When Do We (Not) Need Complex Assume-Guarantee Rules?

Antti Siirtola
University of Oulu

Stavros Tripakis
University of California, Berkeley
Aalto University

Keijo Heljanko
Helsinki Institute for Information Technology HIIT
Aalto University

***Abstract*—Assume-guarantee (AG) reasoning is a compositional verification method where a verification task involving many processes is broken into multiple verification tasks involving fewer and/or simpler processes. Unfortunately, AG verification rules, and especially circular rules are often complex and hence hard to reason about. This raises the question whether complex rules are really necessary, especially in view of formalisms that already enable compositional reasoning via simple rules based on precongruence. This paper investigates this question for two formalisms: (1) labelled transition systems (LTS) with parallel composition and weak simulation, and (2) interface automata (IA) with composition and alternating simulation $\geqslant_O^I$. In (1), not all AG rules are sound and the precongruence rule cannot replace all sound ones, but we can provide a generic and sound AG rule that complements the precongruence rule. We show that in (2) all AG rules are sound and can be replaced by a simple rule where all premisses are of the form $P_i \geqslant_O^I Q_i$. Moreover, we show that proofs in the LTS AG rule can be converted into proofs in the simple IA rule. This suggests that circular reasoning is a built-in feature of the IA formalism, and provided system components can be modelled as IA, complex assume-guarantee rules are not needed.**

I. Introduction

Suppose that the goal is to do verification of a system composed of two processes, P_1 and P_2. We want to show that the composition of P_1 and P_2 satisfies some property ϕ, denoted $P_1 \parallel P_2 \models \phi$. Because of problems like state-explosion, we may fail to prove directly $P_1 \parallel P_2 \models \phi$. We can then turn to compositional methods based on *assume-guarantee reasoning* (e.g., see [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]), *process calculi* [12], [13] and *interface automata* [14], [15], [16], [17], [18], [19].

In compositional verification, typically we try to find abstract versions Q_1 and Q_2 of P_1 and P_2, respectively. If Q_1 and Q_2 are sufficiently small, we may be able to show $Q_1 \parallel Q_2 \models \phi$. What remains to be shown is that $P_1 \parallel P_2$ refines $Q_1 \parallel Q_2$, written $P_1 \parallel P_2 \preceq Q_1 \parallel Q_2$. Since we want to avoid constructing the whole reachable state space of $P_1 \parallel P_2$, we usually cannot establish the refinement directly. However, if the refinement relation is a precongruence, this can be established by showing that $P_1 \preceq Q_1$ and $P_2 \preceq Q_2$. Assuming that $P_1 \parallel P_2 \preceq Q_1 \parallel Q_2$ and $Q_1 \parallel Q_2 \models \phi$ imply $P_1 \parallel P_2 \models \phi$, as it usually does, we are done. But things may not be so simple, and it may be that $P_1 \not\preceq Q_1$ or $P_2 \not\preceq Q_2$, in which case we cannot directly exploit the simple precongruence rule. In that case, we may need to use a complex rule such as a circular assume-guarantee rule (see, e.g., [4], [6], [11]).

There are various circular assume-guarantee (AG) rules proposed in the literature, but they more or less follow the form:

$$\text{CAG1: } \frac{P_1 \parallel Q_2 \preceq Q_1 \quad P_2 \parallel Q_1 \preceq Q_2}{P_1 \parallel P_2 \preceq Q_1 \parallel Q_2} \, \Gamma(P_1, Q_1, P_2, Q_2)$$

The rule says that if process P_1 (resp., P_2), when run in parallel with the abstract version Q_2 (resp., Q_1) of P_2 (resp., P_1), is a refinement of Q_1 (resp., Q_2) and the side condition $\Gamma(P_1, Q_1, P_2, Q_2)$ holds, then the composition of P_1 and P_2 is a refinement of the composition of Q_1 and Q_2. The rule is circular, because, e.g., Q_1 appears on the both sides of the refinement in the premisses, on the right hand side of the former premise and on the left hand side of the latter premise. Reasoning with circular rules can be complicated because a change in Q_1 may result in a change in Q_2 and so on. Moreover, the most complex part of an AG rule is usually hidden in the side condition, which is nevertheless typically needed for the soundness of the rule.

Compositional frameworks, in particular process calculi and interface theories such as *interface automata* [15] offer mainly the precongruence rule. This means that the refinement is a reflexive, transitive and compositional relation on the set of processes. Here, we concentrate on the compositionality property which is the most interesting part of precongruence from the viewpoint of AG reasoning. The compositionality property, abbreviated SIA in what follows, can be written as follows:

$$\text{SIA: } \frac{P_1 \preceq Q_1 \qquad \cdots \qquad P_n \preceq Q_n}{\parallel_{i=1}^n P_i \preceq \parallel_{i=1}^n Q_i} \, .$$

This rule is *simple* in two ways. First, the premisses are of the simple form $P_i \preceq Q_i$, involving only a process P_i and its abstraction Q_i. Second, the rule has no side condition.

Since SIA already is a powerful tool, this raises the question whether complex compositional verification rules are really necessary. We study this question in this paper, in the context of two formalisms: *labelled transition systems* (LTSs) and interface automata (IA). The main contributions of the paper are the following:

In the case of LTSs, not all assume-guarantee (AG) rules are sound and the precongruence rule cannot replace all sound ones (see Ex. 8), but we can provide a generic and sound AG rule that complements the precongruence rule (Theorem 15).

In the case of IA, we show that all AG rules (given by Definition 2) are sound and that they can be replaced by SIA (Theorem 24). This suggests that circular reasoning is a built-in feature of the IA formalism, in the sense that complex assume-guarantee rules are not needed provided system components can be modelled as IA. For example, most software systems, where for each communication event, there is a single sender but possibly multiple receivers, can be modelled as IA.

Finally, we show that any proof using the sound LTS AG rule of Theorem 15 can be converted into a proof using SIA (Theorem 26), provided a standard assumption on input-determinism holds. The conversion may not give any complexity gains but the result suggests that if we need to do AG reasoning, it makes sense to give the IA formalism a try. That is because everything you can prove by using the LTS AG rule can be proved in the IA world as well and the AG proofs on IA can be made at least conceptually simpler as complex rules are not needed.

In order to formulate and prove our results, we introduce a novel technical tool, a *constriction operator*. This is needed to express the behaviour of a process in the presence of other processes if the composition operator affects the alphabet.

The proofs can be found in the online appendix [20].

II. Assume-Guarantee Rules

We consider a set $\mathcal{P}$ of processes equipped with a commutative and associative (partial) composition operator $\|: \mathcal{P} \times \mathcal{P} \to \mathcal{P}$, a compositional refinement preorder, i.e., a precongruence, $\preceq$ on $\mathcal{P}$ and a compositional equivalence, i.e., a congruence, $\equiv$ on $\mathcal{P}$ such that $\equiv$ is a subset of the kernel $\{(P_1, P_2) \in \mathcal{P} \times \mathcal{P} \mid P_1 \preceq P_2, P_2 \preceq P_1\}$ of $\preceq$. We say that P_1 and P_2 are *composable* when $P_1 \| P_2$ is defined. We assume that (1) composability is preserved under refinement, i.e., if P and P_1 are composable and $P_2 \preceq P_1$, then P and P_2 are composable, too, (2) whenever P_1, P_2, P_3 are pairwise composable processes, then $(P_1 \| P_2)$ and P_3 are composable as well and $(P_1 \| P_2) \| P_3 \equiv P_1 \| (P_2 \| P_3)$ (partial associativity), and (3) there is an *identity process* P_{id} such that $P \| P_{id} \equiv P$ for every process $P \in \mathcal{P}$. This allows us to extend the composition to finite families $\{P_i\}_{i=1}^n$ of pairwise composable processes by defining $\|_{i=1}^n P_i$ as the process $P_1 \| \cdots \| P_n$ when $n \geq 1$, and the identity process P_{id} otherwise.

In the rule CAG1 above, the composition operator is used to express assumptions on the behaviour of the processes in the presence of other processes. However, this works only in simple cases where the composition operator does not introduce any new behaviour. For example, if the composition operator affects the alphabet, e.g., the composition operator of the original IA formalism [14] or CCS (Calculus of Communicating Systems [12]) which synchronises matching input and output events and turns them into the output or invisible internal event, we need a more sophisticated way to express and extract "the behaviour of a process P when run in parallel with a process Q".

To overcome this problem we introduce a (partial) *constriction operator* $| : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$, which is our main technical tool. The notation $P|_Q$ reads the *constriction of P with respect to Q* and intuitively means the behaviour of P when run in parallel with Q.

