Compositional Verification in Linear-Time Temporal Logic*

Extended Abstract

Yih-Kuen Tsay

Department of Information Management, National Taiwan University
tsay@im.ntu.edu.tw

Abstract. In the compositional verification of a concurrent system, one seeks to deduce properties of the system from properties of its constituent modules. This paper supplements our previous work on the same subject to provide a comprehensive compositional framework in linear-time temporal logic. It has been shown by many that specifying properties of a module in the assumption-guarantee style is effective in achieving compositionality. We consider two forms of temporal formulas that correspond to two interpretations of an assumption-guarantee specification and investigate how they can be applied in compositional verification. We argue by examples that the two forms complement each other and both are needed to facilitate the compositional approach. We also show how to handle assumption-guarantee specifications where the assumption contains a liveness property.

1 Introduction

A concurrent system typically is or can be decomposed as the parallel composition of several modules. In the compositional verification of a system, one seeks to deduce properties of the system from properties of its constituent modules. We assume that the system to be verified is closed, i.e., the system is meant to be executed in isolation (without any interferences, except perhaps part of the initialization, from the environment). Nonetheless, we provide sufficient details showing how the results of this paper can be extended straightforwardly to the compositional verification of an open system, which is essentially a module.

Properties of a system are represented by assertions on computations of the system and so are properties of a module. Computations of a system are the sequences of states produced when the system is executed in isolation. In contrast, computations of a module are the sequences of states produced when the module is executed in parallel with an arbitrary but (syntactically) compatible environment, i.e., the computations of an imaginary system obtained from composing the module with the arbitrary environment. A system or module

* This research was supported in part by grants NSC 86-2213-E-002-002 and NSC 87-2213-E-002-015 from the National Science Council, Taiwan (R.O.C.) and a research award from College of Management, National Taiwan University.

J. Tiuryn (Ed.): FOSSACS 2000, LNCS 1784, pp. 344–358, 2000.
© Springer-Verlag Berlin Heidelberg 2000

satisfies a certain property if the corresponding assertion holds for each of its computations.

A module will behave properly only if its environment does. When specifying properties of a module, one should therefore include (1) assumed properties about its environment and (2) guaranteed properties of the module if the environment obeys the assumption. This type of specification is essentially a generalization of pre and post-conditions for sequential programs [14]. The generalization was adopted in the early 1980's by Misra and Chandy [23], Jones [15], and Lamport [19] and became the so-called *assumption-guarantee* (also known as rely-guarantee or assumption-commitment) paradigm.

Consider an assumption-guarantee specification with assumption A and guarantee G. There are at least two possible interpretations of the specification over a sequence of states. Informally, one interpretation states that G holds at least one step longer than A does. The other states that G holds as long as A does, which is a weaker interpretation than the first. A third even weaker interpretation is the ordinary implication from A to G; however, it is practically equivalent to the second interpretation, as a module should not have the ability to predict the future behavior of its environment and hence the future violation of A by its environment. We refer to properties according to the first interpretation as *strong* assumption-guarantee properties and those according to the second as *weak* assumption-guarantee properties. As has been pointed out by Abadi and Lamport [2], if A and G cannot be falsified simultaneously by any step of the module or its environment, then the two interpretations are equivalent.

In this paper, we intend to further advance the use of temporal logic in specifying and reasoning about assumption-guarantee properties and investigate how this kind of properties can be applied in compositional verification. Temporal logic is one convenient formalism for specifying the behavior of a concurrent system. The idea of representing concurrent systems and their specifications as formulas in temporal logic was first proposed by Pnueli [24].

We have proposed in [16] to formulate assumption-guarantee specifications using the linear-time temporal logic (LTL) of Manna and Pnueli [21]. We showed how to specify and reason about strong assumption-guarantee properties in the full set of LTL. Our formulation of assumption-guarantee specifications as well as the derived composition rules are syntactic and entirely within LTL. *This paper complements and differs from our previous work in three aspects: First, we consider both strong and weak (not just strong) assumption-guarantee properties in this paper. Second, we emphasize the use of assumption-guarantee specifications in compositional verification rather than hierarchical development. Last, we extend the previous work to include composition rules that permit assumptions with liveness properties.* Hiding (of local variables) is not treated here for a more focused exposition; it can be handled syntactically in the same way as in our previous work. Together this paper and the previous work provide a comprehensive compositional framework in LTL.

