Compositional Proofs with Symbolic Execution

Simon Bäumler, Florian Nafz, Michael Balser, and Wolfgang Reif

Institut für Informatik – University of Augsburg
Augsburg, Germany

Abstract. A proof method is described which combines compositional proofs of interleaved parallel programs with the intuitive and highly automatic strategy of symbolic execution. As logic we use an extended variant of Interval Temporal Logic that allows to formulate programs directly in the Simple Programming Language (SPL). The notation includes a complex interleaving operator. The interactive proof method we use for temporal properties is symbolic execution with induction. Here, we show how to combine this proof method with an assumption-guarantee approach to decompose proofs for safety properties. We demonstrate the application of this technique with a producer-channel-consumer case study. [1]

1 Introduction

Verification of concurrent systems is an important topic, as, in comparison to sequential programs, the system execution is much more complex. Validation of concurrent systems by testing is very difficult and often not feasible, as there are many more test cases and it is hard to reproduce tests. But also formal verification of concurrent systems is complicated, because reasoning over all possible execution traces tends to result in a huge state space which makes automatic and interactive verification very difficult.

To avoid reasoning over the complete concurrent system, a common technique is compositional reasoning. The basic idea of this technique is, to split the system into several subcomponents. Then, the overall property is proved only with corresponding properties of the subcomponents. This idea was first formulated by Dijkstra [1]. In compositional reasoning the proof is often done with a compositional theorem. Such a theorem provides a number of proof obligations, which have to be fulfilled, so that the overall property is valid. Ideally, these proof obligations contain only single subcomponents and properties of these subcomponents, but not the complete system itself. This results in several proofs of feasible size.

A common compositional proof technique is the assumption-guarantee paradigm, which was introduced by Jones [2] and by Misra & Chandy [3]. The basic idea of this paradigm is, that each component can make specific assumptions to its environment in order to guarantee a specific behavior. An overview of recent works on compositionality in general can be found e.g. in de Roever et. al. [4] or Furia [5].

[1] This work has been funded by the DFG program INOPSYS II, under contract number Re 828/6-3.

Symbolic execution, on the other hand, is a successful technique for interactive verification of sequential programs (e.g. Dynamic Logic [6, 7]). It is a very intuitive strategy for programs as the proof advances step by step similar as most humans do it when trying to understand a program [8, 9]. Furthermore, it can be automated to a large extend. Balser [10] presented an ITL[2]-based logic with calculus that allows the symbolic execution of concurrent systems. This calculus was integrated into the interactive theorem prover KIV [12]. Arbitrary specification languages can be nested into this logic and thus making it unnecessary to translate a system specification into a special specification language for formal verification. Even more important is that the interleaving in this logic is compositional. That means, it is possible to replace a subcomponent with an abstraction of the component in a concurrent proof. While this feature simplifies concurrent proofs, it is still necessary to use symbolic execution on the whole parallel system in order to prove a property. A compositional theorem for this method would make it possible to prove properties of concurrent systems by reasoning only over single subcomponents at a time.

The goal of this paper is to present an assumption-guarantee rule for the logic presented in [10]. This would enable us to fully use the advantages of both techniques, compositional reasoning and symbolic execution, as well as the tool support, which is available for this logic.

We assume that the reader has at least basic knowledge in temporal logic and sequent calculus. The remainder of the paper is structured as follows: A short overview of our logic is given in Section 2. The compositional theorem we use is presented in Section 3, its application is shown in Section 4 on a producer-channel-consumer case study. Section 5 concludes the paper with related work and an outlook.

2 Temporal Logic Framework

In the following an informal overview over the used temporal logic calculus is given. The formal semantic is described in [13] and [10]. The calculus is integrated into the interactive theorem prover KIV. The temporal logic framework is a variant of ITL [11, 14] that is extended by explicitly including the behavior of the environment into each step. The basis for ITL are finite or infinite sequences π of valuations, which are called *intervals*. Valuations in π are called *states*. Each state is described by a first-order predicate logic formula over dynamic variables v, which also can be *primed* v' or *double primed* v''. The relation between v and v' is called *system transition*, whereas the relation between v' and v'' *environment transition*. The value of v'' in a state must be equal to the value of v in the next successive state. Thereby the system and the environment transition alternate. A selection of the supported temporal operators are:

[2] Interval Temporal Logic, introduced by Moszkowski [11]

$\circ\varphi$	there is a *next* state and it satisfies φ
last	the current state is the last state
$\Box\varphi$	φ holds *always* from now on in every state
φ **unless** ψ	either φ holds always from now on in every state or ψ holds in any state and φ holds in every state before
$\lceil \boldsymbol{v} \rceil$	*frame assumption*, only variables in $\boldsymbol{v}$ are modified
$\varphi_1 \parallel \varphi_2$	interleaving

Further, programs are written in a SPL (Simple Programming Language) [15] like program syntax. The selection of the used SPL-operators are:

$x := t$	assignment	**await** ψ	synchronization
$\varphi_1 \vert \varphi_2$	parallel assignment	**while** ψ **do** φ	loop
$\varphi_1 ; \varphi_2$	sequential composition		

Semantically, a program describes a set of traces. Therefore, it is possible to embed programs into temporal formulas. This can be used for the parallel composition of programs with the tl-interleaving operator.