Definition 1 (Constriction operator)**.** A (partial) function $| : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ is a *constriction operator* if

1) $P_1|_{P_2}$ is defined whenever P_1 and P_2 are composable,
2) $P|_{P_{id}} \equiv P$ for all processes P,
3) if P, P_1 are composable and $P_1 \equiv P_2$, then $P_1|_P \equiv P_2|_P$ and $P|_{P_1} \equiv P|_{P_2}$,
4) if P, Q_1, Q_2 are pairwise composable, then $(P|_{Q_1})|_{Q_2} \equiv P|_{Q_1 \| Q_2}$, and
5) if P_1, P_2 and P_2, Q are composable, then $(P_1|_{P_2}) \| (P_2|_Q) \equiv P_1 \| (P_2|_Q)$.

The second item states that constriction with respect to the identity process has no observable effect. The third item says that $\equiv$ needs to be monotonic with respect to constriction. The following item states that successive constrictions can be combined. The last item captures the fundamental idea of constriction: if P_1 is run in parallel with a constriction of P_2, constricting P_1 further with respect to a process P_2 has no observable effect. A constriction operator should not be confused with a *quotient* operator [18] which, given a global specification and the specification of a subsystem, gives the specification of the subsystem yet to be implemented.

With the aid of a constriction operator, AG rules can be expressed in the following form:

Definition 2 (AG rule)**.** An *assume-guarantee (AG) rule* is an inference rule of the form

$$\frac{P_1|_{C_1} \preceq Q_1 \quad \cdots \quad P_n|_{C_n} \preceq Q_n}{\|_{i=1}^n P_i \preceq \|_{i=1}^n Q_i} \, \Gamma(P_1, Q_1, \ldots, P_n, Q_n) \, ,$$

where

1) $\{P_i\}_{i=1}^n$ and $\{Q_i\}_{i=1}^n$ are two non-empty sets of pairwise composable processes
2) $\Gamma : \mathcal{P}^{2n} \to \{\mathit{false}, \mathit{true}\}$ is a side condition and
3) $C_i := Q_{i,1} \| \cdots \| Q_{i,k_i}$ for all $i \in \{1, \ldots, n\}$, where $k_i \geq 0$ is an integer and $\{Q_{i,1}, \ldots, Q_{i,k_i}\} \subseteq \{Q_1, \ldots, Q_n\} \setminus \{Q_i\}$.

For example, rule CAG1 presented earlier is a special case of the generic form above, where the constriction $P_1|_{Q_2}$ is defined simply as the parallel composition $P_1 \| Q_2$. (Provided it is idempotent, $\|$ satisfies the five requirements for $|$.) A precongruence-based rule such as SIA can also be put in the form of Definition 2, by defining $P|_Q := P$. Finally, an acyclic rule such as [10]

$$\text{NAG1:} \ \frac{P_1 \preceq Q_1 \qquad Q_1 \| P_2 \preceq Q_2}{P_1 \| P_2 \preceq Q_2}$$

can be seen as a special case of the precongruence rule, as its end result $P_1 \| P_2 \preceq Q_2$ follows from $P_1 \| P_2 \preceq Q_1 \| P_2$ (compositionality of $\preceq$ applied to the first premise $P_1 \preceq Q_1$), the second premise $Q_1 \| P_2 \preceq Q_2$, and transitivity of $\preceq$.

An AG rule is *non-circular* if $k_1 = \ldots = k_n = 0$, i.e., $C_i = P_{id}$ for all $i \in \{1, \ldots, n\}$, otherwise the rule is *circular*.

An AG rule is *simple* if it is non-circular and has no side condition, i.e., the side condition is always true, otherwise the rule is *complex*.

We say that an AG rule is *sound* if whenever the premisses and the side condition are true, the conclusion holds as well. An AG rule *R1* is *at least as general as* an AG rule *R2*, if (i) whenever the premisses and the side condition of *R2* are true, then the premisses and the side condition of *R1* hold as well, and (ii) the conclusion of *R1* implies the conclusion of *R2*.

Example 3. As a running example, we consider a system in Figure 1, taken from [5], where two components, Gen and Add communicate through binary channels x, y and z. Add is an adder; it receives bits x and y and outputs $z = x + y$, where addition is modulo 2. Gen is a generator; if it receives 0 from the channel z, it outputs 1 on both the channels x and y, but if it receives 1 then the output is arbitrary. Obviously, there is circular dependency between Gen and Add and if we make no assumptions on the initial values of x and y, nothing can be said about the value of z and vice versa. However, if Gen starts by outputting 1 to both the channels x and y, then it is intuitively obvious that only 0 is seen in the channel z and 1 in the channels x and y. On the other hand, intuition often results in incomplete conclusions and the overall state space may be too large for exhaustive exploration. That is why we would like to use formal assume-guarantee reasoning to prove that no other values are seen on the channels.

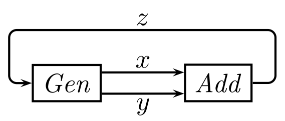


Fig. 1. System of two components connected to each other (example taken from [5]).

III. Labelled Transition Systems

In this section, we recall basic definitions for *labelled transition systems* (LTSs). We consider LTSs with a parallel composition operator and weak simulation as a refinement preorder. A labelled transition system [13] is basically a directed graph whose vertices are called states, one of the states is marked as initial, and edges are called transitions and they are labelled by events. To put it more formally, we assume that there is a countably infinite set of *events* among which there is a single *invisible* event, denoted τ, representing the internal actions of processes. The other events are called *visible* and they are used for communication between processes and their environment.

Definition 4 (LTS)**.** A *labelled transition system (LTS)* is a quadruple $L := (S, E, \longrightarrow, \hat{s})$, where (1) S is a finite non-empty set of *states*, (2) E is a finite set of visible events, (3) $\longrightarrow \subseteq S \times (E \cup \{\tau\}) \times S$ is a set of *transitions*, and (4) $\hat{s} \in S$ is the *initial* state.

The set S of the states of L is denoted by $\mathrm{st}(L)$. The set E is the *alphabet* of L, denoted $\Sigma(L)$, and $\Sigma_\tau(L)$ denotes $\Sigma(L) \cup \{\tau\}$. The set of all LTSs is denoted by $\mathcal{L}$.

We write $s \xrightarrow{\alpha} s'$ short for $(s, \alpha, s') \in \longrightarrow$. For every state s and every event α, we write $s \xrightarrow{\alpha}$, if there is a state s' such that $s \xrightarrow{\alpha} s'$, and $s \not\xrightarrow{\alpha}$ if there is no such state. Notation $s \xrightarrow{\alpha_1 \cdots \alpha_n} s'$ means that there are states $s_0, \ldots, s_n$ such that $s_0 = s$, $s_n = s'$ and $s_{i-1} \xrightarrow{\alpha_i} s_i$ for all $i \in \{1, \ldots, n\}$. Note that n can be zero as well. Given a set A of visible events, we say that an LTS L is *A-enabled* if $s \xrightarrow{\alpha}$ for every state s and every event $\alpha \in A$.

We write $s \xRightarrow{\tau} s'$ short for $s \xrightarrow{\tau \cdots \tau} s'$. Notation $s \xRightarrow{\alpha} s'$, where α is a visible event, means that there is a state s'' such that $s \xRightarrow{\tau} s''$ and $s'' \xrightarrow{\alpha} s'$. For every state s and every event α, we write $s \xRightarrow{\alpha}$, if there is a state s' such that $s \xRightarrow{\alpha} s'$, and $s \not\xRightarrow{\alpha}$ if there is no such state. L is *A-deterministic* if for every state s and every event $\alpha \in A$, $s \xRightarrow{\alpha} s'_1$ and $s \xRightarrow{\alpha} s'_2$ implies $s'_1 = s'_2$. For a set S of states and a set A of events, we write $\mathrm{cl}_A(S)$ for the set $\{s' \mid \exists s \in S : \exists \alpha_1, \ldots, \alpha_n \in A : s \xrightarrow{\alpha_1 \cdots \alpha_n} s'\}$ of all states reachable from any $s \in S$ via events in A. The *reachable* states of L are the states in $\mathrm{cl}_{\Sigma_\tau(L)}(\{\hat{s}\})$.

Modelling system components as LTSs allows us to perform refinement checks between them. We use weak simulation as a notion of refinement, which preserves safety properties. Intuitively, a specification LTS weakly simulates an implementation LTS if every transition of the implementation LTS can be matched by the specification modulo the invisible events.

