure from Figure 17 would submit each instance to the Oracle, obtain the same counterexample c and add the same suffix before adding the instance to the queue. In this case it would make sense to add the particular suffix directly to $\mathcal{T}$ and only then generate instances.

To determine whether the membership of some string $c = a_1 \dots a_n$ in the language $\mathcal{L}(C)$ is independent of the chosen instantiation, we look at the states that C goes through when reading c. By the construction of C (see Section 3.1), each transition $\delta(row(s), a) = row(sa)$ involves two rows of T_i, which also correspond to two rows of $\mathcal{T}$ (since $\mathcal{T}_i$ is an instance of $\mathcal{T}$). Hence the set of all rows which may influence the membership of c in $\mathcal{L}(C)$ is

$$\{\ row(s_{i-1}, a_i),\ row(s_i) \mid i = 1 \dots n \text{ and } s_0 = \lambda\ \}$$

If no row in this set contains $*$, then the membership of c in $\mathcal{L}(C)$ is independent of the instantiation of $\mathcal{T}$.

Hence the heuristic consists in checking, for every counterexample to a conjecture, if the counterexample is independent of the particular instantiation. If so, then we backtrack and add the obtained suffix to the "parent", i.e. the observation table from which the current instance was derived. This backtracking can be realized both in breadth-first and depth-first search.

A possible variation of the independent counterexample heuristic might consist in checking independence of counterexamples not only with respect to the last instantiation (as described above), but to take into account all entries that have been instantiated at some point in the search. If a counterexample can be shown to be independent wrt. to, say, the last n instantiation steps, the search procedure could backtrack n steps, back to the table to which the first instantiation was applied.

Another variation might consist in building non-complete automata from "*-free" parts of the observation table, i.e. using only rows which do not contain "don't know" entries. Counterexamples returned for such conjectures would then also be independent of the chosen instantiation. This approach remains to be investigated.

5.2 Conservative instantiation

Since our goal is to obtain an intermediate assumption of minimal size, a natural idea for limiting the search space is to instantiate $*$ entries in a way that introduces as little new rows into the table as possible. Remember that new rows are added whenever the observation table is not closed (see Section 3.1). Hence if we can match each row in the lower part of the table (those labeled by strings

from $S \cdot \Sigma$) to some row in the upper part of the table, no instantiation of the resulting table should make it necessary to add new rows. Figure 19 illustrates some aspects of this heuristic. The row labeled by t_1 can be matched to either s_2 or s_3, each matching inducing a different instantiation of $t_1 e_2$. Rows t_2 and t_3 can each be matched to s_1, but they place contradicting requirements on the value of $\gamma(s_1 e_2)$. No matter which value we chose for $\gamma(s_1 e_2)$, the table will not be closed and either t_2 or t_3 will have to be added to the set of prefixes.

	e_1	e_2
s_1	0	$*$
s_2	1	0
s_3	1	1
t_1	1	$*$
t_2	0	0
t_3	0	1

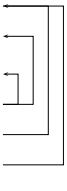

Figure 19: Conservative instantiation

This heuristic leaves some room for adjustments, since one may or may not want to explore all combinations of possible matchings. More specifically, in the example from Figure 19, there are four ways to match rows from the lower part of the table to rows in the upper part. In each case, a prefix (either t_2 or t_3) needs to be added. These four matchings happen to cover all four possible instances of the table. In the cases where a row can be matched to several different rows, or several different rows can be matched to one (like here), one might however limit oneself to fixing one combination of matchings.

But even if all combinations are considered, the heuristic still does not necessarily cover all instances of an observation table. The reason is illustrated in Figure 20.

	e_1	e_2
s_1	1	1
t_1	1	$*$
t_2	1	$*$

Figure 20: A situation in which conservative instantiation is incomplete

In this case, both lower rows would be matched to the only upper row, with the effect of making both lower rows identical. In general, whenever two or more lower rows can be matched to only one upper row, this induces equality constraints between the lower rows, which reduce the number of possible instances.