2.1 Symbolic Execution

A typical sequent in proofs about interleaved programs has the form $P, \Gamma \vdash \Delta$. Here, P is the interleaved program, Γ contains a temporal formula that describes the environment behavior and a first order formula for the current variable assignment, while Δ contains the temporal property which has to be shown.

Symbolic execution on the following example sequent is done in two steps:

$$m := m + 1; <\text{prog}>,\ \Box m' = m'', m = 2\ \vdash\ \Delta$$

First, all temporal and program formulas are rewritten to a so called first-next form, which encodes the transition to the next state in a predicate logic formula. For this, the following rule[3] is used:

$$\frac{m' = m + 1,\ \circ <\text{prog}>,\ m' = m'',\ \circ\Box m' = m'',\ m = 2\ \vdash\ \Delta}{m := m + 1; <\text{prog}>,\ \Box m' = m'',\ m = 2\ \vdash\ \Delta} \quad \text{(prenex)}$$

This rule separates propositons about the current state from propositions about all following states. So after application of *prenex* each formula is either a first-order formula, describing the first state in the trace or a temporal formula with a leading *next*-Operator, that describes the remaining trace.

Now it is possible to advance one step in the trace. In all first-order formulas, unprimed and primed variables are replaced by new static variables, while the double primed variables are replaced by their unprimed version. Further, all

[3] Note that rules in the sequent calculus are read bottom-up, with the conclusion at the bottom and the corresponding proof obligations on the top part.

next-operators of temporal formulas are eliminated. In the example, this is done by the following rule-application:

$$\frac{M_1 = M_0 + 1,\ <\text{prog}>,\ M_1 = m',\ \Box m' = m'',\ M_0 = 2\ \vdash\ \Delta}{m' = m + 1,\ \circ <\text{prog}>,\ m' = m'',\ \circ\Box m' = m'',\ m = 2\ \vdash\ \Delta} \quad \text{(tl-step)}$$

This results in the following sequent after simplification:

$$<\text{prog}>,\ \Box m' = m'',\ m = 3\ \vdash\ \Delta$$

The rules for symbolic execution of formulas in the succedent are very similar. In KIV these rules, *prenex*, *tl-step* and simplification, are combined to a single complex rule called *step*.

2.2 Executing Interleaved Programs

To execute two interleaved formulas a first transition from one or the other formula is executed. After this, execution continues with interleaving the remaining formulas. For example, if there are two interleaved programs in the antecedent $m := 1; \ldots \parallel n := 2; \ldots,\ \Gamma\ \vdash\ \Delta$ this formula is executed by symbolically executing either program first. For this, the following rule is used:

$$\frac{\begin{array}{c} m := 1; (\ldots \parallel n := 2; \ldots),\ \Gamma\ \vdash\ \Delta \\ n := 2; (m := 1; \ldots \parallel \ldots),\ \Gamma\ \vdash\ \Delta \end{array}}{m := 1; \ldots \parallel n := 2; \ldots,\ \Gamma\ \vdash\ \Delta} \quad \text{(interleaved left)}$$

Furthermore the following equation holds for the interleaving operator

$$\textbf{last} \parallel \phi \quad \leftrightarrow \quad \phi$$

which can be used to eliminate terminated programs. In the case that one of the programs is blocked, only the other program is executed.

One important feature of our interleaving operator is that it is compositional. This means, that the following rule can be applied:

$$\frac{\vdash\ \varphi_1 \rightarrow \varphi_2 \qquad \varphi_2 \parallel \psi,\ \Gamma\ \vdash\ \Delta}{\varphi_1 \parallel \psi,\ \Gamma\ \vdash\ \Delta} \quad \text{(comp)}$$

This feature is very important for the proofs of the theorems in chapter 3 and for abstraction in general.

Note, that our interleaving operator also supports features like fairness and blocking. These features and the general case, where the interleaving operator contains arbitrary temporal formulas, are also described in detail in [16] or [10].

2.3 Induction and Sequencing

The basic idea to proof safety properties is to advance in the interval until a valuation is reached that was considered earlier in the interval, so that a loop was executed. If it can be proven that the property is true before and during the loop so it is invariant, then the proof can be finished with an inductive argument. A special rule *start induction* is used to generate a suitable induction hypothesis.

Symbolic execution can lead to many paths, that have to be explored. Often two different paths lead to same configurations (two sequent have the same configuration if all temporal logic formulas are the same). To minimize the proof effort a rule called sequencing is used, that allows to close a open premise when there exists another premise with the same configuration, but with more general predicate logic formulas.

3 Compositional Theorem

Most assumption/guarantee based compositional proof techniques use a special operator similar to the "while-plus" operator $\overset{+}{\rightarrowtriangle}$ presented in [17]. Informally, the term $A \overset{+}{\rightarrowtriangle} G$ means, that if A holds up to step i, then G must hold up to step $i+1$. This operator enables the formulation that a component violates its guarantee G only after its assumption A is violated. It is needed to break the circularity of the used compositional rule.