Definition 5 (Weak simulation)**.** Let L_1 and L_2 be LTSs over the same alphabet, i.e., $L_i = (S_i, E, \longrightarrow_i, \hat{s}_i)$ for both $i \in \{1, 2\}$. A relation $R \subseteq S_1 \times S_2$ is a *weak simulation (from L_1 to L_2)* if $(s_1, s_2) \in R$ and $s_1 \xrightarrow{\alpha}_1 s'_1$ implies that there is s'_2 such that $s_2 \xRightarrow{\alpha}_2 s'_2$ and $(s'_1, s'_2) \in R$.

L_2 weakly simulates L_1, denoted $L_1 \preceq_{\mathrm{sim}} L_2$, if there is a weak simulation R from L_1 to L_2 such that $(\hat{s}_1, \hat{s}_2) \in R$. We write $L_1 \equiv_{\mathrm{sim}} L_2$ if there is a weak simulation R from L_1 to L_2 such that $\{(s, t) \mid (t, s) \in R\}$ is a weak simulation from L_2 to L_1 and $(\hat{s}_1, \hat{s}_2) \in R$, i.e., R is a *weak bisimulation* between L_1 and L_2. Obviously, a weak simulation is a preorder and a weak bisimulation an equivalence on the set of LTSs.

Often, a system implementation involves several concurrent components. That is why an LTS representing the system implementation is typically constructed from smaller LTSs by using parallel composition.

Definition 6 (Parallel composition on LTSs)**.** Let L_i be the LTS $(S_i, E_i, \longrightarrow_i, \hat{s}_i)$ for both $i \in \{1, 2\}$. The *parallel composition of L_1 and L_2* is a quadruple

$$L_1 \parallel L_2 := (S_1 \times S_2, E_1 \cup E_2, \longrightarrow, (\hat{s}_1, \hat{s}_2)) ,$$

where $\longrightarrow$ is the set of all triples $((s_1, s_2), \alpha, (s'_1, s'_2))$ such that

1) either $\alpha \neq \tau$ and $s_i \xrightarrow{\alpha}_i s'_i$ for both $i \in \{1, 2\}$;
2) or $\alpha \in \Sigma_\tau(L_1) \setminus \Sigma(L_2)$, $s_1 \xrightarrow{\alpha}_1 s'_1$, $s_2 \in S_2$ and $s_2 = s'_2$;
3) or $\alpha \in \Sigma_\tau(L_2) \setminus \Sigma(L_1)$, $s_2 \xrightarrow{\alpha}_2 s'_2$, $s_1 \in S_1$ and $s_1 = s'_1$.

Obviously, the parallel composition of LTSs results in an LTS and the operator is commutative and associative. A single state LTS $L_{id} := (\{\hat{s}\}, \emptyset, \emptyset, \hat{s})$ with the empty alphabet and no transition is an identity element of $\|$. Moreover, weak simulation is preserved under parallel composition which implies that weak simulation is a precongruence on the set of LTSs. Similarly, weak bisimulation can be shown to be a congruence on the set of LTSs.

Proposition 7. *Let L_1, L_2, L_3 be LTSs. Then the following holds.*

1) $L_1 \| L_2 \equiv_{\text{sim}} L_2 \| L_1$ *(commutativity).*
2) $L_1 \| (L_2 \| L_3) \equiv_{\text{sim}} (L_1 \| L_2) \| L_3$ *(associativity).*
3) $L_1 \| L_{id} \equiv L_1$ *(identity).*
4) *If* $L_1 \preceq_{\text{sim}} L_2$*, then* $L_1 \| L_3 \preceq_{\text{sim}} L_2 \| L_3$*. If* $L_1 \equiv_{\text{sim}} L_2$*, then* $L_1 \| L_3 \equiv_{\text{sim}} L_2 \| L_3$ *(compositionality).*

IV. Sound AG Rule for LTSs

In the calculus of LTSs, AG rules are not sound, in general. In this section, we show that in addition to simple (thus non-circular) rules, which are sound by the precongruence of weak simulation, there is another class of sound AG rules (Theorem 15). Similar results are also reported earlier [21], but Theorem 15 is generic in the sense that it holds for any constriction operator instead of a specific parallel composition operator.

Example 8. To see the need for sound LTS AG rules, let us consider our example system. The generator *Gen* and the adder *Add* are naturally modelled as LTSs L_G and L_A shown in Figure 2, where an event $ab \in \{00, 01, 10, 11\}$ denotes that a and b are received from/sent to the channels x and y, respectively, and an event $c \in \{0, 1\}$ means that c is received from/sent to the channel z.

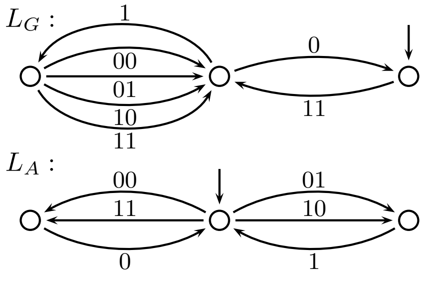
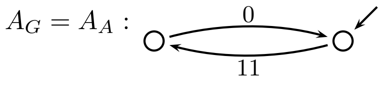


Fig. 2. LTSs L_G and L_A representing the behaviour of the generator *Gen* and the adder *Add* as individual components.

Based on our understanding about the system, it should be possible to abstract L_G and L_A as LTSs A_G and A_A in Figure 3, which actually are the same LTS and clearly formalise the fact that only 0 is seen in the channel z and 1 in the channels x and y. However, since $L_G \not\preceq_{\text{sim}} A_G$ and $L_A \not\preceq_{\text{sim}} A_A$, we cannot deduce that the parallel composition of L_G and L_A satisfies $A_G \| A_A = A_G$, i.e., $L_G \| L_A \preceq_{\text{sim}} A_G \| A_A$, by only using the precongruence rule. Hence, we need a circular AG rule of the form

$$\frac{L_1|_{A_2} \preceq_{\text{sim}} A_1 \qquad L_2|_{A_1} \preceq_{\text{sim}} A_2}{L_1 \| L_2 \preceq_{\text{sim}} A_1 \| A_2} \, \Gamma(L_1, A_1, L_2, A_2) \, .$$

Unfortunately, such rules are not sound in general. For example, if the side condition is always true and L_1 and L_2 are LTSs that can perform an event a repeatedly and A_1 and A_2 are LTSs that block a, then the premisses of the AG rule hold but the conclusion is false. □



$$\Sigma(A_G) = \Sigma(A_A) = \{0, 1, 00, 01, 10, 11\}$$

Fig. 3. LTSs A_G and A_A representing the expected behaviour of the generator *Gen* and the adder *Add* within the system.

In order to present a sound LTS AG rule, we need to define a constriction operator on LTSs first. A trivial choice for an LTS constriction operator $|^t : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ is to define $L_1|^t_{L_2} := L_1$. To see that there is a non-trivial LTS constriction operator, we define the natural constriction of L_1 w.r.t. L_2 that only captures the behaviours of L_1 that are possible in the parallel composition of L_1 and L_2. Formally, the natural constriction is defined as the product of L_1 and the determinised version of L_2 where only the events of L_1 are explicitly recorded.

Definition 9 (Natural constriction on LTSs)**.** Let $L_i := (S_i, E_i, \longrightarrow_i, \hat{s}_i)$ be an LTS for both $i \in \{1, 2\}$. The *natural constriction* of L_1 with respect to L_2 is a quadruple

$$(L_1 \upharpoonright_{L_2}) := (S_1 \times \mathbb{P}(S_2), E_1, \longrightarrow, (\hat{s}_1, \text{cl}_{\Sigma_\tau(L_2) \setminus \Sigma(L_1)}(\{\hat{s}_2\}))),$$

where $\longrightarrow$ is the set of all triples $((s_1, Q_2), \alpha, (s_1', Q_2'))$ such that $Q_2, Q_2' \neq \emptyset$ and

1) $\alpha \in \Sigma_\tau(L_1) \setminus \Sigma(L_2)$, $s_1 \xrightarrow{\alpha}_1 s_1'$ and $Q_2 = Q_2' \subseteq S_2$, or
2) $\alpha \in \Sigma(L_1) \cap \Sigma(L_2)$, $s_1 \xrightarrow{\alpha}_1 s_1'$ and $Q_2' = \text{cl}_{\Sigma_\tau(L_2) \setminus \Sigma(L_1)}(\{s_2' \in S_2 \mid \exists s_2 \in Q_2 : s_2 \xrightarrow{\alpha}_1 s_2'\})$.

It is straightforward to check that the natural constriction of L_1 w.r.t. L_2 is an LTS. Moreover, $\upharpoonright$ satisfies our requirements of the constriction operator.

