Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 250 (2009) 135–151
www.elsevier.com/locate/entcs

Compositional CSP Traces Refinement Checking

Heike Wehrheim[1] Daniel Wonisch[2]

Institut für Informatik
Universitt Paderborn
33098 Paderborn, Germany

Abstract

Compositional verification using assume-guarantee reasoning has recently seen an uprise due to the introduction of automatic techniques for *learning* assumptions. In this paper, we transfer this technique to a setting with CSP as modelling and property specification language, and present an approach to compositional traces refinement checking. The approach has been implemented using the CSP model checker FDR as *teacher* during learning. The implementation shows that the compositional approach can both drastically outperform as well as underperform FDR's performance, depending on the example at hand.

Keywords: Compositional verification, assume-guarantee rule, CSP, learning, refinement.

1 Introduction

Even after 25 years of model checking [9], the complexity of state spaces is still the major issue in an automatic verification of formally specified models. One factor of state explosion in complex systems are parallel components, because - in principle - the size of the state space grows exponentially in the number of components. Techniques to specifically treat this problem have been and are still under development. Among these are for instance partial order reductions [19], symmetry reductions [6], specific abstraction techniques for parallel systems [16] or compositional verification, the technique we will be interested in here.

Compositional verification [5] takes a divide-and-conquer approach to checking correctness: instead of verifying the system as a whole, only the system components are checked and the verification results are combined. One specific approach to compositional verification is *assume-guarantee (AG) reasoning* [12], [15]. Checking a system $S = S_1||S_2$ for the holding of a property $Prop$ is carried out in two

[1] Email: wehrheim@mail.upb.de
[2] Email: dwonisch@mail.upb.de

doi:10.1016/j.entcs.2009.08.022

steps: first, we show that S_2 guarantees $Prop$ under an assumption A about its environment, and then S_1 is shown to guarantee this assumption. In terms of a proof rule:

$$\frac{\langle true\rangle\ S_1\ \langle A\rangle \qquad \langle A\rangle\ S_2\ \langle Prop\rangle}{\langle true\rangle\ S_1||S_2\ \langle Prop\rangle}$$

For a long time this rule seemed to be impractical as it involved a manual identification of the assumption A. Recently, [8] proposed an algorithm for automatically generating assumptions when verifying safety properties. The approach uses the algorithm of Angluin [2] (and a successor refinement [17]) for *learning* regular languages (finite state automata). The basic idea is to have a teacher for answering questions like "is this a word in the language?" (membership queries) and "is this the correct automaton?" (equivalence queries). The answers of the teacher help to successively improve the initial guess of the learner. In case of compositional verification, the regular language is the assumption, the learner the algorithm for generating the assumption and the teacher a model checker.

In this paper, we transfer this approach to a CSP setting. CSP (Communicating Sequential Processes, [11]) is a process algebra used for describing parallel, communicating processes. Properties of CSP models of systems can be checked using one of the three *refinement orderings* traces, stable failures or failures divergences refinement. Here, we will be interested in safety properties of processes and thus employ traces refinement. For using the above sketched compositional verification with assumption learning in a CSP setting, we need to transfer (a) the above proof rule and (b) the membership and equivalence queries of learning to CSP. Moreover, we need a teacher, which in this case is straightforward since CSP comes with the model checker FDR [7] for checking all three refinement orderings and – as we will later see – both membership and equivalence queries can be rephrased as traces refinement checks. Besides the above given AG-rule, we have furthermore also transferred a parallel symmetric proof rule of [10] to CSP which employs a third query on emptiness of intersection of languages.

The thus transferred approach has been implemented in Java using a Tcl interface to FDR. A number of optimisations (e.g. recursive application of learning, caching of previously computed results, combination of basic and parallel proof rules) furthermore speed up verification. We carried out a number of experiments on artificial examples as well as ones from the literature. A comparison with FDR's performance on the full CSP specification shows that a compositional verification can drastically outperform a non-compositional verification but can also make things worse. The performance depends on the size of the assumption generated during learning, which in turn depends on the property under interest. For the example specification in this paper we both give a property for which the compositional approach turns out to be excellent as well as one for which FDR's performance is much better. Our implementation has already been successfully employed in [13], where

a decomposition technique for compositional verification is proposed and applied to the verification of the Two-Phase-Commit-Protocol.

2 Background

Since we base our compositional verification on the specification language CSP, some background information on CSP, Labeled Transition Systems (LTS), and traces refinements is needed. We briefly explain these basic concepts by providing the following example.

2.1 The scheduler

The example is based on a CSP specification of Simon Gay found in the sample folder of FDR. A similar version of the example is also described in [18] and in [14].

First of all imagine a set of jobs which are to be handled by a scheduler. Let n be the number of these jobs. A job can be started or finished. Therefore, when looking at a single job the *events start* and $finish$ may occur. In order to identify the job i with the events *start* and $finish$, we write $start.i$ and $finish.i$ respectively $(0 \leq i < n)$. After a job i has finished, it may be restarted using $start.i$.

The purpose of the scheduler is to maintain certain constraints for the execution of jobs. In this simple example there are only two constraints we are interested in:

(i) In between two *start*-calls of the job i, there must have occurred exactly one $finish.i$ event. Thus a job may only be restarted if it was finished before. Furthermore, a $finish.i$ event may only occur after a preceding execution of a $start.i$ event.

(ii) The jobs should be started "Round-Robin"-wise. This means that first job 0 is started, then job 1, then job 2, and so on until all n jobs have been started. After the last job has started, job 0 may be started again.