Related works on assumption-guarantee specifications, including [23, 15, 19, 12, 1, 3, 2, 9, 10, 27], typically reason about relevant properties at the semantic

level or define a special-purpose logic. In [1], Abadi and Lamport gave a comprehensive treatment of compositionality in a general semantic setting with agents (which are used essentially for identifying a module). Their semantic composition rule used the notion of the "realizable part" of a specification which in general cannot be extracted by simpler operations on the specification. Xu, Cau, and Collette [27] provided an explanation of the difference between two well-known composition rules respectively for message-passing and shared-variable models. They show that the two rules can be derived from a more general one. In [4], Alur and Henzinger suggested the notion of local liveness in place of the weaker notion of receptiveness involved in the compositionality issue. They argue that receptiveness is unnecessarily weak and computationally hard to check, while local liveness on the other hand is satisfied by most existing models and is easier to check. A collection of survey papers on the general subject of compositional verification has recently been published as [11].

Barringer and Kuiper [6] are, to our knowledge, the first to formulate assumption-guarantee specifications in temporal logic. They used the notion of an agent and considered only strong assumption-guarantee properties. Manna and Pnueli proposed a compositional verification rule using weak assumption-guarantee properties in their recent book [22]. Using the Temporal Logic of Actions (TLA, a variant of temporal logic) [20], the work of Abadi and Lamport [2] is an improvement over earlier temporal logic-based works in handling hiding and liveness properties. They focused on assumption-guarantee specifications where the assumption and the guarantee cannot be falsified simultaneously. With a limited set of temporal operators in TLA, they had to work mostly at the semantic level. Abadi and Lamport's formulation of an assumption-guarantee specification allows liveness properties in the assumption part. However, their composition rule only works for safety assumptions. Collette [10], adapting the work of [2], proposed a UNITY-like [7] logic for assumption-guarantee specifications with restricted forms of assumption and guarantee.

Assumption-guarantee specifications have also found applications in the area of model checking [8]. They are useful for compositional (or modular) model checking, which provides one possible way to tackle the state-explosion problem. Virtually all existing works on modular model checking are for branching-time temporal logic or a combination of linear-time and branching-time logics. Grumberg and Long [13] considered a subset of CTL (Computation Tree Logic, a branching-time temporal logic) for which satisfaction is preserved under parallel composition. In their work, the assumption of the specification of a module is represented by another abstract module; the composition of the two modules is then checked against the desired property. In [5], Aziz et. al. proposed to reduce the size of each module of a system via an equivalence so that the given specification is preserved. Their method handles full CTL. The complexity of modular model checking of CTL formulas has been shown to be at least as high as that of (propositional) LTL formulas by Kupferman and Vardi [26, 17, 18].

2 Preliminaries

2.1 Temporal Logic

Linear-time temporal logic (LTL) is a logic for expressing assertions on infinite sequences of states, where each state is an assignment to a predefined universe of variables. An LTL formula is interpreted with respect to a position $i \geq 0$ in a sequence of states. *State formulas* are the basic type of LTL formula built only from variables, constants, functions, and predicates using the usual first-order logic connectives. The interpretation of a state formula in position i is performed as usual using the particular interpretation of variables in state i (plus the fixed interpretations of constants, functions, and predicates). General LTL formulas also contain temporal operators; in this paper, we will use only the following:

- $\bigcirc$ means "in the next state". The formula $\bigcirc\varphi$ is true in position i of a sequence σ (denoted $(\sigma, i) \models \bigcirc\varphi$) iff φ is true in position $i+1$ of σ (i.e., $(\sigma, i+1) \models \varphi$).
- $\Box$ means "always in the future (including the present)"; $(\sigma, i) \models \Box\varphi$ iff $\forall k \geq i : (\sigma, k) \models \varphi$.
- $\ominus$ means "in the previous state, if there is any"; $(\sigma, i) \models \ominus\varphi$ iff $(i > 0) \rightarrow ((\sigma, i-1) \models \varphi)$. $\circleddash$ is a weaker version of $\ominus$, which means "in the previous state"; $(\sigma, i) \models \circleddash\varphi$ iff $(i > 0) \wedge ((\sigma, i-1) \models \varphi)$. It follows that $(\sigma, i) \models \circleddash\varphi$ iff $(\sigma, i) \models \neg\ominus\neg\varphi$.
- $\boxminus$ means "always in the past (including the present)"; $(\sigma, i) \models \boxminus\varphi$ iff $\forall k : 0 \leq k \leq i : (\sigma, k) \models \varphi$.
- For a variable u, the interpretation of u^- (the previous value of u) in position i is the same as the interpretation of variable u in position $i-1$; by convention, the interpretation of u^- in position 0 is the same as the interpretation of u in position 0.[1]
- *first* is an abbreviation for $\circleddash$*false*, which is true only in position 0.

We say that a sequence σ satisfies a formula φ (or φ holds for σ) if $(\sigma, 0) \models \varphi$. A formula φ is *valid*, denoted $\models \varphi$ or simply φ when it is clear that validity is intended, if φ is satisfied by every sequence.

A formula without temporal operators but possibly with "$^-$"-superscribed variables is called a *transition formula*; this definition is slightly different from that in [21], where a transition formula always contains $\neg$*first* as a conjunct. A formula without any future operator $\bigcirc$, $\Box$, or $\Diamond$ (though liveness is considered, $\Diamond$ is not explicitly used in this paper) is called a *past formula*; in particular, a transition formula is a past formula. A *safety formula* is one that specifies a safety property and a *liveness formula* is one that specifies a liveness property. Of

[1] In contrast to Lamport and others who use "$^+$"-superscribed (or primed) variables to denote their values in the next state, we use "$^-$"-superscribed variables to denote their values in the previous state. The reason is that (for conformity) we wish to use only past operators, except the outmost $\Box$, in the safety part of a specification. The introduction of "$^-$"-superscribed variables is convenient but not essential, since they can be encoded by the $\circleddash$ operator.

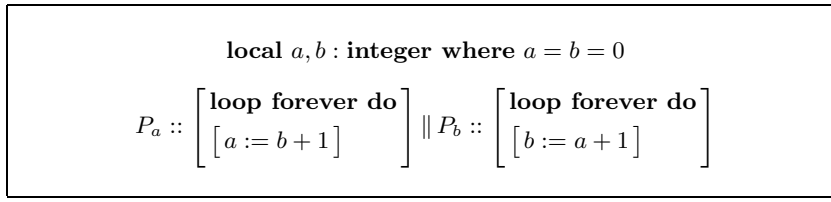

Fig. 1. Program KEEP-AHEAD.

particular importance, formulas of the form $\Box H$, where H is a past formula, are for certain safety formulas; they will be referred to as *canonical safety formulas*. Specific forms of liveness formulas are not important for our purposes. Formulas of the form $\Box H \wedge L$, where H is a past formula and L a liveness formula, will be referred to as *canonical formulas*.

2.2 Specifying Concurrent Systems

A concurrent system consists of a set of variables, an initial condition on the variables, and a set of transitions that specify how the system may change the values of its variables in an execution step. Semantically, a concurrent system is associated with a set of computations or sequences of states, each of which represents a possible execution of the system. We will mostly concentrate on safety properties of a system. For our purpose, we distinguish two kinds of specification: system specification and requirement specification.