Assumptions and guarantees can be formulated with propositional predicates over unprimed and primed variables (e.g. Cau and Collette [18]). We use the same approach, but for the assumptions we use predicates over primed and doubly primed variables. In this way it can be formalized which steps are allowed for the components and which steps are allowed for the environment. This also allows to use a standard TL operator **unless** as $\overset{+}{\rightarrowtriangle}$ operator, i.e.:

$$A \overset{+}{\rightarrowtriangle} G := G\,\textbf{unless}\,(G \wedge \neg A)$$

With these preliminaries we are able to construct a compositional theorem:

Theorem 1. *If:*

i. *for all* $i = 1, \ldots, n$: $M_i \vdash A_i(v', v'') \overset{+}{\rightarrowtriangle} G_i(v, v')$
ii. *for all* $i = 1, \ldots, n$: $G_i(v_1, v_2) \vdash G(v_1, v_2) \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} A_j(v_1, v_2)$
iii. *for all* $i = 1, \ldots, n$: $A_i(v_1, v_2) \wedge A_i(v_2, v_3) \vdash A_i(v_1, v_3)$
iv. $A(v_1, v_2) \vdash \bigwedge_{i \in \{1..n\}} A_i(v_1, v_2)$

then:

$$M_1 \parallel \ldots \parallel M_n \vdash A(v', v'') \overset{+}{\rightarrowtriangle} G(v, v')$$

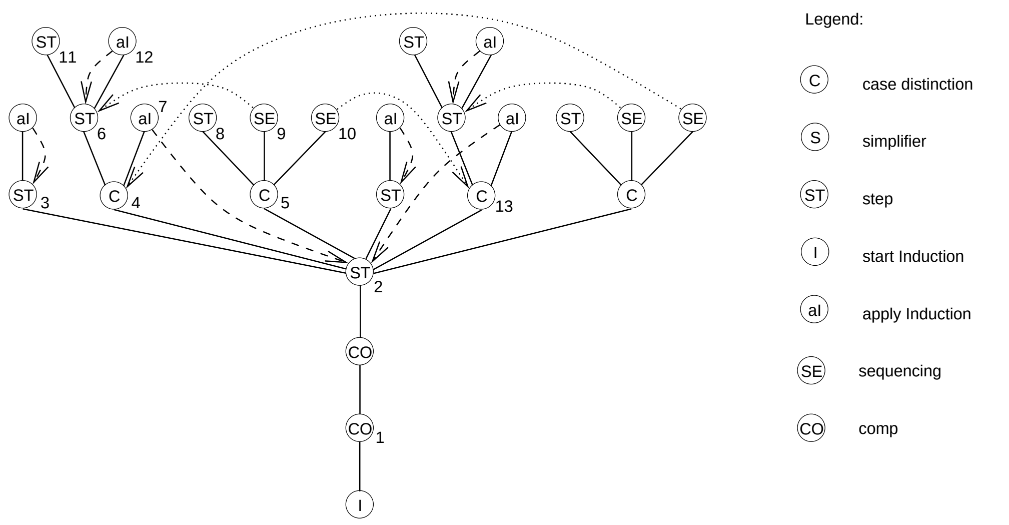


Fig. 1. Proof Graph for Theorem 1

Premise *i* is a temporal logic sequent while premise *ii* - *iv* contain only predicate logic formulas. These four proof obligations have the following informal meaning:

i. All components must sustain their guarantee as long as the assumption holds. These are the only proof obligations which require a temporal logic proof.
ii. The guarantee of each component preserves the global guarantee and does not violate the assumptions of all other components.
iii. The assumptions of all components are transitive. With this property, the components assumption is preserved even if other components make several steps.
iv. All component assumptions hold if the global assumption holds. Therefore, no component assumption is violated in the environment-step.

Proof (Outline).
The theorem was formally proven with the theorem prover KIV by using the ITL calculus described in section 2. As first step the proof for two components was done by symbolic execution of two abstract and interleaved components. The simplified proof graph[4] for this first step is depicted in figure 1. The premises *i*-*iv* of theorem 1 are used as lemmas for this proof. Premises *ii*-*iv* are applied by the KIV simplifier on predicate logic premises, which are all closed automatically by KIV. These simplifier steps are omitted in figure 1 for the sake of brevity.

The proof starts with the sequent $M_1 \parallel M_2 \vdash A(v', v'') \overset{+}{\rhd} G(v, v')$ (node 1). M_1 and M_2 are abstract programs that have arbitrary behavior. At first both

[4] The rules *apply Induction* and *Sequencing* refer both to another node in the proof tree, as explained in section 2. Therefore we depict proofs as graphs. The nodes that are referred by the rules *apply Induction* and *Sequencing* are represented by dashed arrows and pointed arrows respectively.

programs are replaced with their assumption-guarantee formulas of premise i of the theorem via the rule *comp*, so node 2 has the following sequent: $A_1(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G_1(v, v') \parallel A_2(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G_2(v, v') \vdash A(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G(v, v')$ Here, the *step* rule is applied for symbolic execution. In the following, only the nodes 3-5 are described, as the other three premises of node 2 are symmetrical to these nodes.

In node 3 the first parallel component has terminated, so it must be shown that $A_2(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G_2(v, v') \vdash A(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G(v, v')$ holds. This can by done using *step* and *apply induction*.

In node 4 the first component has made a normal step (i.e. it is neither terminated nor blocked). The case distinction discerns if $A_1(v', v'')$ holds (node 7) in this step or not (node 6). Node 6 has the sequent $\neg A_1(v', v'') \parallel A_2(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G_2(v, v') \vdash A(v', v'') \overset{+}{\rightarrow\!\!\triangleright} G(v, v')$. Here, in the next step there are three possibilities:

- The left component makes a step (not depicted in the graph, as it can be closed automatically by the simplifier).
- The right component makes a step and $A_2(v', v'')$ is violated too (node 11). This can be closed automatically by another step.
- The right component makes a step and $A_2(v', v'')$ holds (node 12). This premise can be closed by induction.