After having an intuition on how the scheduler should work, the task is now to express this system using the language CSP. Before actually constructing a CSP *process* fulfilling the constraints, we first focus on describing the constraints themselves as CSP processes. This will prove useful later.

The constraint (i) is very simple to model for a single job i in CSP. For a single job i, the constraint allows the events $start.i$ and $finish.i$ to occur only in turns. Such a sequence of events is called a *trace*. Therefore for instance the trace $\langle start.i, finish.i, start.i \rangle$ should be allowed for a CSP process $JobProp_i$ describing constraint (i). Since the job i can be executed infinitely often, the *set of all traces* of $JobProp_i$, denoted as traces$(JobProp_i)$, should contain infinitely many elements. To describe such a process in CSP we just need the prefix-operator, denoted by $e \rightarrow P$ for some event e and a process P. This operator is used to describe a process in which the event e is executed first and which behaves exactly as P afterwards. Using this operator, constraint (i) for a single job i may be described as follows:

$$JobProp_i = start.i \rightarrow finish.i \rightarrow JobProp_i$$

We now want to describe a CSP process $SchedProp$ expressing constraint (i) for all jobs. Basically any *interleaving* of the traces of $JobProp_i$ for $0 \leq i < n$ should be allowed. For example, the trace $\langle start.0, start.1, finish.0, finish.1\rangle$ is allowed by constraint (i) just as the trace $\langle start.0, finish.0, start.1, finish.1\rangle$ is. This can be expressed in CSP using the interleaving-operator $|||$:

$$SchedProp = |||_{i=0}^{n-1} JobProp_i$$

Similarly, constraint (ii) can be modeled as a CSP process using the prefix-operator again:

$$\begin{aligned} CycleProp_0 &= start.0 \rightarrow CycleProp_1 \\ CycleProp_1 &= start.1 \rightarrow CycleProp_2 \\ &\vdots \\ CycleProp_{n-2} &= start.(n-2) \rightarrow CycleProp_{n-1} \\ CycleProp_{n-1} &= start.(n-1) \rightarrow CycleProp_0 \end{aligned}$$

To make the idea of the *trace semantics* of CSP clearer, we want to transform these CSP processes to corresponding automata. In fact, CSP has an operational semantics allowing us to convert any CSP process P to a Labeled Transition System (LTS). If the LTS is finite state, we can view it as an automaton (by making all states final) and thus speak of the language of an LTS. When looking at the trace semantics of the CSP process P, we thus get that the set of allowed traces of P (traces(P)) is equal to the language of the LTS. Figure 1 shows the LTS for $SchedProp$ with $n = 2$ and Figure 2 shows the LTS for $CycleProp_0$ with $n = 5$.

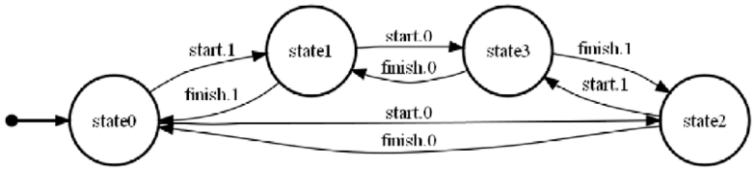

Fig. 1. $SchedProp$ with $n = 2$

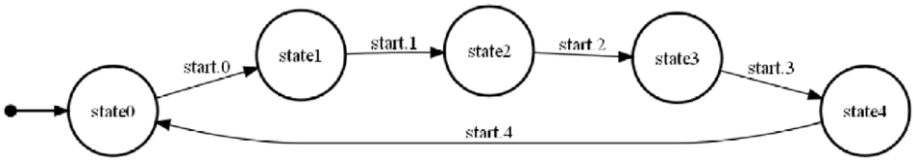

Fig. 2. $CycleProp_0$ with $n = 5$

Next, we design a scheduler obeying both constraints. To this end, we first introduce a new operator. The *parallel composition* operator is used to express that two CSP processes P_1, P_2 are executed concurrently, denoted by $P_1 \ {}_{X_1}||_{X_2} \ P_2$. Although P_1 and P_2 are executed concurrently, they may have to synchronise on some events. This is why the *alphabets* X_1 and X_2 are needed. An alphabet is a set of events,

and in this case X_1 basically describes the set of events which may occur in process P_1. The same applies to X_2 and P_2. If an event e is in the alphabet X_1 and not in X_2 and it can be executed by P_1, then P_1 can also execute the event e in the parallel composition $P_1 \ {}_{X_1}||_{X_2}\ P_2$ and vice versa. But if P_1 can only execute the event e' which is in the alphabet $X_1 \cap X_2$, then in the parallel composition P_1 has to wait until P_2 also executes e'. Therefore events $e' \in X_1 \cap X_2$ can only be executed jointly.