Proposition 10. *Let L, L_1, L_2, A_1, A_2 be LTSs. Then the following holds.*

1) $L_1 \upharpoonright_{L_2}$ *is defined.*
2) $L \upharpoonright_{L_{id}} \equiv_{\text{sim}} L$.
3) *If* $L_1 \equiv_{\text{sim}} L_2$*, then* $L_1 \upharpoonright_L \equiv_{\text{sim}} L_2 \upharpoonright_L$ *and* $L \upharpoonright_{L_1} \equiv_{\text{sim}} L \upharpoonright_{L_2}$.
4) $(L \upharpoonright_{A_1}) \upharpoonright_{A_2} \equiv_{\text{sim}} L \upharpoonright_{A_1 \| A_2}$.
5) $(L_1 \upharpoonright_{L_2}) \| (L_2 \upharpoonright_{A_1}) \equiv_{\text{sim}} L_1 \| (L_2 \upharpoonright_{A_1})$.

Example 11. Regarding our example, the reachable part of the natural constriction of L_G w.r.t. A_A is A_G and the reachable part of the natural constriction of L_A w.r.t. A_G is A_A, i.e., $L_G \upharpoonright_{A_A} = A_G$ and $L_A \upharpoonright_{A_G} = A_A$ (modulo the states not reachable). Hence, $\upharpoonright$ is a non-trivial constriction operator. □

Having defined a constriction operator, we can present a generic and sound AG rule for LTSs. The only restriction is the side condition, which requires that each event is blocked by at most one LTS representing an abstract process. This corresponds to a realistic situation where for each communication event, there is a single sender but possibly multiple receivers. Most software systems satisfy this requirement.

Theorem 12 (Sound LTS AG rule for the natural constriction)**.** *Let n be an integer such that $n \geq 2$, $L_1, \ldots, L_n, A_1, \ldots, A_n$*

LTSs, and for every $i \in \{1,\dots,n\}$, let $C_i := A_{i,1} \parallel \cdots \parallel A_{i,k_i}$, where $\{A_{i,1},\dots,A_{i,k_i}\}$ is a possibly empty subset of $\{A_1,\dots,A_n\} \setminus \{A_i\}$. Then

$$\frac{L_1{\upharpoonright}_{C_1} \preceq_{\text{sim}} A_1 \quad \cdots \quad L_n{\upharpoonright}_{C_n} \preceq_{\text{sim}} A_n}{\parallel_{i=1}^n L_i \preceq_{\text{sim}} \parallel_{i=1}^n A_i} \,\Gamma(A_1,\dots,A_n)$$

is a sound AG rule, where the side condition $\Gamma(A_1,\dots,A_n)$ is as follows:

> *for each $\alpha \in \bigcup_{i=1}^n \Sigma(A_i)$ there is $j_\alpha \in \{1,\dots,n\}$ such that $\alpha \in \Sigma(A_{j_\alpha})$ and for all $i \in \{1,\dots,n\} \setminus \{j_\alpha\}$, either $\alpha \notin \Sigma(A_i)$ or A_i is $\{\alpha\}$-enabled.*

The enabledness condition cannot be removed in general. To see this, let L_1 and L_2 be the single state LTS $Chaos_\alpha := (\hat{s}, \{\alpha\}, \{(\hat{s},\alpha,\hat{s})\}, \hat{s})$ which can perform α repeatedly and let A_1 and A_2 be the single state LTS $Stop_\alpha := (\hat{s}, \{\alpha\}, \emptyset, \hat{s})$ which can only block α. Now the premisses $L_1 \upharpoonright A_2 \preceq_{\text{sim}} A_1$ and $L_2 \upharpoonright A_1 \preceq_{\text{sim}} A_2$ of the rule hold but the conclusion $L_1 \parallel L_2 \preceq_{\text{sim}} A_1 \parallel A_2$ is false. That is because the side condition does not hold, either A_1 or A_2 should be $\{\alpha\}$-enabled.

Example 13. Let us apply Theorem 12 to our example system. Unfortunately, even though L_G, L_A, A_G, A_A satisfy the premisses of the AG rule, A_G and A_A do not meet the enabledness requirement of the side condition. However, if we use abstractions A'_G and A'_A in Figure 4 instead, then we can treat our system with the AG rule. Obviously, A'_G is $\{0,1\}$-enabled whereas A'_A is $\{00,01,10,11\}$-enabled. Hence, the side condition of the theorem holds. It is also easy to check that $L_G{\upharpoonright}_{A'_A} = A_G \preceq_{\text{sim}} A'_G$ and $L_A{\upharpoonright}_{A'_G} = A_A \preceq_{\text{sim}} A'_A$ meaning that the premisses are true as well. By the AG rule, it implies that $L_G \parallel L_A \preceq_{\text{sim}} A'_G \parallel A'_A$. When we compute the parallel composition $A'_G \parallel A'_A$, we see that $A'_G \parallel A'_A = A_G$. In other words, only 0 is seen in the channel z and 1 in the channels x and y. □

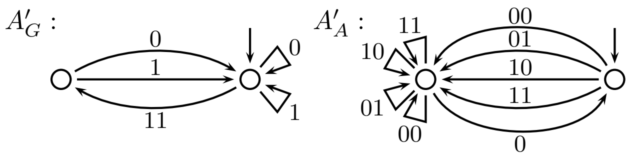


Fig. 4. LTSs A'_G and A'_A representing the behaviour of the abstract generator *Gen* and the abstract adder *Add*.

A drawback of Theorem 12 is that it involves a natural constriction, which is possibly expensive to compute. Instead, we might want to use a constriction operator which is cheaper to compute such as $|^h : (L_1, L_2) \mapsto L_3$, where L_3 is obtained from L_1 by removing states and transitions that are not reachable in $L_1 \parallel L_2$. Note that L_3 is (structurally) at most as big as L_1 whereas the natural constriction of L_1 can be exponentially bigger than L_1. On the other hand, $|^h$ produces an over-approximation of the natural constriction, which holds for all other LTS constriction operators, too.

Lemma 14. *Let $| : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ be a constriction operator. Then $L_1{\upharpoonright}_{L_2} \preceq_{\text{sim}} L_1|_{L_2}$.*

The lemma allows us to generalise Theorem 12 to all LTS constriction operators, including $|^h$. This enables us to overcome a drawback of Theorem 12: the possibly expensive computation of the natural constriction.

Theorem 15 (Sound LTS AG rule)**.** *Let $n \geq 2$ be an integer, $L_1,\dots,L_n, A_1,\dots,A_n$ LTSs, and for every $i \in \{1,\dots,n\}$, let $C_i := A_{i,1} \parallel \cdots \parallel A_{i,k_i}$, where $\{A_{i,1},\dots,A_{i,k_i}\}$ is a possibly empty subset of $\{A_1,\dots,A_n\} \setminus \{A_i\}$. Then*

$$\frac{L_1|_{C_1} \preceq_{\text{sim}} A_1 \quad \cdots \quad L_n|_{C_n} \preceq_{\text{sim}} A_n}{\parallel_{i=1}^n L_i \preceq_{\text{sim}} \parallel_{i=1}^n A_i} \,\Gamma(A_1,\dots,A_n)$$

is a sound AG rule, where the side condition $\Gamma(A_1,\dots,A_n)$ is as follows:

> *for each $\alpha \in \bigcup_{i=1}^n \Sigma(A_i)$ there is $j_\alpha \in \{1,\dots,n\}$ such that $\alpha \in \Sigma(A_{j_\alpha})$ and for all $i \in \{1,\dots,n\} \setminus \{j_\alpha\}$, either $\alpha \notin \Sigma(A_i)$ or A_i is $\{\alpha\}$-enabled.*

Even though Theorems 12 and 15 allow us to verify our example system in a compositional way, we are not quite happy with them. First, the theorems involve a side condition that can be restrictive in some cases. Second, the abstractions A'_G and A'_A we had to use are not natural and not much smaller than the original components L_G and L_A. Instead, we would like to use smaller and natural abstractions A_G and A_A in Figure 3. Next, we will show that this is indeed possible if we switch to the IA framework.

V. Interface Automata

In this section, we recall the basic framework of interface automata with a composition operator and a refinement preorder, the alternating simulation. An interface automaton (IA) is basically an LTS where the alphabet is split into inputs and outputs [15]. But contrary to LTSs, where parallel composition is a total function, in IA it is partial. Moreover, if, during composition, one IA offers an output which cannot be accepted by another IA, the corresponding global state is considered incompatible, and must be unreachable. Like in [22], we use explicit error states to model such incompatibilities.