Node 7 contains exactly the same sequence as node 2, therefore induction can be applied.

Node 5 treats the case if the left component is blocked. Here, three cases are possible:

- Both assumptions $A_1(v', v'')$ and $A_2(v', v'')$ are violated (node 8). This can be closed automatically via step rule.
- Only the assumption $A_1(v', v'')$ is violated (node 9). This is the same case as in node 6, therefore sequencing can be applied.
- $A_1(v', v'')$ holds and the right component has made a step (node 10). This case is covered in node 13, therefore sequencing can be applied.

This proof can be extended to n components by induction over the number of components. The initial induction case for one component can be shown by another temporal induction (similar to node 3 in the proof above). The inductive step can be proved by using the proof for two components as lemma to reduce n components to $n-1$ components. Then the induction hypotheses can be applied.

Usually the construction of a modularization rule is very difficult because of mutual dependencies. One interesting thing in our framework is that symbolic execution and tool support can not only be used to prove the modularization theorem, it actually helps to find the correct premises for the rule. To do so, the proof is as above, but without using the premises ii-iv as lemmas (as we want to find them at this point). Then we try to close all open premises that contain temporal logic formulas, which results in a similar proof graph as shown in figure 1, but with several additional open premises that contain only predicate logic sequents. So to find the correct premises for the modularization theorem are a minimal set of generic predicate logic formulas from which all open sequents can be shown. By this technique a semantic analysis of the parallel operator is not necessary.

Extended Modularization Rule While this first rule may be useful for very simple systems it must be improved to be usable for more complex cases. First, a variable initialization in the temporal logic proofs (obligations i) is needed. Second, applications of this first rule show, that the guarantees are often redundant. Especially it is often necessary to have an invariance property. This invariant can be used to express the relation between the initial state and all suceeding states. Similar techniques for these additions are used e.g. in [18].

So, using the additional predicates $I(v)$ for the invariant, $Init(v)$ for the initial values of the global system and a family of predicates $Init_i(v)$ for the initial values for every system component leads to an extended version of the compositional rule:

Theorem 2. *If:*

i. *for all* $i = 1, \ldots, n$*:*
$M_i, I(v), Init_i(v) \vdash A_i(v', v'') \overset{+}{\rightarrow} G_i(v, v')$

ii. *for all* $i = 1, \ldots, n$*:*
$G_i(v_1, v_2) \wedge I(v_1) \vdash G(v_1, v_2) \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} A_j(v_1, v_2) \wedge I(v_2)$

iii. *for all* $i = 1, \ldots, n$*:*
$A_i(v_1, v_2) \wedge A_i(v_2, v_3) \wedge I(v_1)) \vdash A_i(v_1, v_3)$

iv. $A(v_1, v_2) \wedge I(v_1) \vdash \bigwedge_{i \in \{1..n\}} A_i(v_1, v_2) \wedge I(v_2)$

v. *for all* $i = 1, \ldots, n$*:*
$A_i(v_1, v_2) \wedge I(v_1) \wedge Init_i(v_1) \vdash Init_i(v_2)$

vi. $Init(v_1) \vdash \bigwedge_{i \in \{1..n\}} Init_i(v_1) \wedge I(v_1)$

then:
$M_1 \parallel \ldots \parallel M_n, Init(v) \vdash A(v', v'') \overset{+}{\rightarrow} G(v, v')$

The informal meaning of the proof obligation of this theorem are as follows:

i. These obligations are mostly the same, except that we can now assume the invariant and the initial condition for the respective component in the antecedent.

ii. - *iv.* These obligations are mostly the same as in the previous rule, except that the predicate I can now be assumed in the antecedent. Also, we have to show in obligations ii. and iv., that the invariant is preserved by the guarantee of each component and the global assumption.

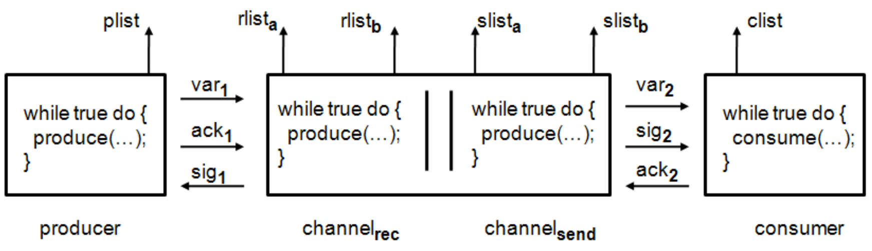

plist
rlist_a
rlist_b
slist_a
slist_b
clist
while true do { produce(...); }
var_1
ack_1
sig_1
while true do { produce(...); }
while true do { produce(...); }
var_2
sig_2
ack_2
while true do { consume(...); }
producer
channel_rec
channel_send
consumer

Fig. 2. Producer-Channel-Consumer (ProChaCon)

v. Here it is shown, that the initial condition of a component is preserved by its assumption.
vi. This obligation establishes the invariant and the initial conditions of the components.