Using this concept the scheduler can be designed. First, we split the scheduler into so called *cells*. A cell of the scheduler can be either active or inactive. Initially, all cells but one shall be inactive. An inactive cell i just waits for some event $c.i$ to occur and switches to an active state afterwards. An active cell i will execute its corresponding job i using $start.i$ and activate the next cell by executing $c.inc(i)$, where inc is defined by $inc(i) := i+1 \bmod n$. After some time, $finish.i$ may occur since the job has finished execution. When the job is finished, the cell i again waits for the event $c.i$ to occur and then starts the cell again afterwards. If the event $c.i$ occurs before $finish.i$, the cell should just wait for $finish.i$ and execute $start.i$ afterwards immediately. These two possibilities can be expressed using the choice-operator in CSP, denoted by $P \,\Box\, Q$ for some CSP process P and Q. Summing up we get the following CSP processes:

$$
\begin{aligned}
ACell_i &= start.i \rightarrow c.inc(i) \rightarrow (finish.i \rightarrow c.i \rightarrow ACell_i \\
&\qquad \Box\ c.i \rightarrow finish.i \rightarrow ACell_i) && 0 \le i < n \\
IACell_i &= c.i \rightarrow ACell_i && 0 \le i < n \\
Cell_0 &= ACell_0 \\
Cell_j &= IACell_j && 0 < j < n
\end{aligned}
$$

These n cells are just executed concurrently synchronising on the events $c.i$:

$$Sched' = Cell_0 \ {}_{C_0}||_{C_1 \cup \ldots \cup C_{n-1}} (\ldots (Cell_{n-2} \ {}_{C_{n-2}}||_{C_{n-1}} Cell_{n-1}) \cdots),$$

where C_i is defined as $C_i = \{start.i, finish.i, c.i, c.inc(i)\}$. Since the events $c.i$ are just artificial events used by our implementation of the scheduler, we want to hide them in the traces of the process. CSP offers the *hiding* operator for this purpose, denoted by $P \setminus X$ for some CSP process P and an alphabet X. Using this operator we can hide the artificial events to get the final scheduler:

$$Sched = Sched' \setminus \{c.0, c.1, \ldots, c.(n-1)\}$$

To express that this CSP process guarantees the properties (i) and (ii), we can use *traces refinements* in CSP. A process P refines Q, denoted as $Q \sqsubseteq_T P$, iff traces$(P) \subseteq$ traces(Q). The constraints (i), (ii) should only allow traces defined by the CSP process $SchedProp$ and $CycleProp_0$, respectively. Therefore the required

properties of *Sched* can be simply written as:

$$SchedProp \sqsubseteq_T Sched$$
$$CycleProp_0 \sqsubseteq_T Sched \setminus \{finish.0, finish.1, \ldots, finish.(n-1)\}$$

In order to verify these refinements we can for instance use the CSP model checker FDR. Unfortunately, the parallel composition of the cells yields a exponential (more precisely $\Theta(n2^n)$, [18]) number of states in the corresponding LTSs. Therefore FDR needs a lot of time to verify the refinements, and even runs out of memory for large ns. This is why we are interested in a way of compositionally checking the refinements of our example by concentrating on the properties of the cells rather than on the complete system.

3 Compositional verification

For applying an assume-guarantee approach to traces refinement checking, and in particular the learning algorithm for assumption generation, we first need to rephrase the proof rule of the introduction and the learning in terms of CSP. We then apply the thus specialized technique to our example of the scheduler.

3.1 General approach

First of all, recall the basic proof rule from the introduction. We want to express the proof rule in terms of CSP now. Most obviously, the components *Prop*, S_1, S_2, and A need to be CSP processes. Furthermore we only consider systems S here which can be expressed as $S = S_1 \ {}_{\Sigma_1}||_{\Sigma_2} \ S_2$ for some alphabets Σ_1 and Σ_2. As seen in Section 2, we can express that a system S fulfills a property P using traces refinement. Hence we can express it as $P \sqsubseteq_T S \setminus (\text{Events} \setminus \Sigma_P)$, where Σ_P is the corresponding alphabet to P and the alphabet *Events* contains all possible events of the whole CSP specification. Finally, to express that system S_2 runs under an assumption A, we can use the parallel composition (i.e. $A \ {}_{\Sigma_w}||_{\Sigma_2} \ S_2$, where Σ_w and Σ_2 are the alphabets of A and S_2, respectively). Summing up we get the following proof rule:

$$\frac{A \sqsubseteq_T S_1 \setminus (\text{Events} \setminus \Sigma_w) \qquad Prop \sqsubseteq_T (A \ {}_{\Sigma_w}||_{\Sigma_2} \ S_2) \setminus (\text{Events} \setminus \Sigma_P)}{Prop \sqsubseteq_T (S_1 \ {}_{\Sigma_1}||_{\Sigma_2} \ S_2) \setminus (\text{Events} \setminus \Sigma_P)}$$

While the alphabets Σ_P, Σ_1, and Σ_2 are defined by the corresponding processes, the alphabet Σ_w can be more or less freely chosen. As we want to get the simplest assumption A possible, we use the smallest possible alphabet for Σ_w. The assumption A basically describes the behaviour of S_1 which is important for S_2 to guarantee *Prop*. Therefore, the alphabet of A should include all events which are in the alphabet of S_1 and either in the alphabet of S_2 or in the alphabet of *Prop*.

Thus we set Σ_w in this paper to:

$$\Sigma_w = (\Sigma_2 \cup \Sigma_P) \cap \Sigma_1$$

Using this alphabet, the above given proof rule is sound and complete ([4]). Given the CSP processes S_1, S_2, and $Prop$ the task is now to automatically find, if possible, an assumption A fulfilling the conditions of the proof rule. For this purpose we use a learning algorithm developed by Angluin ([2], improved by [17]) and proposed for assumption generation in [4]. To this end we split the task into two components: a learner and a teacher. Using the teacher, the learner can successively build assumptions A which may be sufficient for the proof rule. In fact, the learner successively tries to learn the so called *weakest assumption* A_w. The weakest assumption acts as A in the proof rule if the refinement

$$Prop \sqsubseteq_T (S_1 \;{}_{\Sigma_1}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$$

is true. Otherwise A_w does not fulfill the precondition of the proof rule. Then A_w acts as witness for the fact that the refinement is false. Formally A_w is characterised as follows:

Definition 3.1 Let S_2 and $Prop$ be CSP processes defined over the alphabets Σ_2 and Σ_P. Let Σ_1 be some alphabet. A CSP process A_w defined over the alphabet $\Sigma_w := (\Sigma_2 \cup \Sigma_P) \cap \Sigma_1$ is the *weakest assumption* for processes S_2 and $Prop$ iff for *any* process S_1 defined over the alphabet Σ_1 the following holds:

$$Prop \sqsubseteq_T (S_1 \;{}_{\Sigma_1}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P) \Leftrightarrow A_w \sqsubseteq_T S_1 \setminus (\text{Events} \setminus \Sigma_w)$$

Although the learner tries to learn the weakest assumption, other assumptions may be found as an intermediate step in the learning process, which are sufficient to show that the refinement holds or to show that it does not hold. In that case, of course, the learning algorithm terminates immediately instead of continuing learning the weakest assumption. In fact, this is the anticipated case.

It can be shown, that the weakest assumption is unique in sense of the trace semantics ([20]), and it contains the largest set of traces such that $Prop \sqsubseteq_T (A_w \;{}_{\Sigma_w}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$ holds ([20]). Furthermore the corresponding LTS for A_w is finite if the LTSs for S_2 and $Prop$ are finite ([20],[8]). This is fundamental for the learner since it can only learn *regular* languages. Figure 3 shows how the learner and the teacher work together to actually learn the weakest assumption.
The learning algorithm starts at the asterisks marked state. There, the algorithm tries to create an initial candidate for the weakest assumption. At the beginning the algorithm has no information at all about the weakest assumption. Thus the learner has to ask the teacher for the membership of some traces to get an idea on how the weakest assumption may look like. Technically, the learner actually starts filling a so called *observation table* containing all the observation knowledge. After some time, the learner has made enough observations to make a guess on how the weakest assumption may look like. Therefore it creates a candidate and asks the

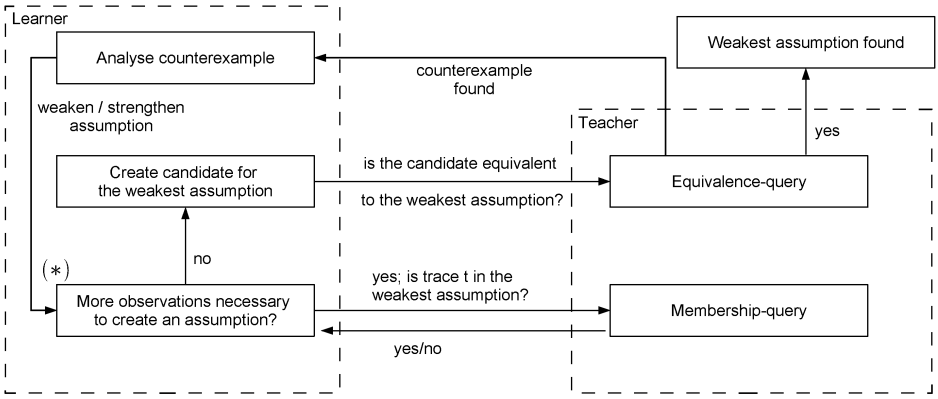

Fig. 3. Learning algorithm

teacher whether the candidate is correct or not. If the teacher returns *yes*, we are done and the algorithm finishes. Otherwise, the teacher returns a counterexample proving that the candidate was not correct. In that case the learning algorithm returns to its initial state. This time, however, the learner has knowledge about the already observed traces and additional knowledge about a trace which proves that the candidate was not correct. Hence the learner adds the counterexample to the observation table and tries to create a new, better candidate out of it. In order to achieve this, the learner will probably have to ask the teacher for the membership of some traces again. This whole process is repeated until the weakest assumption is found.

Given a correct teacher the learner terminates after at most $\mathcal{O}(n)$ equivalence-query calls and at most $\mathcal{O}(|\Sigma_w|n^2 + n \log m)$ membership-query calls, where n is the number of states of the weakest assumption and m the length of the longest counterexample returned by the teacher ([17]).

Note that the teacher has to answer two types of questions: membership-queries and equivalence-queries. In terms of CSP and with respect to the weakest assumption A_w, the teacher has to answer the questions "is this trace an element of the weakest assumption?" (membership-query) and "is this CSP process trace equivalent to the CSP process A_w?" (equivalence-query).

To implement the learner we use the algorithm proposed in [2] and [17]. However, we do not go into detail here how to actually implement the learner since it is mostly independent of the specification language used. Instead we focus on the teacher, beginning with the membership-query.

Membership-query

The teacher has to answer the question whether a given trace t is an element of the weakest assumption (i.e. check $t \in \text{traces}(A_w)$). It can be shown ([20]) that this can be answered by checking the refinement

$$Prop \sqsubseteq_T (CP(t) \;{}_{\Sigma_w}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$$

where

$$CP(\langle t_1, t_2, \ldots, t_n \rangle) := t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow \text{STOP}$$

is the process executing the events $t_1, \ldots, t_n$ and going into a deadlock (STOP) afterwards. Basically, the refinement checks for a given trace t whether under this single trace t, S_2 cannot violate $Prop$. If we put all these traces t, which do not violate $Prop$ together with S_2, into to an assumption A^*, then A^* would have in fact the largest set of traces which fulfills the refinement $Prop \sqsubseteq_T A^* \,{}_{\Sigma_w}||_{\Sigma_2}\, S_2$. Therefore $\text{traces}(A^*) = \text{traces}(A_w)$ holds.