Definition 16 (IA)**.** An *interface automaton (IA)* is a sextuple $P := (S, I, O, \longrightarrow, F, \hat{s})$, where

1) $(S, I \cup O, \longrightarrow, \hat{s})$ is an LTS such that I and O are disjoint sets of visible events and $\longrightarrow$ is input-deterministic, i.e., $s \xrightarrow{\alpha} s_1, s \xrightarrow{\alpha} s_2, \alpha \in I$ implies $s_1 = s_2$ (this is needed for compositionality, Proposition 20), and
2) $F \subseteq S$ is a set of *error states* such that $(s,\alpha,s'), \alpha \in O \cup \{\tau\}, s' \in F$ implies $s \in F$ (F is downward-closed with respect to $O \cup \{\tau\}$, the output and invisible events cannot be controlled by an environment).

The sets I and O, called the *input* and *output alphabet* of P, respectively, are denoted by $\Sigma_I(P)$ and $\Sigma_O(P)$. P is called *closed* if I is the empty set. We say that P is *compatible* if its initial state is not an error state. The set of all IA is denoted by $\mathcal{I}$.

Given an LTS $L := (S, E, \longrightarrow, \hat{s})$ and a set I of visible events such that $s \xrightarrow{\alpha} s_1, s \xrightarrow{\alpha} s_2, \alpha \in I$ implies $s_1 = s_2$ (input-determinism), we convert L to an IA

$$IA(L, I) := (S, I \cap E, E \setminus I, \longrightarrow, \emptyset, \hat{s})$$

by splitting the alphabet into inputs $E \cap I$ and outputs $E \setminus I$. Respectively, given a compatible IA $P := (S, I, O, \longrightarrow, F, \hat{s})$, we convert it to an LTS

$$LTS(P) := (S \setminus F, I \cup O, \longrightarrow', \hat{s}) \ ,$$

where

$$\longrightarrow' = \longrightarrow \cap (S \setminus F) \times (I \cup O \cup \{\tau\}) \times (S \setminus F) \ ,$$

by merging the input and output alphabet and by removing all error states and incident transitions. Since an IA can be seen as an LTS, we use the same notation for IA as with LTSs.

Modelling system components as IA allows us to perform refinement and compatibility checks between them. Intuitively, an implementation IA refines (or implements) a specification IA if the implementation IA does not have more error states, neither stronger input assumptions, nor weaker output guarantees than the specification IA. This notion of refinement is formalised as alternating simulation [15], [17].

Definition 17 (Alternating simulation)**.** Let P_1 and P_2 be IA over the same input and output alphabet, i.e., $P_i = (S_i, I, O, \longrightarrow_i, F_i, \hat{s}_i)$ for both $i \in \{1, 2\}$. A relation $R \subseteq S_1 \times S_2$ is an *alternating simulation (from P_1 to P_2)* if

1) $(s_1, s_2) \in R, s_1 \in F_1$ implies $s_2 \in F_2$, and

for all $(s_1, s_2) \in R$ such that $s_1 \notin F_1, s_2 \notin F_2$ the following holds:

2) if $s_2 \xrightarrow{\alpha}_2 s_2'$ and $\alpha \in I$, then there is s_1' such that $s_1 \xrightarrow{\alpha}_1 s_1'$ and $(s_1', s_2') \in R$;
3) if $s_1 \xrightarrow{\alpha}_1 s_1'$ and $\alpha \in O \cup \{\tau\}$, then there is s_2' such that $s_2 \overset{\alpha}{\Longrightarrow}_2 s_2'$ and $(s_1', s_2') \in R$.

P_1 *implements* or *refines* P_2, denoted $P_1 \gtrless^I_O P_2$, if there is an alternating simulation R from P_1 to P_2 such that $(\hat{s}_1, \hat{s}_2) \in R$. We write $P_1 \equiv^I_O P_2$ if there is an alternating simulation R from P_1 to P_2 such that $\{(s, t) \mid (t, s) \in R\}$ is an alternating simulation R from P_2 to P_1 and $(\hat{s}_1, \hat{s}_2) \in R$, i.e., R is an *output-weak bisimulation* between P_1 and P_2. Obviously, alternating simulation is a preorder and output-weak bisimulation an equivalence on the set of IA. Moreover, if IA are closed and compatible, then alternating simulation reduces to weak simulation.

Proposition 18. *Let P_1 and P_2 be closed IA such that $P_1 \gtrless^I_O P_2$. If P_2 is compatible, then $LTS(P_1) \preceq_{\mathrm{sim}} LTS(P_2)$.*

We say that P_1 and P_2 are *composable* if they have no common output, i.e., $\Sigma_O(P_1) \cap \Sigma_O(P_2) = \emptyset$. When the composable IA are composed, common inputs are synchronised, the matching inputs and outputs are synchronised and turned into outputs, and the invisible event and the events only in the alphabet of one IA are executed individually in an interleaving fashion. The error states of the IA-composition are those where: (a) either one of the two IA is in an error state, (b) or one IA has an enabled output event which is a disabled input of the other, (c) or such state can be reached via non-inputs.

Definition 19 (Composition on IA)**.** Let $P_i := (S_i, I_i, O_i, \longrightarrow_i, F_i, \hat{s}_i)$ be an IA for $i \in \{1, 2\}$ such that P_1 and P_2 are composable. The *composition of P_1 and P_2* is a sextuple

$$(P_1 \parallel P_2) := (S_1 \times S_2, I, O, \longrightarrow, F, (\hat{s}_1, \hat{s}_2)) \ ,$$

where $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$, $O = O_1 \cup O_2$, $\longrightarrow$ is the set of all triples $((s_1, s_2), \alpha, (s_1', s_2'))$ such that

1) there are $i, j \in \{1, 2\}$ such that $\alpha \notin I_j \cup O_j$, $s_i \xrightarrow{\alpha}_i s_i'$, $s_j \in S_j$ and $s_j = s_j'$, or
2) $\alpha \in (I_1 \cup O_1) \cap (I_2 \cup O_2)$ such that $s_i \xrightarrow{\alpha}_i s_i'$ for both $i \in \{1, 2\}$,

and F is the smallest subset of $S_1 \times S_2$ satisfying the following:

1) $(F_1 \times S_2) \cup (S_1 \times F_2) \subseteq F$;
2) if $i, j \in \{1, 2\}$ and $\alpha \in O_i \cap I_j$ such that $s_i \xrightarrow{\alpha}_i$ and $s_j \not\xrightarrow{\alpha}_j$, then $(s_1, s_2) \in F$;
3) if $((s_1, s_2), \alpha, (s_1', s_2')) \in \longrightarrow$, where $\alpha \in O_1 \cup O_2 \cup \{\tau\}$ and $(s_1', s_2') \in F$, then $(s_1, s_2) \in F$.

Obviously, the composition of composable IA results in an IA and the operator is commutative. Moreover, a single state compatible IA $P_{id} := (\{\hat{s}\}, \emptyset, \emptyset, \emptyset, \emptyset, \hat{s})$ with the empty alphabet and no transition is an identity element of the composition. The composition operator is associative on the set of pairwise composable IA, too. Finally, alternating simulation and output-weak bisimulation are *compositional* with respect to composition. Hence, alternating simulation is a precongruence and output-weak bisimulation a congruence on the set of IA.

Proposition 20. *The following holds for all pairwise composable IA P_1, P_2, P_3.*

1) *$P_1 \parallel P_2 \equiv^I_O P_2 \parallel P_1$ (commutativity).*
2) *$P_1 \parallel (P_2 \parallel P_3) \equiv^I_O (P_1 \parallel P_2) \parallel P_3$ (associativity).*
3) *If $P_1 \gtrless^I_O P_2$, then $P_1 \parallel P_3 \gtrless^I_O P_2 \parallel P_3$. If $P_1 \equiv^I_O P_2$, then $P_1 \parallel P_3 \equiv^I_O P_2 \parallel P_3$. (compositionality).*

VI. AG Rules for IA = Sound AG Rules for IA = Simple AG Rules for IA

In this section, we present the first main contribution of this paper, namely, that all AG rules for IA are sound and can be replaced by a simple rule.

We begin by introducing a natural IA constriction operator. Intuitively, $P_1\restriction_{P_2}$ removes from P_1 all its input transitions which are irrelevant when P_1 is composed with P_2, because P_2 does not offer them as outputs. Formally, the construction of the natural IA constriction is analogous to the construction of the natural LTS constriction except that we also preserve those output transitions of P_1 that are unreachable in $P_1 \parallel P_2$ but may generate error states. This is needed for Item 6 of Lemma 22, i.e., $P_1 \gtrless^I_O P_1\restriction_{P_2}$, which is the key result behind the main theorem of the section.