System specifications are basically programs in the form of a temporal formula. Consider Program KEEP-AHEAD in Figure 1. The system specification of KEEP-AHEAD is given by $\Phi_{\text{KEEP-AHEAD}}$ as defined below.

$$\Phi_{\text{KEEP-AHEAD}} \triangleq (a = 0) \wedge (b = 0) \wedge \Box \begin{pmatrix} (a = b^- + 1) \wedge (b = b^-) \\ \vee\ (b = a^- + 1) \wedge (a = a^-) \\ \vee\ (a = a^-) \wedge (b = b^-) \end{pmatrix}$$

The formula $\Phi_{\text{KEEP-AHEAD}}$ states that the values of a and b are initially 0. It also states via the disjunction of three transition formulas that, in each step of an execution, either the value of a becomes $b + 1$ (while the value of b is unchanged), the value of b becomes $a + 1$, or nothing is changed. The transition formula $(a = a^-) \wedge (b = b^-)$ is called a stuttering transition and is included to make the specification invariant under stuttering.

We regard system specifications as formal definitions of concurrent systems so that we can do without a formal semantics of the programming language; programs are informal notations for readability. To take fairness into account, one may conjoin an appropriate liveness formula to the system specification. The safety formula in a system specification can be put in the canonical form of $\Box H$, specifically in the form of $\Box((\mathit{first} \wedge \mathit{Init}) \vee (\neg \mathit{first} \wedge N))$, where Init is a state formula and N the disjunction of several transition formulas. As N will

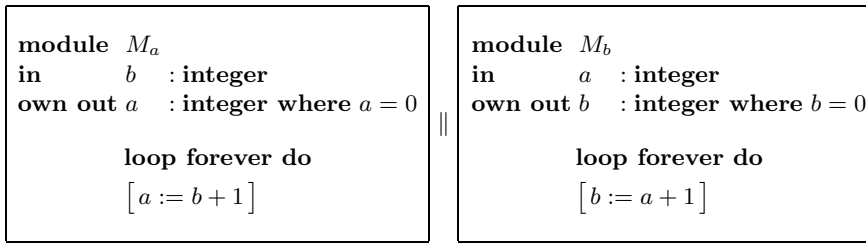

Fig. 2. Program KEEP-AHEAD as the parallel composition of two modules.

always contain a stuttering transition, $\Box((\mathit{first} \wedge \mathit{Init}) \vee (\neg \mathit{first} \wedge N))$ simplifies to $\Box((\mathit{first} \wedge \mathit{Init}) \vee N)$.

Requirement specification is the usual type of temporal-logic specification. A property is represented by a temporal formula. A system (program) S is said to satisfy a formula φ if every computation of S satisfies φ. Let Φ_S denote the system specification of S. We will regard $\Phi_S \rightarrow \varphi$ as the formal definition of the fact that S satisfies φ, denoted as $S \models \varphi$. The safety formula in a requirement specification can usually be put in the canonical form.

2.3 Parallel Composition as Conjunction

Program KEEP-AHEAD can be decomposed as the parallel composition of two modules as shown in Figure 2. A module may read but not change the value of an **in** (input) variable. A *compatible* environment of a module may read but not change the value of an **own out** (owned output) variable of the module. In the system $M_a \,\|\, M_b$, M_b is the environment of M_a and M_a is the environment of M_b; both are clearly compatible with each other.

The system specifications Φ_{M_a} and Φ_{M_b} of modules M_a and M_b respectively are defined as follows:

$$\Phi_{M_a} \triangleq (a = 0) \wedge \Box \begin{pmatrix} (a = b^- + 1) \wedge (b = b^-) \\ \vee\ (a = a^-) \end{pmatrix}$$

$$\Phi_{M_b} \triangleq (b = 0) \wedge \Box \begin{pmatrix} (b = a^- + 1) \wedge (a = a^-) \\ \vee\ (b = b^-) \end{pmatrix}$$

It is perhaps more accurate to say that Φ_{M_a} is the system specification of an imaginary system composed of M_a and an arbitrary but compatible environment; analogously, for Φ_{M_b}. A little calculation shows that

$$\models \Phi_{M_a} \wedge \Phi_{M_b} \leftrightarrow \Phi_{\text{KEEP-AHEAD}}.$$

This formally confirms that $M_a \,\|\, M_b$ is equivalent to Program KEEP-AHEAD.