Proof (Sketch). This theorem was also formally proven with KIV. The proof for theorem 2 and 1 are very similar. However, it must be shown for theorem 2 that $Init_1(v)$, $Init_2(v)$ and $I(v)$ holds in the first state. To do that, a case distinction is used before the first step (node 2 in figure 1). The cases where one of the formulas $Init_1(v)$, $Init_2(v)$ and $I(v)$ does not hold can be proved via step and induction, similar to node 8 in figure 1.

4 Case Study

In this section an example for applying the introduced theorem is presented. After an introduction of the producer-channel-consumer case study (short "ProChaCon") and its specification the formulation of the assumption-guarantee (short "AG") properties is described. The section closes with a description of the proofs of some of the proof obligations.

ProChaCon consists, as the name implies, of three interleaved components, depicted in Figure 2. Usually the values of the *producer* component are derived from an application or another component. For our task it is sufficient to generate them randomly. These values are sent using a classical two-way-handshake protocol [19] to the *channel* component. The *channel* is again divided into a receiver and a sender component. Both are connected through a buffer in which the incoming values are stored. The receiver is responsible to store the incoming values into the buffer and the senders job is to forward the buffered values. Thereto, the receiver attaches the incoming value to the buffer-list and the sender transmit the first value of the list as long as the buffer is not empty. The buffered values are transmitted to the *consumer* component, which processes the received values in an arbitrary way. The history of sent and received values is modeled by inserting history lists on certain points, e.g plist, also depicted in Figure 2. They

```
producer:
begin
  while true do
    await ch$_a$.$sig$ =ch$_a$.$ack$;
    a := [?];
    ch$_a$:= mkch(a, ch$_a$.sig, ch$_a$.ack) ;
    ch$_a$:= mkch(ch$_a$.data,
                  $\neg$ ch$_a$.sig, ch$_a$.ack)|
      plist := plist + a
end;
```

```
consumer:
begin
  while true do
    await ch$_b$.$sig$ $\neq$ch$_b$.$ack$;
    b := ch$_b$.$data$;
    ch$_b$ := mkch(ch$_b$.$data$,
                  ch$_b$.$sig$,ch$_b$.$ack$)|
      clist := clist + b
end;
```

```
channel:
 begin
1  while true do
2    await ch$_a$.$sig$ $\neq$ch$_a$.$ack$;
3    c := ch$_a$.$data$;
4    ch$_a$ := $mkch$(ch$_a$.$data$,
                  ch$_a$.$sig$, $\neg$ch$_a$.$ack$)|
       elist$_a$ := elist$_a$ + c ;
5    chbuf := chbuf + c|
       elist$_b$ := elist$_b$ + c
   end;
||
begin
  while true do
    await chbuf$\neq$ [];
    d:= chbuf.$first$|
      chbuf := chbuf.$rest$|
      slist$_a$ := slist$_a$ + chbuf.$first$;
    await ch$_b$.sig = ch$_b$.$ack$;
    ch$_b$ := $mkch$(d, ch$_b$.$sig$,ch$_b$.$ack$);
    ch$_b$ := $mkch$(ch$_b$.$data$,
                  $\neg$ch$_b$.$sig$,ch$_b$.$ack$),
    slist$_b$ := slist$_b$ + d
end;
```

Fig. 3. SPL Representation of ProdChaCon

are implemented as atomic assignments attached to the accordant program step. A specification of the components with SPL is shown below in Figure 4.

First some abbreviatory notations are described that will be used in the following. The sets of all used unprimed, primed and doubleprimed variables are denoted with V, V' and V''. As mentioned in the introduction a step consists of a system step and an environment step. In the following it is often necessary to express that a component only change a set of variables L. This is formulated by a frame assumption, which corresponds to the formula

$$\lceil L \rceil :\Leftrightarrow \bigwedge_{w \in V \setminus L} w' = w$$

which states that all program variables except L are unchanged. Here, L is a subset of V. Further, during the environment step some variables are unchanged. This is formulated with the following predicate

$$Unchanged_{env}(L) :\Leftrightarrow \bigwedge_{w \in L} w' = w''.$$

The verified property is “The list of received values is always a prefix of the list of the sent values”. In other words, only values that have been sent are received and the order is unchanged. So for the overall guarantee the formula

$clist \sqsubseteq plist \rightarrow clist' \sqsubseteq plist'$ is used, where $\sqsubseteq$ is the prefix operator. The global assumption states that all variables are unchanged by the environment. This leads to the following proof obligation for the complete system.

$$Unchanged_{env}(V) \xrightarrow{+} (\text{clist} \sqsubseteq \text{plist} \rightarrow \text{clist}' \sqsubseteq \text{plist}')$$

The system uses, as mentioned, a classical handshake to transmit values. Therefore, the involved components have to guarantee at least that they fulfill their part accurate. Sending components must guarantee that they transmit a value only if it is their turn and that the history-lists are updated in a correct way, formally expressed in $Handshake_{send}$.

$$\begin{aligned} & Handshake_{send}(\text{ch}, \text{hlist}) :\Leftrightarrow \\ & \quad (\text{ch}.sig \neq \text{ch}.ack \rightarrow (\text{ch} = \text{ch}' \wedge \text{hlist} = \text{hlist}')) \\ & \wedge (\text{ch}.sig = \text{ch}.ack \wedge \text{ch}'.sig = \text{ch}'.ack) \rightarrow \text{hlist} = \text{hlist}') \\ & \wedge (\text{ch}.sig = \text{ch}.ack \wedge \text{ch}'.sig \neq \text{ch}'.ack) \rightarrow \text{hlist} + \text{ch}.data = \text{hlist}' \end{aligned}$$

Analogously $Handshake_{receive}$ express that the receiver has to guarantee that values are only received if the handshake variables are unequal. The history list is updated if the handshake variables signalizing that a value was received successfully and the next value can be transmitted.