Equivalence-query

This time the teacher has to answer for a given process A whether it is trace equivalent to the weakest assumption A_w (i.e. check $\text{traces}(A) = \text{traces}(A_w)$). If this is not the case, the teacher should provide a counterexample to the learner.
We now describe an algorithm for the equivalence-query which always gives the right answer to the question, but which may also terminate the complete algorithm by providing the result of the refinement check $Prop \sqsubseteq_T (S_1 \,{}_{\Sigma_1}||_{\Sigma_2}\, S_2) \setminus (\text{Events} \setminus \Sigma_P)$. The algorithm is based on [4] and works in three steps:

(i) We begin with checking the following refinement:

$$Prop \sqsubseteq_T (A \,{}_{\Sigma_w}||_{\Sigma_2}\, S_2) \setminus (\text{Events} \setminus \Sigma_P)$$

As seen above, the weakest assumption satisfies this refinement. Thus, if the refinement does not hold when using A instead of A_w, then A cannot be the weakest assumption. In that case we just return the counterexample provided by the refinement check as a proof for the conclusion $\text{traces}(A) \neq \text{traces}(A_w)$. If the refinement holds, we continue with step (ii).

(ii) Since we would like to complete our proof rule we now check the following refinement:

$$A \sqsubseteq_T S_1 \setminus (\text{Events} \setminus \Sigma_w)$$

If this refinement is in fact true, we can terminate our algorithm returning `true` since using the proof rule we verified that the system $S_1 \,{}_{\Sigma_1}||_{\Sigma_2}\, S_2$ guarantees $Prop$.
If we get a counterexample c for the refinement instead, we continue with step (iii).

(iii) With the previous steps we now know that the proof rule cannot be applied using the given assumption A. This may have two reasons: the assumption A was chosen badly or the refinement $Prop \sqsubseteq_T (S_1 \,{}_{\Sigma_1}||_{\Sigma_2}\, S_2) \setminus (\text{Events} \setminus \Sigma_P)$ does not hold. We now have to distinguish these two cases. We accomplish this by doing a membership-query for our counterexample c from step (ii):

$$Prop \sqsubseteq_T (CP(c) \,{}_{\Sigma_w}||_{\Sigma_2}\, S_2) \setminus (\text{Events} \setminus \Sigma_P)$$

If this is false, then $c \notin \text{traces}(A_w)$ holds. But in that case c is also a coun-

terexample for the following refinement:

$$A_w \sqsubseteq_T S_1 \setminus (\text{Events} \setminus \Sigma_w)$$

By definition of the weakest assumption (Def. 3.1) we get:

$$Prop \not\sqsubseteq_T (S_1 \;{}_{\Sigma_1}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$$

Therefore we terminate our algorithm and basically return the counterexample provided by the membership-query.

If the membership-query is true, $c \in \text{traces}(A_w)$ holds. Since c is a counterexample for step (ii), we have $c \notin \text{traces}(A)$. Thus $\text{traces}(A) \neq \text{traces}(A_w)$ holds and we return c as an evidence for this conclusion.

With the membership-query and equivalence-query explained we now have a complete teacher as needed for the learning algorithm. Hence we can next apply it to our example of the scheduler.

3.2 The scheduler revisited

Recall that we have a property $CycleProp_0$ and a system *Sched*. While our general approach is applicable to systems consisting of two components combined using parallel composition, *Sched* consists of n components (cells). This is no problem of course since we can just put for example $\lfloor \frac{n}{2} \rfloor$ cells together to form component S_1 and the others into component S_2 and combine both two components using parallel composition. For this example we consider $n = 4$ and put two cells into each component. Using our learning algorithm we get four generated assumptions over the alphabet:

$$\Sigma_w = \{start.0, start.1, c.0, c.2\}$$

Before looking at the generated assumptions, we first explain how the assumption for S_2 should look like. The assumption should contain the behaviour (or should impose restrictions) which S_2 needs to fulfill $CycleProp_0$. The most basic behaviour needed for such an assumption is that $start.1$ is only executed after a preceding $start.0$. Similarly, the assumption has to allow $start.2$ by executing $c.2$ only after a preceding $start.1$. Finally the assumption has to wait until S_2 finished executing $start.2$ and $start.3$ by waiting for $c.0$. Altogether, the assumption should at least allow any repetition of the trace $\langle start.0, start.1, c.2, c.0 \rangle$, but also not allow much more than this.

Figure 4 shows the generated assumptions. Note that we provide the LTS representation of the assumptions here. Internally the assumptions are handled as CSP processes which is needed for the refinement checks.

The last assumption indeed matches our expectation for an assumption for S_2. Note that $c.0$ executed in the start state for instance causes a deadlock in parallel composition with S_2, as S_2 waits for $c.2$ before synchronising on $c.0$. However, deadlocks are of course not forbidden by the property, any deadlocking process fulfills the safety property anyway.