Definition 21 (Natural constriction on IA)**.** Let $P_i := (S_i, I_i, O_i, \longrightarrow_i, F_i, \hat{s}_i)$ be an IA for both $i \in \{1, 2\}$ such that P_1 and P_2 are composable. The *constriction* of P_1 with respect to P_2 is a sextuple

$$(P_1\restriction_{P_2}) := (S_1 \times \mathbb{P}(S_2), I_1, O_1, \longrightarrow, F_1 \times \mathbb{P}(S_2), (\hat{s}_1, \hat{Q}_2)) \ ,$$

where $\hat{Q}_2 = \mathrm{cl}_{\Sigma_\tau(P_2)\setminus\Sigma(P_1)}(\{\hat{s}_2\})$ and $\longrightarrow$ is the set of all triples $((s_1, Q_2), \alpha, (s_1', Q_2'))$ such that $Q_2 \subseteq S_2$ and

1) $\alpha \in \Sigma_\tau(P_1)\setminus\Sigma(P_2)$, $s_1 \xrightarrow{\alpha}_1 s_1'$ and $Q_2 = Q_2' \subseteq S_2$, or
2) $\alpha \in O_1 \cap \Sigma(P_2)$, $s_1 \xrightarrow{\alpha}_1 s_1'$ and $Q_2' = \mathrm{cl}_{\Sigma_\tau(P_2)\setminus\Sigma(P_1)}(\{s_2' \in S_2 \mid \exists s_2 \in Q_2 : s_2 \xrightarrow{\alpha}_1 s_2'\})$, or
3) $\alpha \in I_1 \cap \Sigma(P_2)$, $s_1 \xrightarrow{\alpha}_1 s_1'$ and $Q_2' = \mathrm{cl}_{\Sigma_\tau(P_2)\setminus\Sigma(P_1)}(\{s_2' \in S_2 \mid \exists s_2 \in Q_2 : s_2 \xrightarrow{\alpha}_1 s_2'\})$ is non-empty.

It is straightforward to check that the natural constriction of P_1 w.r.t. P_2 is an IA. Moreover, the operator not only satisfies our requirements of the constriction operator but produces IA that are greater in terms of alternating simulation than its primary (left hand side) operand.

Lemma 22. *Let P, P_1, P_2, Q_1, Q_2 be IA. Then the following holds.*

1) *If P_1, P_2 are composable, then $P_1\upharpoonright_{P_2}$ is defined.*
2) $P\upharpoonright_{P_{id}} \equiv^I_O P$.
3) *If P, P_1 are composable and $P_1 \equiv^I_O P_2$, then $P_1\upharpoonright_P \equiv^I_O P_2\upharpoonright_P$ and $P\upharpoonright_{P_1} \equiv^I_O P\upharpoonright_{P_2}$.*
4) *If P, Q_1, Q_2 are pairwise composable, then $(P\upharpoonright_{Q_1})\upharpoonright_{Q_2} \equiv^I_O P\upharpoonright_{Q_1\|Q_2}$.*
5) *If P_1, P_2 and P_2, Q are composable, then $(P_1\upharpoonright_{P_2}) \parallel (P_2\upharpoonright_Q) \equiv^I_O P_1 \parallel (P_2\upharpoonright_Q)$.*
6) *If P_1, P_2 are composable, then $P_1 \gtrless^I_O P_1\upharpoonright_{P_2}$.*

Example 23. Let us illustrate the natural IA constriction operator on our running example. Consider the LTSs L_G, A_G, L_A and A_A from Figure 2 and 3. Let us regard these LTSs as IA $P_G := \mathit{IA}(L_G, I_G)$, $Q_G := \mathit{IA}(A_G, I_G)$, $P_A := \mathit{IA}(L_A, I_A)$ and $Q_A := \mathit{IA}(A_A, I_A)$, respectively, where the input alphabets are $I_G = \{0, 1\}$ and $I_A = \{00, 01, 10, 11\}$. This is possible because L_G and A_G are I_G-deterministic and L_A and A_A are I_A-deterministic. Q_G and Q_A are shown in Figure 5, where inputs are marked with the question and outputs with the exclamation mark. Analogously to LTSs, the constriction of P_G w.r.t. Q_A is Q_G, and the constriction of P_A w.r.t. Q_G is Q_A, i.e., $P_G\upharpoonright_{Q_A} = Q_G$ and $P_A\upharpoonright_{Q_G} = Q_A$. $\square$

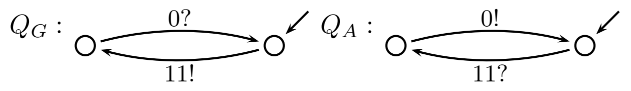


Fig. 5. IA Q_G and Q_A representing the expected behaviour of the generator *Gen* and the adder *Add* within the system.

By using Lemma 22, we can show that every IA AG rule on the natural constriction is sound and can be replaced by a simple IA AG rule. This is the first main result of the paper.

Theorem 24 (Complex IA AG rules are unnecessary). *Let n be an integer such that $n \geq 2$, $\{P_1, \ldots, P_n\}$ and $\{Q_1, \ldots, Q_n\}$ two sets of pairwise composable IA, $\Gamma : \mathcal{I}^{2n} \to \{\mathit{false}, \mathit{true}\}$ a side condition, and for every $i \in \{1, \ldots, n\}$, let $E_i := Q_{i,1} \parallel \cdots \parallel Q_{i,k_i}$, where $\{Q_{i,1}, \ldots, Q_{i,k_i}\}$ is a possibly empty subset of $\{Q_1, \ldots, Q_n\} \setminus \{Q_i\}$. Then*

$$\frac{P_1\upharpoonright_{E_1} \gtrless^I_O Q_1 \quad \cdots \quad P_n\upharpoonright_{E_n} \gtrless^I_O Q_n}{\|_{i=1}^n P_i \gtrless^I_O \|_{i=1}^n Q_i}\ \Gamma(P_1, Q_1, \ldots, P_n, Q_n)$$

is a sound AG rule and

$$\mathit{SIA}\!: \quad \frac{P_1 \gtrless^I_O Q_1 \qquad \cdots \qquad P_n \gtrless^I_O Q_n}{\|_{i=1}^n P_i \gtrless^I_O \|_{i=1}^n Q_i}$$

is a simple and sound AG rule at least as general as the one above.

The theorem can be extended to any IA constriction operator that satisfies Item 6 of Lemma 22, i.e., $P_1 \gtrless^I_O P_1|_{P_2}$. It is an open question whether there are also IA constriction operators which do not satisfy the property.

Example 25. We can now see how the IA framework allows us to solve the problem with our running example and achieve our objective, namely, to use the models of Figure 3 as abstractions for the models of Figure 2 in a simple AG rule. Specifically, consider again the IA P_G, Q_G, P_A, Q_A from Ex. 23. Observe that $P_G \gtrless^I_O Q_G$ and $P_A \gtrless^I_O Q_A$. Hence, we can directly use the SIA rule to deduce that

$$P_G \parallel P_A \gtrless^I_O Q_G \parallel Q_A \ .$$

Obviously, $Q_G \parallel Q_A$ has no error state, which implies that $P_G \parallel P_A$ has no error state either. Moreover, since both compositions $P_G \parallel P_A$ and $Q_G \parallel Q_A$ are closed IA, we have

$$\mathit{LTS}(P_G \parallel P_A) \preceq_{\mathrm{sim}} \mathit{LTS}(Q_G \parallel Q_A) \ .$$

In other words, only 0 is seen in the channel z and 1 in the channels x and y. $\square$

Theorem 24 says that in the IA framework all we need is SIA. SIA, in turn, is better than the sound LTS AG rule of Theorem 15 in two ways: there is no side condition and all the premisses are simple, i.e., they do not involve constriction. However, even though we were able to exploit SIA in place of the sound LTS AG rule in our example system, we have not yet shown whether we can do so in general. In the following section, we will prove that this is indeed possible.

VII. Reducing LTS AG Rule to IA AG Rule

In this section, we present the second main contribution of the paper. We show how our sound LTS AG rule can be converted into a sound IA AG rule, provided the LTSs can be interpreted as IA without violating the assumption on input-determinism. The conversion is done by (i) making the LTSs representing concrete processes input-enabled and (ii) by removing unnecessary input transitions from abstract LTSs.