The producer component has to guarantee two things. First, that only internal variables and the handshake channel are changed. Second, that the handshake protocol is implemented correctly. The producers environment assumption A_1 states that the environment does not change the internal variables as long as the producer could transmit a value. This is captured in G_1 and A_1.

$$\begin{aligned} G_1(V, V') &:\Leftrightarrow \lceil \text{a}, \text{ch}_a, \text{plist} \rceil \wedge Handshake_{send}(\text{ch}_a, \text{plist}) \\ A_1(V', V'') &:\Leftrightarrow Unchanged_{env}(\text{a}, \text{plist}) \wedge (\text{ch}'_a.\text{sig} = \text{ch}'_a.\text{ack} \rightarrow \text{ch}'_a = \text{ch}''_a)) \end{aligned}$$

The AG of the consumer can be formalized analogously:

$$\begin{aligned} G_4(V, V') &:\Leftrightarrow \lceil \text{b}, \text{ch}_b, \text{clist} \rceil \wedge Handshake_{receive}(ch_b, clist) \\ A_4(V', V'') &:\Leftrightarrow Unchanged_{env}(\text{b}, \text{clist}) \wedge (\text{ch}'_b.\text{sig} \neq \text{ch}'_b.\text{ack} \rightarrow \text{ch}'_b = \text{ch}''_b)) \end{aligned}$$

In a similar way the AGs for both channel components (channel$_{rec}$, channel$_{send}$) can be formalized. They need additional guarantees, because they pass the values via a buffer. That this is done correctly is formalized by the two guarantees $Buffer_{in}$ and $Buffer_{out}$.

$$\begin{aligned} & Buffer_{in}(\text{buffer}, \text{hlist}_{in}, \text{value}_{in}) :\Leftrightarrow \\ & \qquad (\text{hlist}_{in} = \text{hlist}'_{in} \wedge \text{buffer} = \text{buffer}') \\ & \quad \vee (\quad \text{hlist}_{in} + \text{value}_{in} = \text{hlist}'_{in} \\ & \qquad \wedge \text{buffer} + \text{value}_{in} = \text{buffer}') \end{aligned}$$

$$\begin{aligned}&\mathit{Buffer}_{out}(\text{buffer}, \text{hlist}_{out}) :\Leftrightarrow\\ &\qquad \text{hlist}_{out} + \text{buffer} = \text{hlist}'_{out} + \text{buffer}'\end{aligned}$$

The complete guarantee for channel$_{rec}$ consists of the statements that channel$_{rec}$ only changes its internal variables, that the receiver part of the handshake protocol is implemented in a correct way and that the component writes into the buffer correctly. Additionally, the component needs to guarantee that the prefix property also holds between both internal history lists. As assumption it can be presumed that the environment does not change the internal variables and the channel is not changed as long as channel$_{rec}$ can receive a value. That leads to the following AG.

$$\begin{aligned}G_2(V, V') \;&:\Leftrightarrow\; \lceil\, \text{c}, \text{ch}_a, \text{chbuf}, \text{rlist}_a, \text{rlist}_b \rceil\\ &\quad\wedge\; \mathit{Handshake}_{receive}(\text{ch}_a, \text{rlist}_a)\\ &\quad\wedge\; \mathit{Buffer}_{in}(\text{chbuf}, \text{rlist}_b, \text{c})\\ &\quad\wedge\; \text{rlist}'_b \sqsubseteq \text{rlist}'_a\\ A_2(V', V'') \;&:\Leftrightarrow\; \mathit{Unchanged}_{env}(\text{c}, \text{rlist}_a, \text{rlist}_b)\\ &\quad\wedge\; (\text{ch}'_a.\mathit{sig} \neq \text{ch}'_a.\mathit{ack} \;\rightarrow\; \text{ch}'_a = \text{ch}''_a))\end{aligned}$$

The AG of the other channel component (channel$_{send}$) can be formalized analogously with *Buffer*$_{out}$ and *Handshake*$_{send}$.

The system always has to be in a correct state. In other words the buffers have to be empty or at least have to be filled in a non-conflicting way. This is expressed as an invariant. Theoretically, it is also possible to put all these into the AGs of the components, but it is more concise to have only local properties there. Therefore statements consisting of variables of more than one component are separated within an invariant, which expresses the connection of the components. First it states that depending on the handshake variables the two neighbor history lists are either equal or they differ in the value that is set in the data field.