5.3 An interactive approach

Even when using heuristics presented so far, finding the first instance which passes both Oracles can require the exploration of a very large search space. Since it is difficult to give a general estimate of what consitutes an acceptable tradeoff between optimality and computation time, a natural idea is to let the user decide, based on the information he or she has about the system being analyzed. The approach we chose is that of an interactive depth-first search without depth limit. The search proceeds along one path, choosing always the first (or 0th) instance as successor, until it reaches an instance that passes both Oracles. Since the choice of the instantiation is arbitrary and depth-first search is not optimal, this first result may not be the smallest one. Now however, the users can judge wether the result is sufficiently small for his or her purposes. If the user decides that this is not the case, the search is resumed by backtracking to the next higher level and expanding the remaining instances.

5.4 Generating prefix-closed instances

As discussed in Section 4.3, only prefix-closed instances of observation tables need to be considered. Given an observation table $\mathcal{T}$ containing n $*$ entries (assuming we those entries which represent the same string in $(S \cup (S \cdot \Sigma)) \cdot E$ count as a single one), a naive approach might consist in generating all 2^n possible instances, testing each one for prefix-closedness and discarding those that are not prefix-closed. This is obviously a very expensive procedure, since exponentially many instances are generated and each instance needs to be tested for prefix-closedness, which takes time $O(n^2)$ in the worst case. It is hence desirable to have a procedure which generates exactly those instances that are prefix-closed, which then also allows to drop the test for prefix-closedness.

We consider the strict partial order $\prec$ over Σ^* defined by

$$s \prec s' \text{ iff } \exists s'' \neq \lambda : s' = ss''$$

Let S_* be the set of strings that label $*$ entries in $\mathcal{T}$. A partial mapping γ which assigns 0 or 1 to all strings in S_* represents a prefix-closed instance of $\mathcal{T}$ iff for all $s, s' \in S_*$, if $s \prec s'$, then

$$\gamma(s) = 0 \Rightarrow \gamma(s') = 0$$

holds.

Note that the smallest element of Σ^* wrt. $\prec$, λ, is not necessarily in S_*. In fact, it will usually be the case that $\lambda \notin S_*$, since otherwise we would have $\mathcal{L}(M_2) = \emptyset$. However, if we add λ to S_*, the Hasse diagram of S_* wrt. $\prec$ is a tree with root λ. To represent γ, we can label each node s of this tree by $\gamma(s)$. Then the trees representing prefix-closed instances will be exactly those in which no subtree with root labeled by 0 contains any nodes labeled by 1. We hence need a procedure that, given a tree, returns a list of trees of the same shape, representing all possible ways of labeling the nodes of the given tree

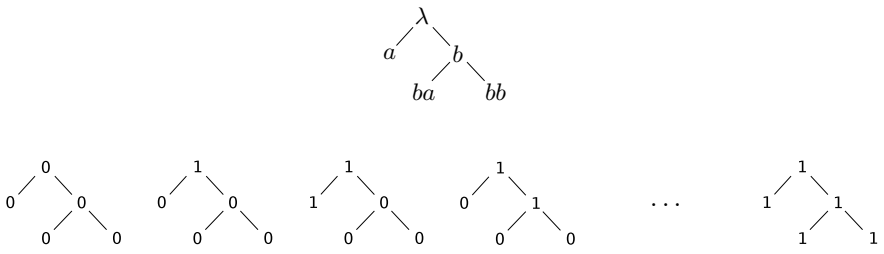

Figure 21: Hasse diagram and some prefix-closed variants for $\{a, b, ba, bb\} \cup \{\lambda\}$

which satisfy the prefix-closedness requirement. Call these trees **prefix-closed variants** of the original tree.

Figure 21 shows an example for $S_* \cup \{\lambda\}$ and $S_* = \{a, b, ba, bb\}$. The upper tree represents the Hasse diagram of $S_* \cup \lambda$ wrt. $\prec$. In the lower row, some of the variants returned by the procedure are shown. There are 11 of them. Depending on the depth of the tree, the number of prefix-closed variants can vary between linear (if the tree is linear) and exponential (tree of depth 1) in the number of nodes. Adding λ to S_* simply allows to represent each γ using a single tree, since otherwise there would be one separate tree for each minimal element in S_*. If λ originally was not in S_*, the first tree (representing $\gamma(\lambda) = 0$) can be discarded.