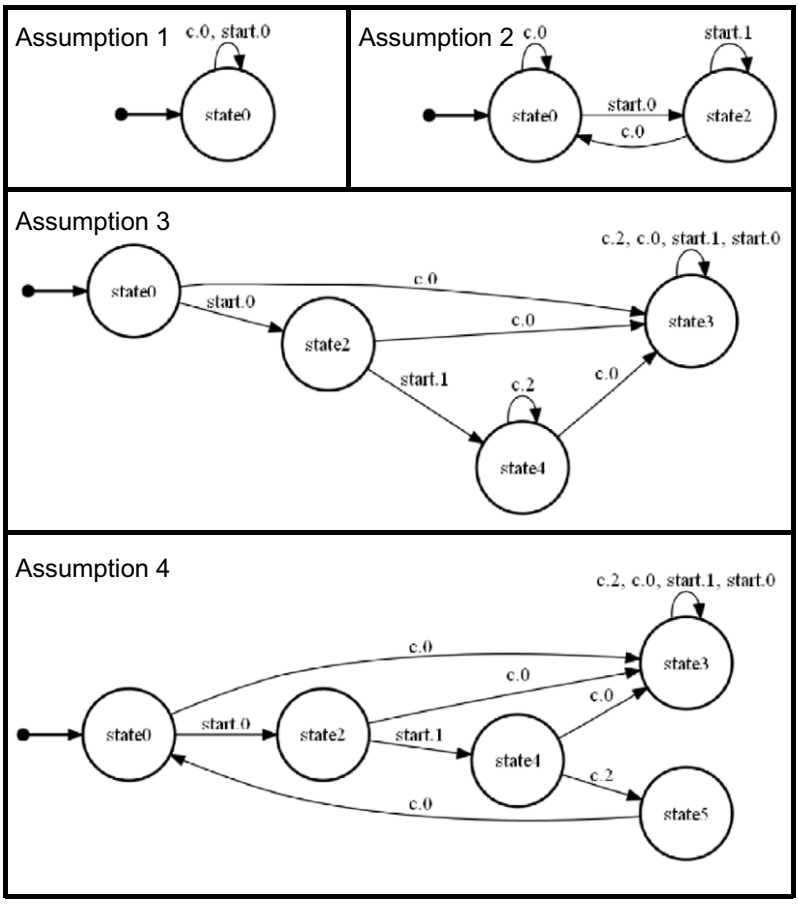

Fig. 4. Generated Assumption

On the other hand, the first assumption obviously allows too many traces since the assumption itself can create traces violating $CycleProp_0$. Therefore the teacher provides the trace $\langle start.0, start.0\rangle$ as an counterexample in the first step in the equivalence-query. The second assumption still allows too many traces. This time $\langle start.0, start.1, start.1\rangle$ is returned as a proof for this. Even the third assumption has not completed the chain of states which basically should only allow the trace $\langle start.0, start.1, c.2, c.0\rangle$. Hence the teacher returns $\langle start.0, start.1, c.2, c.0, start.1\rangle$ as a counterexample in the first step. The last assumption finally passes the first step of the equivalence-query. Furthermore, the assumption is not too restrictive, thus the second step refinement holds too. Using our proof rule the teacher terminates the learning algorithm with `true` as the result of the verification. Thus our scheduler does in fact fulfill constraint (ii) when using $n = 4$.

4 Implementation

In this section we first describe how we implemented the learning algorithm proposed in section 3.1. After that we compare the performance of our implementation of the algorithm with the performance of a direct FDR call.

We implemented the learning algorithm using the programming language Java [3]. The implementation reads in a refinement assertion like

$$Prop \sqsubseteq_T (S_1 \ {}_{\Sigma_1}||_{\Sigma_2} \ S_2) \setminus X$$

and checks if the refinement holds using the learning algorithm described in Section 3.1. We use FDR ([1]) to check the refinements needed for the membership-queries and equivalence-queries. The implementations consists of two main components (packages): a parser and the learning algorithm. The parser simply parses the input refinement given in FDR syntax ([7]). After this step the learning algorithm is started. The learning algorithm component can be divided into the learner part and the teacher part. For the teacher part there is an interface called `Teacher` which basically describes methods for membership-queries and equivalence-queries. The learner part is directly implemented in a `GenericLearner` class, which – generally spoken – uses a given `Teacher` to learn a regular language.

To actually implement the equivalence-query and membership-query as needed for the `Teacher` interface, we use as stated before FDR. In fact, to perform the refinement checks needed in the equivalence-query we use the batch mode of FDR which takes a CSP file as input and checks all traces refinement assertions given in the file. We then parse the result and – if applicable – the counterexample to use it in our implementation. Since there are not that many equivalence-queries needed, the overhead of creating a file containing the refinement checks is acceptable. However, in the case of membership-queries this overhead is not acceptable since there may be a large number of very simple to check membership-queries. Therefore, we use the Tcl interface to FDR for this purpose. This allows us to check the refinements directly without creating a new file and without creating a new FDR process for each query. Using this interface the queries can be handled very quickly in most cases.

Although the membership-queries can be handled quite quickly using the Tcl interface to FDR, there is still some optimisation possible. Most obvious, we can save many FDR calls by simply caching the membership-queries. Thus, whenever the learner asks for the membership of a trace t more than once, we can just return the cached result. This is of course faster than the FDR call. But we can save even more FDR calls by using properties of CSP. The set of traces of a CSP process is prefix closed ([18]), thus the following holds for some CSP process P, a trace t and a prefix s of t:

$$t \in \text{traces}(P) \Rightarrow s \in \text{traces}(P)$$

This is in particular true for the weakest assumption. We now use this property for

[3] http://www.cs.uni-paderborn.de/index.php?id=8967&L=1

two kinds of caching ([3]). First, the *prefix caching* checks for a trace t if there is a prefix s of t which is already cached. If there is a cached prefix s and $s \notin \text{traces}(A_w)$, then $t \notin \text{traces}(A_w)$ holds as well. Thus we return `false` in this case. Otherwise we use the so called *suffix caching*. In fact, we check for the shortest cached extension t' of t such that t is a prefix of t'. If $t' \in \text{traces}(A_w)$ holds, we can deduce that $t \in \text{traces}(A_w)$ holds and we return `true`. If this caching strategy fails as well, the query is checked using FDR.