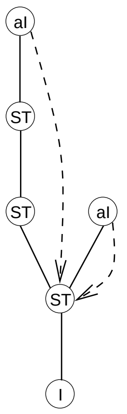
$$\begin{aligned}I_1(V) :\Leftrightarrow\; &(\,\text{ch}_a.\text{sig} = \text{ch}_a.\text{ack} \;\rightarrow\; \text{elist}_a = \text{plist})\\ &\wedge\; (\text{ch}_a.\text{sig} \neq \text{ch}_a.\text{ack} \;\rightarrow\; \text{elist}_a + \text{ch}_a.\text{data} = \text{plist})\\ &\wedge\; (\text{ch}_b.\text{sig} = \text{ch}_b.\text{ack} \;\rightarrow\; \text{clist} = \text{slist}_b)\\ &\wedge\; (\text{ch}_b.\text{sig} \neq \text{ch}_b.\text{ack} \;\rightarrow\; \text{clist} + \text{ch}_b.\text{data} = \text{slist}_b)\end{aligned}$$

For the channel it is stated that all values that were written into the buffer are either still in the buffer or were already send to the consumer component. This is formalised with *slist + chbuf = elist*$_b$. Additionally, some prefix properties are needed to show the overall property:

$$\begin{aligned}I_2(V) :\Leftrightarrow\; &\text{clist} \sqsubseteq \text{slist}_b \wedge \text{slist}_b \sqsubseteq \text{slist}_a \wedge \text{slist}_a \sqsubseteq \text{elist}_b\\ &\wedge\; \text{elist}_a \sqsubseteq \text{elist}_a \wedge \text{elist}_a \sqsubseteq \text{plist}\end{aligned}$$

The overall invariant $I(V)$ is $I_1(V) \wedge I_2(V)$. The only needed initial information is, that the history-lists of both channel components are equal. This is formulated with $\text{init}_2 \equiv (\text{rlist}_a = \text{rlist}_b)$ and $\text{init}_3 \equiv (\text{slist}_a = \text{slist}_b)$.

All proof obligations were formally proven with KIV. To give an impression of the proof effort for the components, we describe as example proof of the temporal logic proof obligation i for channel_{rec}, which is as follows:



$$M_2, I(V), Init_2(V) \vdash A_2(V', V'') \xrightarrow{+} G_2(V, V')$$

The proof graph for this obligation is shown on the right side. In the beginning we start induction, explained in section 2.3. Initially the program is in position 1 (the numbers refer to the program of page 21). The *while*-loop could be evaluated, so that the program is in position 2. Executing the first step leads to a case distinction. Either the *await*-statement could be evaluated to *true* and the program is on position 3 or to *false* and the program remains at position 2. In the second branch induction is applied, as the sequent has not changed. In the first branch further steps are executed till the program is again at position 1, which has been encountered before. In this case induction is applied and the proof is finished. The other three temporal logic proof obligations can be verified analogously without additional effort.

The proofs for the predicate logic proof obligations are straight forward. They start with a case distinction of the conjunctions on the right side of the sequence. All premises can then be closed by the simplifier of KIV automatically.

All in all the reuse of the AGs is very high, for example every component that uses a handshake protocol has to fulfill the handshake guarantees. Only the invariant depends on the property we want to verify. All proofs are simple and can be automated to a large extend. One reason for this is, that the components are no longer interleaved after modularization and so symbolic execution leads to only few new cases.

5 Related Work and Summary

In summary, we have presented a method how to use symbolic execution together with compositional reasoning. As basis for our work we use an ITL variant [10] that supports symbolic execution. Furthermore it provides a compositional interleaving operator, which allows us to formulate an assumption-guarantee theorem and prove it on syntactic level. The logic is fully integrated into the interactive theorem prover KIV and all proofs where done within this tool. A further advantage of our logic is the possibility to directly include multiple system description languages into the logic formalism, e.g. SPL which is used in this work. Other languages that were also successfully integrated into the logic are Statemate and UML statecharts [20, 21] as well as Asbru, a language used for the verification of medical protocols [22]. The tool support and the syntactic nature of the theorem simplifies adaption of the theorem to particularities of these languages (e.g. to have better support for events in statecharts). The ability of symbolic execution of programs and statecharts supports intuitive and understandable proofs. To our knowledge this is the first work combining symbolic execution with compositional reasoning.

Our compositional theorem is inspired by the work of Abadi and Lamport [17]. They introduced the $\overset{+}{-\!\!\rhd}$ operator and a theorem which is suitable for safety and liveness properties. In comparison to our work they use conjunction for the composition of components. While conjunction is a more elementary operator than our interleaved operator, all components must be specified as stutter equivalent components. To achieve this, their components must be specified in a special formula in normal form, while we are able to specify the components directly in various description languages. Due to the inclusion of the double primed variables we have a stuttering mechanism directly in our semantics.

We use a similar technique for defining assumptions and guarantees as Cau and Collette [18]. Their theoretical work is more general as the described theorem can be adapted to state based as well as message based systems. Compared to this our focus was to provide a calculus and tool support for our technique.

Solanki et. al. [23] use compositional reasoning together with ITL. They use an AG variant that allows guarantees to be formulated in ITL. As tool they use (ana)Tempura [14, 11]. This technique is applied to a semantic web service description.

In a paper by Zwiers et. al. [24] invariants and preconditions are integrated in a compositional framework for concurrency. Joseph and Pandya [25] integrate invariants in a framework for total correctness. They use CSP-like distributed programs. Moszkowski [26] uses ITL for a compositional specification and proof technique. Further work about compositionality are e.g. Pnueli [27], Stirling [28] or Woodcock and Dickinson [29].