We will now give a formal description of the generating procedure. In ML-style syntax, the type of boolean-labeled trees can be written as follows:

```
type tree = T of bool * tree list
```

A recursive formulation of the procedure (call it *variants*) is then as follows (we assume that it is initially called with a tree whose nodes are all labeled by 0):

```
variants T(0,nil) = [T(0,nil), T(1,nil)]
variants T(0,ts) =
  T(0,ts) :: ( map (fn ts' -> T(1,ts')) (cross (map variants ts)) )
```

where `[.]` represents lists, `::` is the *cons* operator and *cross* computes a "cross product" on lists, i.e.

$$cross\ [[a_{1,1}, \ldots, a_{1,n_1}], \ldots, [a_{m,1}, \ldots, a_{m,n_m}]] = \\ [[a_{1,1}, \ldots, a_{m,1}], \ldots, [a_{1,n_1}, \ldots, a_{m,n_m}]]$$

This recursive formulation is obviously unsuitable for implementation since the computation of a single variant of a tree requires to complete the computation of all variants of all subtrees.

A better way of generating all prefix-closed variants of a tree is to label the nodes in a bottom-up fashion. Note that if we have chosen a labeling for all the

leaf nodes, we can propagate this assignment up the tree: any leaf that is labeled by 1 must also have its parent labeled by 1 and so on. After this propagation step, all leaves will be labeled, along with zero or more paths from leaves to the root. Removing the assigned leaves from the tree yields a new tree, in which zero or more leaves will be labeled by 1, all other leaves remaining unlabeled. For these unlabeled leaves, we can again chose a labeling and propagate it up, repeating this process until there are no more unlabeled nodes. By labeling any sequence of leaves in a fixed order (e.g., binary counting) we can thus enumerate all prefix-closed variants of a tree.

This procedure is again recursive, but can be made tail-recursive by making its stack explicit. The stack holds lists of leaves, together with their labeling. The advantage of this approach is that we can generate prefix-closed variants in a sequential fashion, computing only those subtree variants which are needed in the current variant. This is especially nice in the context of the independent-counterexample heuristic (see Section 5.1), since we might need to inspect only a very small number of instances before obtaining an independent counterexample. In this case it would be wasteful to compute all possible variants beforehand.

6 Implementation

In order to evaluate the practical usefulness of our minimization approach, an experimental version of it was implemented in the Objective Caml (OCaml) functional progamming language [5]. For the purpose of comparison, the learning framework presented in Section 3 was implemented, together with the the different search procedures developed in this thesis.

6.1 Implemented algorithms

We have implemented a collection of tools which allow to experiment with the algorithms which we have discussed.

- `basic` is our implementation of the original learning framework by Cobleigh et al. as presented in Section 3.2.
- `bfsmin` is an implementation of the breadth-first search procedure as described in Section 4.6.1
- `dfsmin` is an implementation of the iterative-deepening search procedure as discussed in Section 4.6.2 and can also be used to run a single (possibly depth-limited) depth-first search and also allows to use the interactive depth-first search described in Section 5.3.
- `compose` is a tool which allows to compute the parallel composition of LTSs, as described in Section 2.2

The independent counterexample heuristic (see Section 5.1) is available in both `bfsmin` and `dfsmin`. For usage information the reader is referred to the supplied documentation and user manual. The tools are available via `http://react.cs.uni-sb.de`.

6.2 Experimental results

We tested our implementation on a few small example systems.

Simple Channel

For the simple channel introduced in Section 3.3 (Figure 1), our search procedure returns the same result, i.e. the LTS A_2 shown in Figure 9. Hence for this simple system, the intermediate assumption returned by the original framework is already minimal.

Modified Channel

For the modified communication channel (Figure 10 from Section 3.3), the original framework computed an intermediate assumption of size 4 (A_4 in Figure 11). For this example, our search procedure found an intermediate assumption of size 2. The original intermediate assumtpion is shown again in Figure 22 for ease of comparison, along with our result.