Technically, we have classes `CachedAMSTeacher`, `PrefixCachedAMSTeacher`, and `SuffixCachedAMSTeacher` which implement the corresponding caching strategy. Furthermore we have classes `FDRAMSTeacher` and `FDRAEQTeacher` which implements the membership-query and equivalence-query, respectively, using FDR. All these classes can be added to a class called `GenericTeacher`, which basically coordinates added teachers to actually perform the query. By adding or removing instances of the cache-strategy classes we can turn the corresponding caching strategy on or off. By default, *caching* and *prefix caching* is enabled.

An alternative to the `FDREQTeacher` class is the `RecFDRAEQTeacher` class, which can handle equivalence-queries too. Contrary to the `FDRAEQTeacher` class, this class checks refinements by calling the learning algorithm once again if possible. Actually, it recursively checks the first step of the equivalence-query if S_2 consists of more than c components for some user defined constant c. The second step is checked recursively if S_1 consists of more than c components. The third step is checked using FDR since it is basically just a membership-query.

Care must be taken when recursively checking the first step of the equivalence-query to ensure that the algorithm terminates. If we applied the learning algorithm directly on $Prop \sqsubseteq_T (A \;{}_{\Sigma_w}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$ we would be actually trying to find an assumption A' fulfilling the conditions $A' \sqsubseteq_T A \setminus (\text{Events} \setminus \Sigma_w)$ and $Prop \sqsubseteq_T (A' \;{}_{\Sigma_w}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$ of the corresponding proof rule. Thus we would end up in an infinite recursion loop. Therefore we have to swap the order of A and S_2 before applying the learning algorithm. In that case we end up with the refinements $A' \sqsubseteq_T S_2 \setminus (\text{Events} \setminus \Sigma'_w)$ and $Prop \sqsubseteq_T (A' \;{}_{\Sigma'_w}||_{\Sigma_w}\; A) \setminus (\text{Events} \setminus \Sigma_P)$, allowing us to recursively apply the learning algorithm on $A' \sqsubseteq_T S_2 \setminus (\text{Events} \setminus \Sigma'_w)$ again.

The recursive call only makes sense if the direct call takes very long. Currently, we use the following heuristic: if the number of components is more than c, the direct call will take very long. This is a quite rough heuristic of course but due to a lack of other (better) heuristics we use this rough heuristic.

We now provide some experimental results. We ran the tests on a 3GHz (Pentium 4) Linux system (OpenSUSE 10.2) with 1GB RAM. In Table 1 we collected the runtime values of

- the direct FDR call,
- the implementation of the learning algorithm,
- the *recursive* implementation of the learning algorithm

when checking the refinement

$$CycleProp_0 \sqsubseteq_T SCHED \setminus \{finish.0, finish.1, \ldots, finish.(n-1)\}$$

of the scheduler for different n. We use the default caching settings for the implementation (i.e. caching and prefix-caching). Furthermore, for the non-recursive implementation we split the process $SCHED$ by putting $\lfloor \frac{n}{3} \rfloor + 1$ cells into the first component and the others to the second component. When using the recursive implementation we split by half and use $c = 8$.

n	Direct FDR call	Learning algorithm	Learning algorithm (recursive)
10	0.235s	1.348s	5.43s
13	2.247s	1.99s	8.27s
16	26.42s	3.173s	72s
19	297.5s	6.663s	106s
22	3290s	20.33s	145s
25	out of memory	91.25s	222s
30	out of memory	2067s	385s
35	out of memory	out of memory	874s
40	out of memory	out of memory	1534s

Table 1
Runtime for the scheduler using different approaches when checking for the property $CycleProp_0$

In Table 2 we collected the runtime values when checking the refinement $SchedProp \sqsubseteq_T SCHED$ using the same settings as above.
As we see here, for one property compositional verification is much better, for the other, a plain use of FDR. The reason for this is the size of the assumption generated during learning. When the assumption is much smaller than S_1 itself, compositional verification will outperform FDR, otherwise the overhead of learning will make the verification much slower. Thus, what we clearly need for a successful application of this technique is a means for determining beforehand, whether a compositional verification makes sense. This will be the topic of future work.

n	Direct FDR call	Learning algorithm	Learning algorithm (recursive)
3	0.043s	2.323s	N/A
4	0.044s	3.999s	N/A
5	0.047s	6.537s	N/A
6	0.051s	14.63s	N/A
7	0.066s	28.81s	N/A
8	0.093s	80.63s	N/A
9	0.163s	214.8s	268.1s
10	0.366s	1056s	600.3s

Table 2
Runtime for the scheduler using different approaches when checking for the property *SchedProp*

5 A parallel proof rule

Besides the basic assume guarantee proof rule, our implementation also supports the parallel proof rule as proposed in [10], here directly rephrased in CSP:

$$\frac{\begin{array}{c} Prop \sqsubseteq_T (A_1 \;{}_{\Sigma_w}||_{\Sigma_1}\; S_1) \setminus (\text{Events} \setminus \Sigma_P) \\ Prop \sqsubseteq_T (A_2 \;{}_{\Sigma_w}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P) \\ \overline{\text{traces}(A_1)} \cap \overline{\text{traces}(A_2)} = \emptyset \end{array}}{Prop \sqsubseteq_T (S_1 \;{}_{\Sigma_1}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)}$$