Given an LTS $L := (S, E, \longrightarrow, \hat{s})$ and a set I of visible events, we write $en(L, I)$ for a quadruple

$$en(L, I) := (S \cup \{\bar{s}\}, E, \longrightarrow \cup \longrightarrow' \cup (\{\bar{s}\} \times (I \cap E) \times \{\bar{s}\}), \hat{s}) \ ,$$

where $\bar{s} \notin S$ is a new state, used as a sink, and

$$\begin{aligned}\longrightarrow' = \{&(s, \alpha, \bar{s}) \mid s \in S, \alpha \in I \cap E, s \not\overset{\alpha}{\Longrightarrow}\} \\ &\cup \{(s, \alpha, s') \mid \alpha \in I \cap E, s \overset{\alpha}{\Longrightarrow} s'\} \ .\end{aligned}$$

Obviously, $en(L, I)$ is an $(I \cap E)$-enabled LTS. We write $enIA(L, I)$ short for $IA(en(L, I), I)$.

Theorem 26 (From LTS AG proofs to SIA proofs)**.** *Let $L_1, \ldots, L_n$, $A_1, \ldots, A_n$ and $C_1, \ldots, C_n$ be LTSs satisfying the premisses and the side condition of the sound LTS AG rule of Theorem 15 and let $I_i := \{\alpha \in \Sigma(A_i) \mid j_\alpha \neq i\}$ for all $i \in \{1, \ldots, n\}$, where j_α is as defined in Theorem 15. Moreover, for every $i \in \{1, \ldots, n\}$, let L_i be I_i-deterministic, $P_i := enIA(L_i, I_i)$ and $Q_i := IA(A_i{\restriction}_{C_i}, I_i)$.*

Then $\{P_i\}_{i=1}^n$ and $\{Q_i\}_{i=1}^n$ are sets of pairwise composable IA satisfying the premisses of Theorem 24 and $\|_{i=1}^n L_i \preceq_{\mathrm{sim}} LTS(\|_{i=1}^n P_i) \preceq_{\mathrm{sim}} LTS(\|_{i=1}^n Q_i) \preceq_{\mathrm{sim}} \|_{i=1}^n A_i$.

Note that the theorem applies to LTS AG proofs on any LTS constriction operator but when the abstract LTSs are converted into IA, the natural LTS constriction operator is used. The theorem also says that the composition of the converted LTSs model the same concrete system and abstract system as the composition of original LTSs.

Example 27. Let us apply Theorem 26 to our running example. Let $L_1 := L_G$ and $L_2 := L_A$ from Figure 2, $A_1 := A'_G$ and $A_2 := A'_A$ from Figure 4, $I_1 := \{0, 1\}$ and $I_2 := \{00, 10, 10, 11\}$. These LTSs and sets of events satisfy the assumptions of Theorem 26. By applying the theorem, we obtain the concrete IA $P_1 := enIA(L_1, I_1)$ and $P_2 := enIA(L_2, I_2)$ in Figure 6 and the abstract IA $Q_1 := IA(A_1{\restriction}_{A_2}, I_1)$ and $Q_2 := IA(A_2{\restriction}_{A_1}, I_2)$, which turn out to be equal to Q_G and Q_A, respectively, in Figure 5. As can be seen, the abstractions capture the expected behaviour of *Gen* and *Add* when they are composed with each other. Moreover, even though the concrete IA are different from P_G and P_A, their parallel composition is the same, i.e., the reachable parts of $P_1 \parallel P_2$ and $P_G \parallel P_A$ are equal. Hence, $LTS(P_1 \parallel P_2)$ is a correct model of our example system. □

The application of Theorem 26 may not give any complexity gains but if we need to do AG reasoning, it makes sense to give the IA formalism a try. That is because everything you can prove by using the LTS AG rule can be proved in the IA world as well and the AG proofs on IA can be made at least conceptually simpler as complex rules are not needed. This suggests that circular reasoning is a built-in feature of the IA formalism and if you can model your system components as IA, complex assume-guarantee rules are not needed.

It is an open question whether we can prove a similar theorem purely in the LTS framework, i.e., that all LTS AG proofs can be made simple. However, it is unlikely that we could do this by moving constriction to the abstract side. To see this, consider the following example:

1) L_1 is an LTS which executes first a and then b,
2) L_2 is an LTS which blocks both a and b,
3) A_1 is an LTS which blocks b but can execute a repeatedly,
4) A_2 is an LTS which blocks a but can execute b repeatedly.

Now, the premisses of Theorem 12 hold: $L_1{\restriction}_{A_2} \preceq_{\mathrm{sim}} A_1$ and $L_2{\restriction}_{A_1} \preceq_{\mathrm{sim}} A_2$. Also the side condition is true because A_1 is $\{a\}$-enabled and A_2 is $\{b\}$-enabled, i.e., a is thought as an input of L_1 and A_1. Hence, the theorem is applicable, However, there is no obvious way to move constriction to the abstract side because $L_1 \preceq_{\mathrm{sim}} A_1{\restriction}_C$ does not hold no matter how we choose C and even if we re-enable the input event a after constriction.

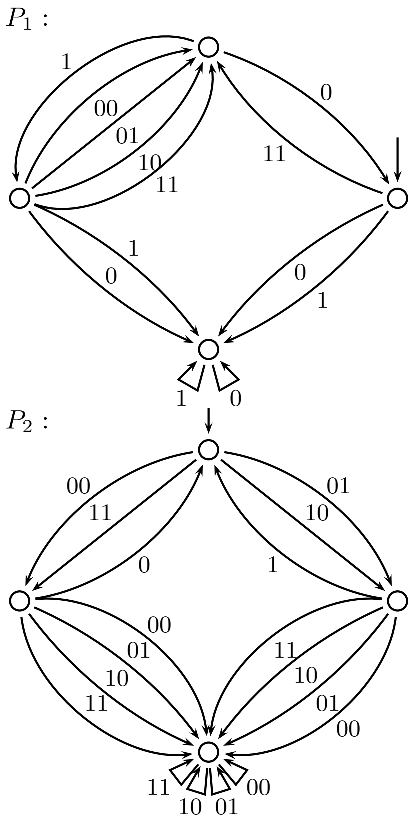


Fig. 6. IA P_1 and P_2 representing the behaviour of the input-enabled generator *Gen* and the adder *Add* components.

VIII. Related Work

Our work is most closely related to [11] which proposes a sound and complete circular AG rule (C3) and shows that every proof derived using the C3 rule can be translated into a proof which uses a non-circular rule (NC). In [11], the specifications are given in LTL and processes execute synchronously, whereas our focus is on refinement-based formalisms where processes execute partly asynchronously.

Results similar to Theorem 15 have been presented in other works, e.g., Rule (5) in [21], although it is worth noting that Theorem 15 is generic in the sense that it holds for any constriction operator instead of a specific parallel composition operator. Still, Theorem 15 is not the main focus of this paper, and it is stated merely to be able to state the result of Theorem 26.

Non-circular assume-guarantee rules are provided in [10] for LTSs, in [23] for IA, and in [24] for I/O components. Moreover, the focus of two former papers is on the automatic generation of abstract models using learning techniques.

Our work is also loosely related to works studying the completeness of AG rules [11], [24]. However, we explore the need for complex circular AG rules, not their completeness. You should also note that a number of AG rules can be shown to be complete, albeit in a straightforward and not very useful way. For instance, SIA with transitivity

$$\frac{P_1 \gtrless^I_O Q_1 \quad \cdots \quad P_n \gtrless^I_O Q_n \qquad\qquad \|_{i=1}^n Q_i \gtrless^I_O R}{\|_{i=1}^n P_i \gtrless^I_O R}$$

is trivially complete by setting $Q_i := P_i$.

Contrary to the SIA rule which is set in the IA framework, many AG rules are formulated within frameworks which make no distinction between inputs and outputs (this is the case, for instance, for the rules in [6], [11]). This is perhaps surprising, given that identifying the assertions holding in the *interface* between processes is recognised as very important for compositional verification [11].

Compositional verification frameworks that do distinguish inputs and outputs, other than IA, are *input/output automata* [25] and *reactive modules* [8]. AG rules similar to SIA and NC have been proven for I/O automata as well as for reactive modules. In both I/O automata and reactive modules processes are forbidden to block inputs, whereas this is allowed in IA, which can therefore be seen as a generalisation. (For arguments on why this generalisation is useful, see [14], [16], [26]).

Interface theories are not introduced with compositional verification as their main goal. Instead, as argued in [26], they are used as "lightweight" verification methods akin to type-checking (only check compatibility of components without checking any global property). Nevertheless, interface theories *can* be used as a compositional verification tool (c.f. [23]), and the question addressed in this paper is how this tool compares to assume-guarantee frameworks.