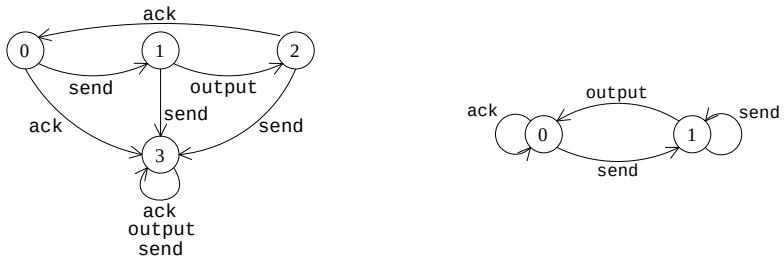

Figure 22: Intermediate assumption for the modified channel: the result by the original framework (left) and our minimal result (right)

Two Channels

In this slightly modified version of the simple communication channel, the sender can acquire messages via two different input actions, and then proceeds to send the message on one of two corresponding channels. The receiver acts analogously. The system, composed of the LTSs $Input_2$ and $Output_2$, is shown in Figure 23, along with the property $Order2$.

The intermediate assumption computed by the original learning algorithm is shown in Figure 24, along with our result.

Peterson's mutual exclusion algorithm

In [9], Peterson presents an elegant algorithm to solve the mutual exclusion problem for two processes, A and B, which use only shared memory for communication. There are three shared variables, x, y and $turn$. Both x and y

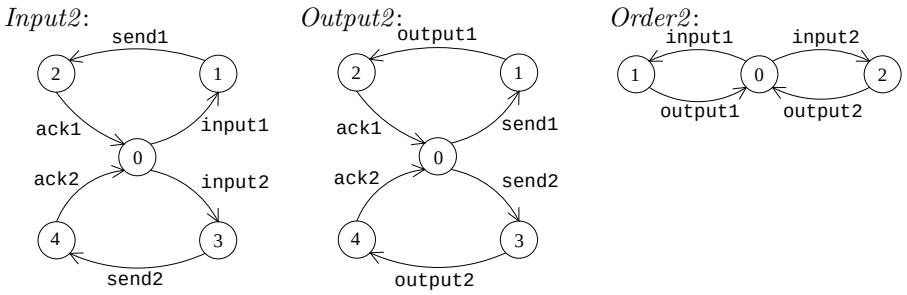

Figure 23: Communication over two channels

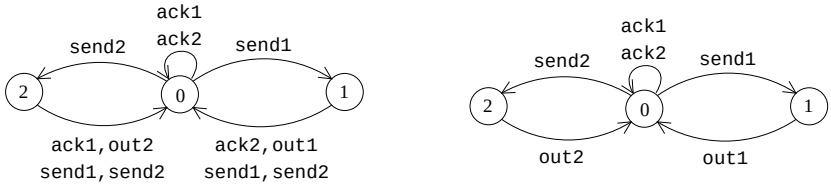

Figure 24: Intermediate assumptions for the system from Figure 23: the result by the original framework (left) and our minimal result (right)

are initially set to zero. The variable *turn* can hold one of two possible values A and B (we use these instead of 0 and 1 for clarity of presentation), and is initially set to A. The algorithm is shown in Figure 25.

Since every variable can only take one of a finite number of values, the state space of the composition of the processes A and B is finite. We have modeled both processes as LTSs. The LTS for process A is shown in Figure 26, the LTS for process B in Figure 27.

The states are labeled by four-tuples, which represent the shared variables as well as the program counter. The tuples are of the form $\langle turn, x, y, pc\rangle$ where the program counter pc counts the number of steps taken so far in each process. The program counter starts with 0 and is increased with each transition that the process makes. Both LTSs have $A000$ as respective initial state. Once a process has left it critical section, it returns to the initial state to start a new run.

Since our notion of labeled transition system does not include reading and writing of variables, we model the communication between the processes using the semantics of parallel composition, introduced in Section 2.2. Hence each process also contains those transitions which correspond to the writing of shared variables by the respective other process. The alphabet includes actions like $x = 1$, which are taken synchronously by both processes whenever process A sets x to 1. The actions $enterA$, $leaveA$ etc. indicate that a process enters or leaves its critical section. Since this does not affect shared variables, process A does not have any transitions labeled by $enterB$ or $leaveB$, and similarly,

Process A:

1. $x \leftarrow 1$
2. $turn \leftarrow B$
3. **while** $y = 1$ and $turn = B$ **do**;
4. enter critical section
5. leave critical section
6. $x \leftarrow 0$

Process B:

1. $y \leftarrow 1$
2. $turn \leftarrow A$
3. **while** $x = 1$ and $turn = A$ **do**;
4. enter critical section
5. leave critical section
6. $y \leftarrow 0$

Figure 25: Peterson's algorithm

process B does not have any transitions labeled by $enterA$ or $leaveA$.