where $\overline{traces(Q)}$ is defined as the complementary set to $traces(Q)$ (i.e. $\Sigma_Q^* \setminus traces(Q)$). It is required that $\Sigma_P \subseteq \Sigma_1 \cup \Sigma_2$ holds. Furthermore the alphabet Σ_w of the assumptions is defined as:

$$\Sigma_w = (\Sigma_1 \cap \Sigma_2) \cup \Sigma_P$$

The main advantage of this parallel proof rule over the basic one is that it is symmetric. Thus we can independently learn assumptions for the components of the system at the same time. More precisely, the CSP processes A_1 and A_2 describe assumptions under which S_1 and S_2 can guarantee $Prop$. The third condition of the proof rule simply ensures that both assumptions do not disallow common traces. This is needed for the proof rule to be sound, since, if we for example used the deadlock process STOP for both assumptions (i.e. disallow *every* nonempty trace), the first two conditions would be trivially fulfilled for any CSP processes S_1 and S_2. If all three conditions of the proof rule hold, $Prop \sqsubseteq_T (S_1 \;{}_{\Sigma_1}||_{\Sigma_2}\; S_2) \setminus (\text{Events} \setminus \Sigma_P)$ also holds ([10]).

Since there are now two assumptions to learn, we also use two learning algorithms. The two learning algorithms again try to learn the weakest assumptions

$\Sigma_{w,1}$ and $\Sigma_{w,2}$. Therefore the membership-queries can be implemented analogically to the membership-queries introduced in Section 3. Both learning algorithms then create candidates A_i ($i \in \{1,2\}$) for the weakest assumptions. These candidates are checked whether they can be used for the proof rule or not. At first glance, the candidates need to fulfill the refinement $Prop \sqsubseteq_T (A_i \;{}_{\Sigma_w}||_{\Sigma_1} S_i) \setminus (\text{Events} \setminus \Sigma_P)$. This can be checked locally in the equivalence-query of the learning algorithms. When candidates fulfilling this are found, we need to check whether $\overline{\text{traces}(A_1)} \cap \overline{\text{traces}(A_2)} = \emptyset$ holds. In order to check this we transform this property to a refinement check:

$$\begin{aligned} & \overline{\text{traces}(A_1)} \cap \overline{\text{traces}(A_2)} = \emptyset \\ \Leftrightarrow\; & \overline{\overline{\text{traces}(A_1)} \cap \overline{\text{traces}(A_2)}} = \Sigma_w^* \\ \Leftrightarrow\; & \text{traces}(A_1) \cup \text{traces}(A_2) = \Sigma_w^* \end{aligned}$$

The right side can be taken as the set of traces of the CSP process run allowing every trace over the alphabet Σ_w (i.e. $run = \Box_{e \in \Sigma_w} e \rightarrow run$). The union of the trace-sets can be expressed using the choice-operator in CSP. Thus we just have to check $A_1 \;\Box\; A_2 =_T run$ which holds iff

$$A_1 \;\Box\; A_2 \sqsubseteq_T run$$

holds. This simple refinement can be checked using FDR again. If it holds, the proof rule is completed and we return `true`. Otherwise, we get a counterexample t. We then check for the membership of t in $traces(A_{w,1})$ and $traces(A_{w,2})$. If t is not member of both $traces(A_{w,i})$, it is a real counterexample and we return `false` ([10]). Otherwise, if t is member in $traces(A_{w,j})$ for some $j \in \{1,2\}$, we return t as a proof that A_j was not the weakest assumption to the corresponding learning algorithm ([10]).

Our implementation can make use of both rules during a refinement check. This is especially useful for implementing a recursive version of the parallel proof rule. In that case we actually call the algorithms for the two proof rules in alternation.

6 Conclusion

In this paper, we proposed a compositional technique for traces refinement checking in CSP. The technique was using basic assume-guarantee reasoning together with the assumption learning algorithm of [4]. The approach made use of the existing CSP model checker FDR, which was employed as the teacher during learning. Although the technique has not been implemented *within* FDR, but instead needed to call FDR everytime the teacher was required, it showed considerable improvement over a plain use of FDR for certain examples. The use of elaborate optimisation techniques and even a recursive application of AG reasoning brought us further speed-up.

In the future, we intend to look more deeply at heuristics determining when a compositional approach outperforms a non-compositional one, as to be able to automatically decide for or against a compositional verification.

References

[1] Fdr2. http://www.fsel.com, Version 2.83.

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

[3] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci.*, 118:3–18, 2005.

[4] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003. http://dblp.uni-trier.de/rec/bibtex/conf/tacas/CobleighGP03.

[5] W.P. de Roever, U. Hanneman, J. Hooiman, Y. Lakhneche, M. Poel, J. Zwiers, and F. de Boer. *Concurrency Verification*. Cambridge University Press, Cambridge, UK, 2001.

[6] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In D. A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254. Springer, 2000.

[7] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement - FDR2 User Manual*, sixth edition edition, 14 June 2005.

[8] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[9] O. Grumberg and H. Veith, editors. *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

[10] C. S. Pasareanu H. Barringer and D. Giannakopoulou. Proof Rules for Automated Compositional Verification through Learning. In *International Workshop on Specification and Verification of Component Based Systems, Finland*, September 2003.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985-2004.

[12] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[13] B. Metzler, H. Wehrheim, and D. Wonisch. Decomposition for compositional verification. In *International Conference on Formal Engineering Methods*, 2008. to appear.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[15] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[16] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2002.

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

[18] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[19] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, 1993.

[20] Daniel Wonisch. Automatisiertes kompositionelles Model Checking von CSP Spezifikationen. Bachelor's thesis, Universität Paderborn, April 2008 in German.