A module M is said to satisfy a formula φ if every computation of M satisfies φ. Let Φ_M denote the system specification of M. Like in the case of specifying

properties of a concurrent system, we will regard $\Phi_M \rightarrow \varphi$ as the formal definition of the fact that M satisfies φ, denoted as $M \models \varphi$. Since parallel composition is conjunction, it follows that, if M is a module of system S, then $M \models \varphi$ implies $S \models \varphi$.

3 Assumption-Guarantee Specifications

We shall concentrate on assumption-guarantee specifications where both the assumption and the guarantee are safety properties; liveness will be treated in Section 6. We assume that safety properties are expressed as canonical safety formulas of the form $\Box H$, where H is a past formula.

3.1 Strong Assumption-Guarantee Formulas

Strong assumption-guarantee formulas specify strong assumption-guarantee properties. A strong assumption-guarantee property of a module with assumption A and guarantee G asserts the following:

> For every computation of the module, G holds initially and, for every $i > 1$, if A holds for the prefix of length $i-1$ (i.e., with $i-1$ states), then G also holds for the prefix of length i.

Notice that, if a safety property does not hold for a prefix of a computation, then the property will not hold for any longer prefix. The above assertion therefore says that G holds at least one step longer than A does.

As A and G are given respectively as $\Box H_A$ and $\Box H_G$, where H_A and H_G are past formulas, the strong assumption-guarantee property can be expressed as $\Box(\circleddash \boxminus H_A \rightarrow \boxminus H_G)$,[2] which is equivalent to $\Box(\circleddash \boxminus H_A \rightarrow H_G)$. Note that $\Box(\circleddash \boxminus H_A \rightarrow H_G)$ implies that H_G holds initially, since $\circleddash \boxminus H_A$ always holds in position 0 of a sequence. To summarize, we define strong assumption-guarantee formulas of the form $A \triangleright G$ as follows:

$$A \triangleright G \ (i.e., \Box H_A \triangleright \Box H_G) \ \stackrel{\Delta}{=} \ \Box(\circleddash \boxminus H_A \rightarrow H_G)$$

Note that $A \triangleright G$ is also a canonical safety formula.

Theorem 1. *Suppose that H_{G_1} and H_{G_2} are past formulas. Then,*

$$\models \ (\Box H_{G_1} \triangleright \Box H_{G_2}) \wedge (\Box H_{G_2} \triangleright \Box H_{G_1}) \ \rightarrow \ \Box H_{G_1} \wedge \Box H_{G_2}.$$

The above theorem is essentially the composition principle formulated by Misra and Chandy [23]. This small result shows that strong assumption-guarantee formulas have a mutual induction mechanism built in and hence permit "circular

[2] Since H_A and H_G are past formulas, "$\Box H_A$ holds for the prefix of length $i-1$ of σ" can be formally stated as "$(\sigma, i) \models \circleddash \boxminus H_A$" and "$\Box H_G$ holds for the prefix of length i of σ" as "$(\sigma, i) \models \boxminus H_G$".

reasoning" (there is of course no real cycle if one looks at the semantic models and reasons state by state from the initial one), i.e., deducing new properties from mutually dependent properties.

We now state a general rule for composing strong assumption-guarantee properties; this rule has been proven in [16].

Theorem 2. *Suppose that* $A_i \equiv \Box H_{A_i}$, $G_i \equiv \Box H_{G_i}$, $A \equiv \Box H_A$, *and* $G \equiv \Box H_G$, *all in the canonical form. Then,*

$$\frac{\begin{array}{l} 1.\ \models\ \Box\Big(\boxminus H_A \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_{A_j}\Big) \text{ for } 1 \le j \le n \\ 2.\ \models\ \Box\Big(\circledcirc \boxminus H_A \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_G\Big) \end{array}}{\models\ \bigwedge_{i=1}^{n} (A_i \triangleright G_i)\ \rightarrow\ (A \triangleright G)}$$

Intuitively, Premise 1 of the above composition rule says that the assumption about the environment of a module should follow from the guarantees of other modules and the assumption about the environment of the entire (open) system, while Premise 2 says that the guarantee of the entire system should follow from the guarantees of individual modules and the assumption about its environment. For closed systems, we take A to be *true* and simplify the rule as follows:

Theorem 3. *Suppose that* $A_i \equiv \Box H_{A_i}$, $G_i \equiv \Box H_{G_i}$, *and* $G \equiv \Box H_G$, *all in the canonical form. Then,*

$$\frac{\begin{array}{l} 1.\ \models\ \Box\Big(\boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_{A_j}\Big) \text{ for } 1 \le j \le n \\ 2.\ \models\ \Box\Big(\boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_G\Big) \end{array}}{\models\ \bigwedge_{i=1}^{n} (A_i \triangleright G_i)\ \rightarrow\ G}$$

Theorem 1, stated earlier, follows immediately from this theorem.

3.2 Weak Assumption-Guarantee Formulas

Weak assumption-guarantee formulas specify weak assumption-guarantee properties. A weak assumption-guarantee property of a module with assumption A and guarantee G asserts the following:

> For every computation of the module, if A holds for some prefix of the computation, then G also holds for the same prefix.

Notice again that, if a safety property does not hold for a prefix of a computation, then the property will not hold for any longer prefix. The above assertion therefore says that G holds as long as A does.

With A and G given respectively as $\Box H_A$ and $\Box H_G$, the weak assumption-guarantee property can be expressed as $\Box(\boxminus H_A \rightarrow \boxminus H_G)$, which is equivalent

to $\Box(\boxminus H_A \rightarrow H_G)$. Hence, we define weak assumption-guarantee formulas of the form $A \trianglerighteq G$ as follows:

$$A \trianglerighteq G \ (i.e., \Box H_A \trianglerighteq \Box H_G) \ \stackrel{\Delta}{=} \ \Box(\boxminus H_A \rightarrow H_G)$$

Weak assumption-guarantee formulas lack the kind of mutual induction mechanism built into strong assumption-guarantee formulas and cannot be readily composed.

A Quick Comparison between Strong and Weak Assumption-Guarantee Formulas: For a property $\Box H_A \triangleright \Box H_G$ to hold for the computations of a module M, no step of an environment compatible with M should be able to falsify both H_A and H_G. On the other hand, $\Box H_A \trianglerighteq \Box H_G$ does not have this constraint. This distinction is further elaborated in Section 7.

4 Compositional Verification

We present two compositional verification rules: one using strong assumption-guarantee formulas and the other using weak assumption-guarantee formulas.

Theorem 4 (Rule MOD-S). *Suppose that A_i, G_i, and G are canonical safety formulas. Then,*

$$\frac{\begin{array}{l} M_i \models A_i \triangleright G_i \text{ for } 1 \leq i \leq n \\ \models \bigwedge_{i=1}^{n} (A_i \triangleright G_i) \rightarrow G \end{array}}{\|_{i=1}^{n} M_i \models G}$$

The first premise may be established by applying a verification rule for canonical safety formulas from [22, Chapter 4] (recall that $(A_i \triangleright G_i) \equiv \Box(\circleddash \boxminus H_{A_i} \rightarrow H_{G_i})$ is a canonical safety formula), while the second premise may be established by applying the composition rule from Theorem 3 or the simpler Theorem 1. If all the modules are finite, then both of the two premises may be established by a suitable model checker. Nonetheless, Theorem 3 may still be useful for reducing the complexity of checking validity.

Regarding compositional verification using weak assumption-guarantee formulas, Manna and Pnueli have proposed in [22, Page 337] a compositional verification rule where the property of a module is exactly in the form of a weak assumption-guarantee formula. Consider a system S that is equivalent to $\|_{i=1}^{n} M_i$. Translated into our notation, the compositional verification rule reads as follows.

Theorem 5 (Rule MOD-W). *Suppose that S is a system equivalent to $\|_{i=1}^{n} M_i$ and A and G are canonical safety formulas. Then,*

$$\frac{\begin{array}{l} S \models A \\ M_j \models A \trianglerighteq G \text{ for some } j \end{array}}{S \models G}$$

As pointed out in [22], Rule MOD-W is normally applied in an incremental manner. One typically starts with A replaced by *true* and proves some property $\Box H_1$ of a module; one then uses $\Box H_1$ in place of A to prove another property $\Box H_2$ of another module; and so on.