IX. Conclusions

The main contributions of this paper are two. First, we show that all circular assume-guarantee rules for interface automata are sound and can be expressed in a simple non-circular form without side conditions. Second, we show that proofs in a sound LTS AG rule can be translated into proofs using this simple IA rule.

These results suggest that circular reasoning is a built-in feature of the IA formalism, and using IA makes compositional proofs at least conceptually simpler. Hence, if you can model your system components as IA, complex assume-guarantee rules are not needed. Whether IA should be preferred in practice depends however also on other types of considerations, in particular algorithmic efficiency and availability of suitable tools. While checking alternating simulation is as hard as checking standard simulation from a worst-case complexity perspective [27], there is still a shortage of tools implementing this and other features of the IA framework.

You should also note that the theorems on sound LTS and IA AG rules can be extended to AG rules of the form

$$\frac{P_1|_{D_1\|C_1} \preceq Q_1|_{E_1} \quad \cdots \quad P_n|_{D_n\|C_n} \preceq Q_n|_{E_n}}{\|_{i=1}^n P_i \preceq \|_{i=1}^n Q_i}\;\Gamma,$$

where Γ is the side condition of the corresponding theorem and for all $i \in \{1, \ldots, n\}$,

1) $C_i := Q_{i,1} \parallel \cdots \parallel Q_{i,k_i}$, where k_i is a non-negative integer and $\{Q_{i,1}, \ldots, Q_{i,k_i}\} \subseteq \{Q_1, \ldots, Q_n\} \setminus \{Q_i\}$,
2) $D_i := P_{i,1} \parallel \cdots \parallel P_{i,l_i}$, where l_i is a non-negative integer and $\{P_{i,1}, \ldots, P_{i,l_i}\} \subseteq \{P_1, \ldots, P_n\} \setminus \{P_i\}$.
3) $E_i := Q_{i,1} \parallel \cdots \parallel Q_{i,m_i}$, where m_i is a non-negative integer and $\{Q_{i,1}, \ldots, Q_{i,m_i}\} \subseteq \{Q_1, \ldots, Q_n\} \setminus \{Q_i\}$.

In other words, we may allow the behaviour of abstract processes to be constricted by other abstract processes as well as the behaviour of concrete processes to be constricted by both abstract and concrete processes. This is possible because by the properties 3–4 of a constriction operator, we may write premisses in the form $(P_i|_{D_i})|_{C_i} \preceq Q_i|_{E_i}$ and, provided that the premisses and the side condition hold, conclude that $\|_{i=1}^n (P_i|_{D_i}) \preceq \|_{i=1}^n (Q_i|_{E_i})$. After that, we can apply the properties 3–5 of the constriction operator and deduce that $\|_{i=1}^n P_i \equiv \|_{i=1}^n (P_i|_{D_i}) \preceq \|_{i=1}^n (Q_i|_{E_i}) \equiv \|_{i=1}^n Q_i$.

We formulated our results using IA and LTSs, since IA is the most well-known interface theory. We expect similar results to be obtainable in a synchronous setting, using synchronous transition systems and the synchronous interfaces of [26]. Reporting on such results is a part of future work. In general, we believe that results similar to Theorem 24 can be proved for most compositional event-based formalisms, such as modal IA [17], where each event is controlled by at most one process and refinement allows for weakening input assumptions while strengthening output guarantees.

Acknowledgements

This work was partially supported by the Academy of Finland and by the NSF via projects *COSMOI: Compositional System Modeling with Interfaces* and *ExCAPE: Expeditions in Computer Augmented Program Engineering*, and by the iCy-Phy Research Center supported by IBM and United Technologies. We would also like to thankfully acknowledge the funding from the SARANA project in the SAFIR 2014 program and the Academy of Finland projects 139402 and 277522.

References

[1] J. Misra and K. Chandy, "Proofs of networks of processes," *IEEE Trans. Softw. Eng.*, vol. 7, no. 4, pp. 417–426, Jul. 1981.

[2] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems*, ser. NATO ASI Series, K. R. Apt, Ed. Springer, 1985, vol. 13, pp. 123–144.

[3] O. Grumberg and D. Long, "Model checking and modular verification," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 843–871, 1994.

[4] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 507–535, 1995.

[5] N. Shankar, "Lazy compositional verification," in *Compositionality: The Significant Difference*, ser. LNCS, W.-P. de Roever, H. Langmaack, and A. Pnueli, Eds. Springer, 1998, vol. 1536, pp. 541–564.

[6] K. McMillan, "Verification of an Implementation of Tomasulo's Algorithm by Compositional bodel Checking," in *CAV '98*, ser. LNCS, A. J. Hu and M. Y. Vardi, Eds., vol. 1427. Springer, 1998, pp. 110–121.

[7] T. Henzinger, S. Qadeer, and S. Rajamani, "You assume, we guarantee: Methodology and case studies," in *CAV '98*, ser. LNCS, A. J. Hu and M. Y. Vardi, Eds. Springer, 1998, vol. 1427, pp. 440–451.

[8] R. Alur and T. Henzinger, “Reactive modules,” *Form. Method. Syst. Des.*, vol. 15, pp. 7–48, 1999.

[9] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[10] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, “Learning assumptions for compositional verification,” in *TACAS’03*, ser. LNCS, H. Garavel and J. Hatcliff, Eds. Springer, 2003, vol. 2619, pp. 331–346.

[11] K. S. Namjoshi and R. J. Trefler, “On the completeness of compositional reasoning methods,” *ACM Trans. Comput. Logic*, vol. 11, no. 3, May 2010.

[12] R. Milner, *A Calculus of Communicating Systems*, ser. LNCS. Springer-Verlag, 1980, vol. 92.

[13] A. W. Roscoe, *Understanding Concurrent Systems*. Springer, 2010.

[14] L. de Alfaro and T. Henzinger, “Interface automata,” *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 109–120, 2001.

[15] ——, “Interface-based design,” in *Engineering Theories of Software Intensive Systems*, ser. NATO Science Series, M. Broy, J. Grnbauer, D. Harel, and T. Hoare, Eds. Springer, 2005, vol. 195, pp. 83–104.

[16] ——, “Interface theories for component-based design,” in *EMSOFT ’01*, ser. LNCS, T. A. Henzinger and C. M. Kirsch, Eds. Springer, 2001, vol. 2211, pp. 148–165.

[17] G. Lüttgen and W. Vogler, “Modal interface automata,” *Log. Meth. Comput. Sci.*, vol. 9, no. 3, 2013.

[18] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone, “A modal interface theory for component-based design,” *Fundam. Inform.*, vol. 108, no. 1–2, pp. 119–149, 2011.

[19] S. Bauer, P. Mayer, A. Schroeder, and R. Hennicker, “On weak modal compatibility, refinement, and the MIO Workbench,” in *TACAS ’10*, ser. LNCS, J. Esparza and R. Majumdar, Eds. Springer, 2010, vol. 6015, pp. 175–189.

[20] Online appendix, available: http://cc.oulu.fi/~asiirtol/papers/cag_app.pdf.

[21] G. Frehse, Z. Han, and B. Krogh, “Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction,” in *CDC ’04*, vol. 1. IEEE, Dec 2004, pp. 479–484.

[22] F. Bujtor and W. Vogler, “Error-pruning in interface automata,” in *SOFSEM ’14*, ser. LNCS, V. Geffert, B. Preneel, B. Rovan, J. tuller, and A. Tjoa, Eds. Springer, 2014, vol. 8327, pp. 162–173.

[23] M. Emmi, D. Giannakopoulou, and C. S. Păsăreanu, “Assume-guarantee verification for interface automata,” in *FM ’08*, ser. LNCS, J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds. Springer, 2008, vol. 5014, pp. 116–131.

[24] C. Chilton, B. Jonsson, and M. Kwiatkowska, “Compositional assumeguarantee reasoning for input/output component theories,” *Sci. Comput. Program.*, vol. 91, no. A, pp. 115 – 137, 2014, special Issue on FACS 12.

[25] N. Lynch and M. Tuttle, “An introduction to input/output automata,” *CWI Quarterly*, vol. 2, pp. 219–246, 1989.

[26] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, “A theory of synchronous relational interfaces,” *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 4, Jul. 2011.

[27] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi, “Alternating refinement relations,” in *CONCUR ’98*, ser. LNCS, D. Sangiorgi and R. de Simone, Eds. Springer, 1998, vol. 1466, pp. 163–178.