These actions however do appear in the LTS of the mutual exclusion property, which is shown in Figure 28. Its initial state is labeled by 00.

For this example, the breadth-first-search was not practical, since it consumed too much memory. On the other hand, using iterative deepening search proved to take too much time and finally only a depth-first search without depth limit returned a conclusive result. This intermediate assumption (see Figure 29) has one state less than the one computed by the original framework (Figure 30), but it may not be minimal.

.

7 Conclusions

Based on previous work of M. Cobleigh et al. [2], this thesis introduced an approach to reducing the size of intermediate assumptions which arise during compositional verification in assume-guarantee style. The intermediate assumptions are constructed incrementally by a variant of Angluin's L* algorithm [1] for learning regular languages. The L* algorithm gains knowledge about a regular language by making membership queries for particular words, and constructing a sequence of conjecture automata. The approach uses the fact that the language of the intermediate assumption lies between two known regular languages, and makes use of this fact when answering membership queries of the learning algorithm. We have modified the learning algorithm such that it learns a language that is consistent with these bounds and has a minimal number of equivalence classes. This allows to represent the intermediate assumption by an automaton of minimal size. The possible conjectures that are consistent with the learning algorithm's partial knowledge form a search space which can be explored by a search procedure. We have discussed different ways to explore this search space, using well-known search paradigms. A number of heuristics to improve the efficiency of this search, possibly at the cost of minimality, were presented.

A basic version of the methods presented in this thesis has been implemented, allowing an evaluation of its practical benefits. Computation of minimal intermediate assumptions is possible for very small systems in the order of a few states, but the example of Peterson's mutual exclusion algorithm already reaches the limites of the current implementation. It is likely that the computational cost of finding minimal intermediate assumptions will drown the advantage gained by the smaller size of of the obtained assumptions.

We must thus conclude that the approach studied in this thesis, at least in its current form, is not suitable for use in practice.

In order to achieve efficiency for systems of realistic size, the implementation as well as the underlying concepts still need to be further elaborated.