The rule could have been formulated as:

$$\frac{\begin{array}{l} S \models \Box H_A \\ M_j \models \Box H_A \rightarrow \Box H_G \text{ for some } j \end{array}}{S \models \Box H_G}$$

The new rule seems to look simpler. However, to establish $M_j \models \Box H_A \rightarrow \Box H_G$, it is inevitable in practice to face the proof obligation of $M_j \models \Box(\boxminus H_A \rightarrow H_G)$; this is due to the inability of a module to predict the future of its environment. Rule MOD-W makes this clearer.

5 Examples

We consider two examples. The examples are very simple and are intended to contrast the respective strengths of strong and weak assumption-guarantee specifications, demonstrating their complementary roles in compositional verification (rather than their abilities in tackling large systems). In each example, a system is decomposed as the parallel composition of two modules and a property of the system is proven compositionally. We argue that Rule MOD-S is more effective for the first example, while Rule MOD-W is more effective for the second.

5.1 Example 1

Consider again Program KEEP-AHEAD that appeared in Section 2. It is easy to see that the values of a and b are monotonically (but not strictly) increasing in KEEP-AHEAD, i.e., KEEP-AHEAD $\models \Box((a \geq a^-) \wedge (b \geq b^-))$. Can the property be verified compositionally?

Theorem 1 suggests that we decompose $\Box((a \geq a^-) \wedge (b \geq b^-))$ as the conjunction of $\Box(b \geq b^-) \rhd \Box(a \geq a^-)$ and $\Box(a \geq a^-) \rhd \Box(b \geq b^-)$. Unfortunately, neither $M_a \models \Box(b \geq b^-) \rhd \Box(a \geq a^-)$ nor $M_b \models \Box(a \geq a^-) \rhd \Box(b \geq b^-)$. For Module M_a, the assumption $\Box(b \geq b^-)$ says nothing about the initial value of b (permitting b to be an arbitrary negative integer initially), it therefore cannot guarantee that the value of a is monotonically increasing in the very first step (a is 0 initially and may become $b+1$ in the first step). An analogy applies to Module M_b.

A simple remedy is to first strengthen the proof obligation as KEEP-AHEAD $\models$ $\Box((\mathit{first} \rightarrow a \geq 0) \wedge (a \geq a^-) \wedge (\mathit{first} \rightarrow b \geq 0) \wedge (b \geq b^-))$. Again, Theorem 1 suggests that we decompose

$$\Box((\mathit{first} \rightarrow a \geq 0) \wedge (a \geq a^-) \wedge (\mathit{first} \rightarrow b \geq 0) \wedge (b \geq b^-))$$

as the conjunction of

$$\Box((\mathit{first} \rightarrow b \geq 0) \wedge b \geq b^-) \rhd \Box((\mathit{first} \rightarrow a \geq 0) \wedge a \geq a^-)$$

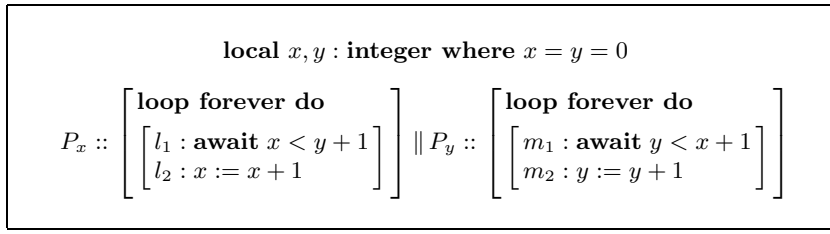

Fig. 3. Program KEEP-UP.

and

$$\Box((\mathit{first} \rightarrow a \geq 0) \wedge a \geq a^-) \rhd \Box((\mathit{first} \rightarrow b \geq 0) \wedge b \geq b^-).$$