The producer-channel-consumer case study is a standard example for compositional reasoning. Pnueli [30] described a producer-channel-consumer example already 1986 formally with temporal logic. Abadi and Lamport [17] also used this example to illustrate how to specify components of concurrent systems. In their example they show that two N-element queues can be composed to an (2N+1)-element queue. Jonsson and Tsay [31] use the same example and property. The producer-channel-consumer example is also verified by Breitling et. al. [32], where streams for modelling the communicationare are used and Rock et. al. [33] in combination with TLA for specification.

Next steps are to apply our approach on liveness properties. First experiments in this direction were very promising. Another interesting topic would be to integrate an objectlevel $\overset{+}{-\!\!\rhd}$ operator similar to [17]. This would allow us to use more complex assumption guarantee properties without abandoning the advantages of our approach: symbolic execution and tool support with various system description languages.

References

1. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9) (1965) 569
2. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4) (1983) 596–619
3. Misra, J., Chandi, K.: Proofs of networks of processes. IEEE Transactions of Software Engineering (1981)
4. de Roever, W.P., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press (2001)
5. Furia, C.A.: A compositional world: a survey of recent works on compositionality in formal methods. Technical Report 2005.22, Dipartimento di Elettronica e Informazione, Politecnico di Milano (March 2005)
6. Harel, D.: Dynamic logic. In Gabbay, D., Guenther, F., eds.: Handbook of Philosophical Logic. Volume 2. Reidel (1984) 496–604
7. Heisel, M., Reif, W., Stephan, W.: A Dynamic Logic for Program Verification. In Meyer, A., Taitslin, M., eds.: Logical Foundations of Computer Science. LNCS 363, Berlin, Logic at Botik, Pereslavl-Zalessky, Russia, Springer (1989) 134–145
8. Burstall, R.M.: Program proving as hand simulation with a little induction. Information processing 74 (1974) 309–312
9. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394
10. Balser, M.: Verifying Concurrent Systems with Symbolic Execution. Shaker Verlag, Germany (2006)
11. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
12. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In Hutter, D., Stephan, W., Traverso, P., Ullmann, M., eds.: Proc. Int. Wsh. Applied Formal Methods. Volume 1641 of LNCS., Springer (1999) 330–337
13. Balser, M., Reif, W.: Interactive verification of concurrent systems using symbolic execution. Technical Report 2008-12, Universität Augsburg (2008)
14. Cau, A., Moszkowski, B., Zedan, H.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK. (2002) http://www.cse.dmu.ac.uk/STRL/ITL/.
15. Manna, Z., Pnueli, A.: Temporal verification diagrams. LNCS 789 (1994) 726–765 Springer-Verlag.
16. Balser, M., Reif, W.: An interval temporal logic with compositional interleaving. Technical Report 2008-11, Universität Augsburg (2008)
17. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)
18. Cau, A., Collette, P.: Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. Acta Inf. **33**(2) (1996) 153–176
19. Mead, C., Conway, L.: Introduction to VLSI systems. Addison-Wesley (1980)
20. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of uml state machines. In Davies, J., Schulte, W., Barnett, M., eds.: Proc. 6th Int. Conf. of Formal Engineering Methods. Volume 3308 of LNCS., Springer (2004)
21. Thums, A.: Formale Fehlerbaumanalyse. PhD thesis, Universität Augsburg, Augsburg, Germany (2004) (in German).
22. Schmitt, J., Balser, M., Reif, W.: Asbru in KIV v2.1 – a tutorial. Technical Report 2006-03, University of Augsburg (2006)

23. Solanki, M., Cau, A., Zedan, H.: Augmenting semantic web service descriptions with compositional specification. In Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E., eds.: Proc. of 13th int. conference on World Wide Web, ACM (2004) 544–552
24. Zwiers, J., de Roever, W.P., van Emde Boas, P.: Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In: Proc. of 12th Colloquium on Automata, Languages and Programming, Springer (1985) 509–519
25. Pandya, P.K., Joseph, M.: P-A logic: a compositional proof system for distributed programs. Distributed Computing **5**(1) (1991) 37–54
26. Moszkowski, B.: Compositional reasoning using interval temporal logic and tempura. LNCS 1536 (1996) 439–464 Springer-Verlag.
27. Pnueli, A.: In transition from global to modular temporal reasoning about programs. (1985) 123–144
28. Stirling, C.: A generalization of Owicki-Gries's Hoare logic for a concurrent while language. Theor. Comput. Sci. **58**(1-3) (1988) 347–359
29. Woodcock, J.C.P., Dickinson, B.: Using VDM with rely and guarantee-conditions. Experiences from real projects. In: Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead, New York, NY, USA, Springer-Verlag New York, Inc. (1988) 434–458
30. Pnueli, A.: Applications of temporal logic to the specification and verification of concurrent systems: A survey of current trends. LNCS 224, Berlin, Springer (1986)
31. Jonsson, B., Tsay, Y.K.: Assumption/guarantee specifications in linear-time temporal logic. Theoretical Computer Science, Vol. 167 (1996)
32. Breitling, M., Philipps, J.: Black box views of state machines. Technical Report TUM-I9916, Technische Univerität München (1999)
33. Rock, G., Stephan, W., Wolpers, A.: Modular reasoning about structured tla specifications. In Berghammer, R., Lakhnech, Y., eds.: Tool Support for System Specification, Development and Verification, Springer (1999)