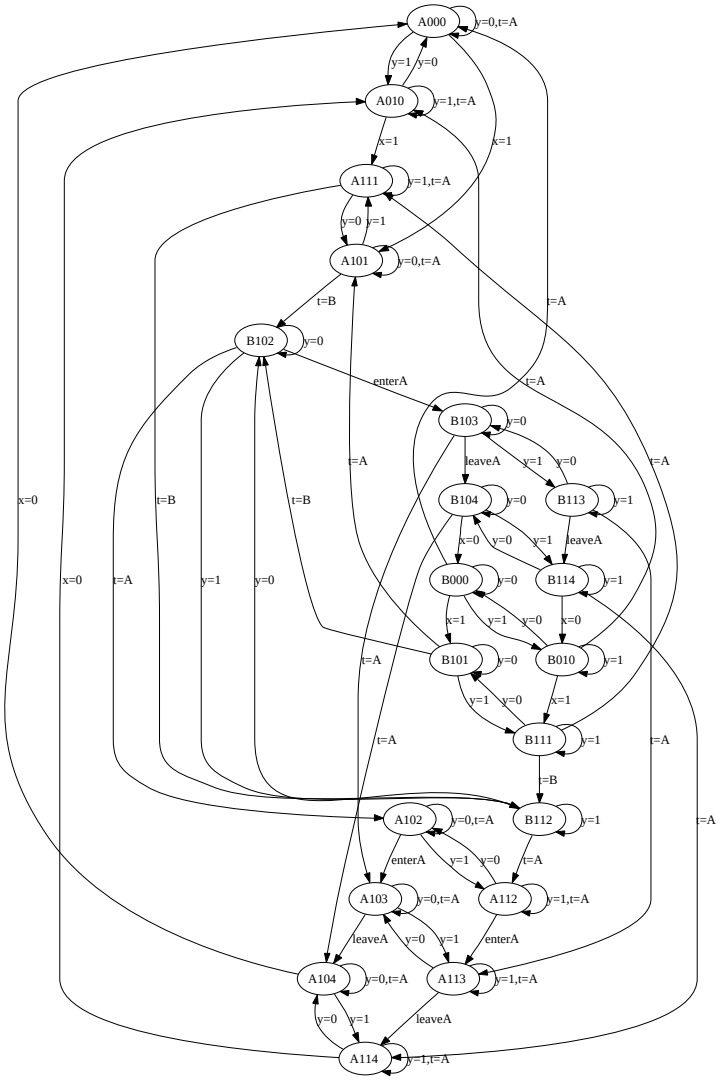

Figure 26: The LTS for process A in Peterson's algorithm

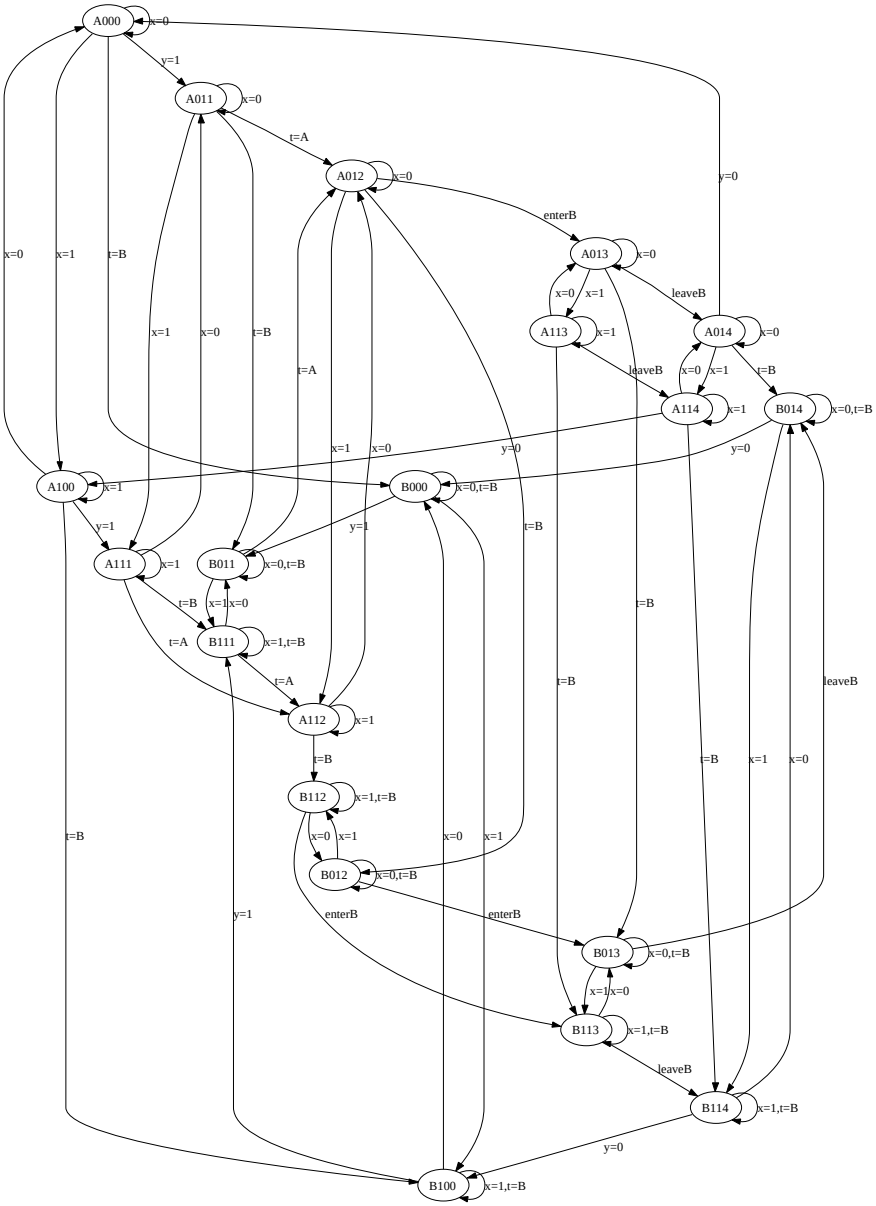

Figure 27: The LTS for process B in Peterson's algorithm

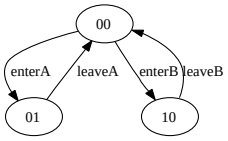

Figure 28: The mutual exclusion property for Peterson's algorithm

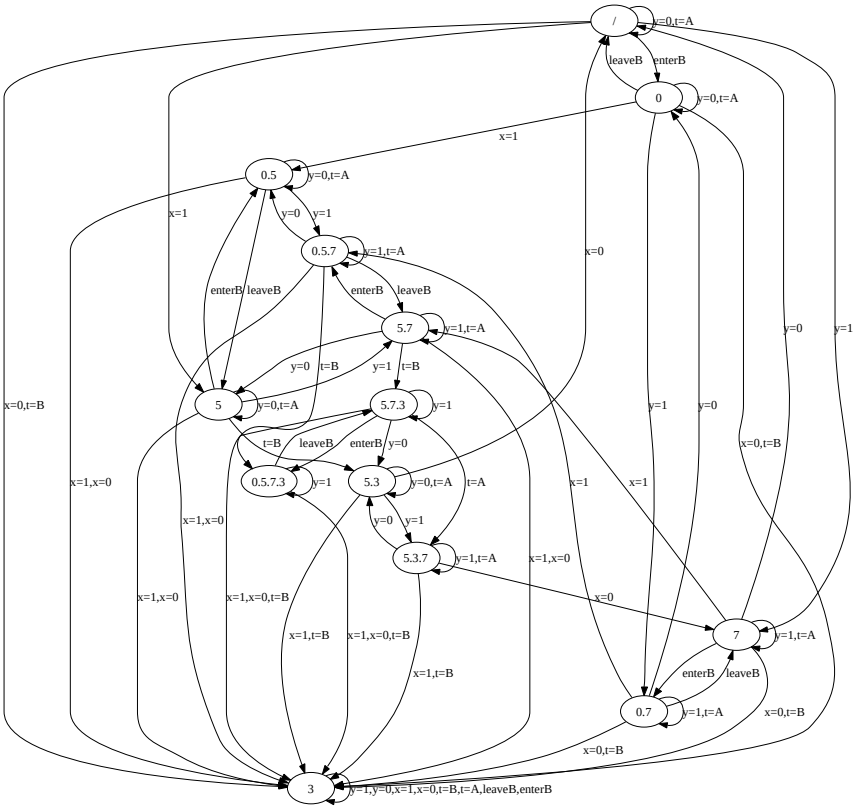

Figure 29: Intermediate assumptions for Peterson's algorithm as returned by the original framework

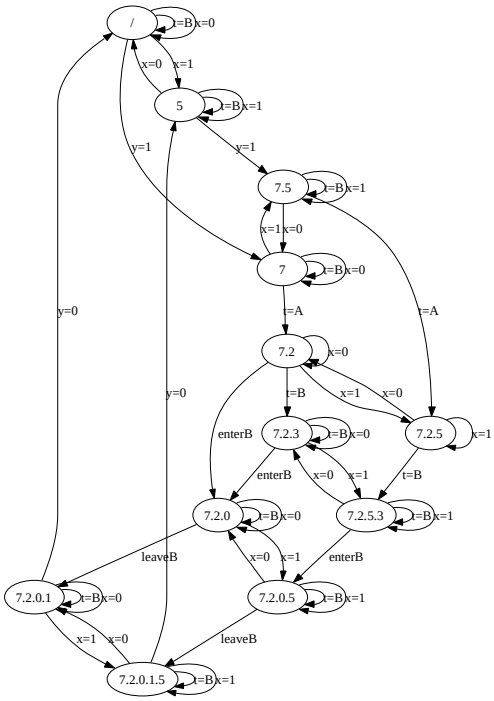

Figure 30: Intermediate assumptions for Peterson's algorithm as returned by depth-first search

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification, 2003.

[3] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[4] O. Grumberg E. M. Clarke and D. A. Peled. Model checking, 1999.

[5] French National Institute for Research in Computer Science and Control (INRIA). Objective caml. `http://caml.inria.fr/ocaml/index.en.html`, 2004.

[6] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[7] O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the 2nd Int. Conf. on Concurrency Theory*, pages 250–265, 1991.

[8] C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proc. of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

[9] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[10] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144. Springer-Verlag, 1984.

[11] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[12] S. Qadeer T. A. Henzinger and S. K. Rjamani. You assume, we guarantee: Methodology and case studies. In *Proc. of the 10th Int. Conf. on Computer-Aided Verification*, pages 440–451, 1998.