It turns out that $M_a \models \Box((\mathit{first} \rightarrow b \geq 0) \wedge b \geq b^-) \rhd \Box((\mathit{first} \rightarrow a \geq 0) \wedge a \geq a^-)$ and $M_b \models \Box((\mathit{first} \rightarrow a \geq 0) \wedge a \geq a^-) \rhd \Box((\mathit{first} \rightarrow b \geq 0) \wedge b \geq b^-)$. Applying Rule MOD-S, we successfully prove KEEP-AHEAD $\models \Box((\mathit{first} \rightarrow a \geq 0) \wedge (a \geq a^-) \wedge (\mathit{first} \rightarrow b \geq 0) \wedge (b \geq b^-))$ and hence KEEP-AHEAD $\models \Box((a \geq a^-) \wedge (b \geq b^-))$ in a compositional way.

It would be inconvenient, if not impossible, to achieve compositionality for the example using Rule MOD-W. With this rule, one somehow has to first establish either KEEP-AHEAD $\models \Box(a \geq a^-)$ or KEEP-AHEAD $\models \Box(b \geq b^-)$. To establish KEEP-AHEAD $\models \Box(a \geq a^-)$ first, for example, one may attempt to prove KEEP-AHEAD $\models \Box(b \geq b^-)$ and $M_a \models \Box(b \geq b^-) \trianglerighteq \Box(a \geq a^-)$. But, KEEP-AHEAD $\models \Box(b \geq b^-)$ then has to be established first, leading to a cycle.

5.2 Example 2

For the second example, we take Program KEEP-UP from [22, Chapter 4], which is recreated in Figure 3. The program is decomposed as the composition of modules M_x and M_y as shown in Figure 4. The system specifications of KEEP-UP, M_x, and M_y are omitted for brevity. Note that in M_x the input variable y is explicitly given an initial value 0 and similarly in M_y the input variable x is given an initial value 0.

It can be shown that KEEP-UP $\models \Box(|x - y| \leq 1)$. In [22], the property $\Box(|x-y| \leq 1)$ is proven compositionally by repeated applications of Rule MOD-W. Note that $\Box(|x - y| \leq 1)$ is equivalent to $\Box((x \leq y + 1) \wedge (y \leq x + 1))$, or $\Box(x \leq y + 1) \wedge \Box(y \leq x + 1)$. Briefly, the compositional verification proceeds as follows:

1. Prove KEEP-UP $\models \Box(x \geq x^-)$ by applying Rule MOD-W with A replaced by *true* and establishing the premise $M_x \models \mathit{true} \trianglerighteq \Box(x \geq x^-)$; prove KEEP-UP $\models \Box(y \geq y^-)$ in an analogous way. These two proofs are independent of each other.
2. Prove KEEP-UP $\models \Box(x \leq y + 1)$ by applying Rule MOD-W with A replaced by $\Box(y \geq y^-)$, which was proven in the previous step, and establishing the

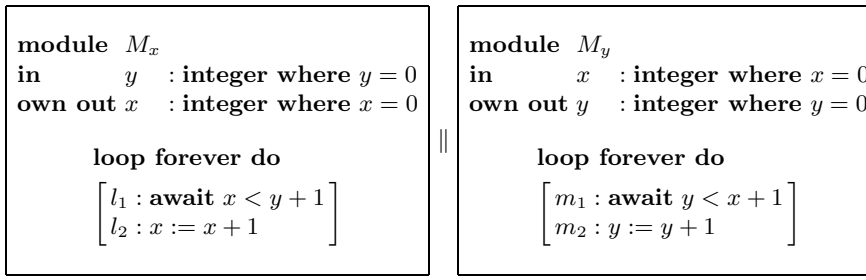

Fig. 4. Program KEEP-UP as the parallel composition of two modules.

premise $M_x \models \Box(y \geq y^-) \trianglerighteq \Box(x \leq y + 1)$; prove KEEP-UP $\models \Box(y \leq x + 1)$ in an analogous way. Again, these two proofs are independent of each other.

An attempt to use Rule MOD-S would fail. The property $\Box(x \leq y+1) \wedge \Box(y \leq x + 1)$ indeed follows from the conjunction of $\Box(x \leq y + 1) \triangleright \Box(y \leq x + 1)$ and $\Box(y \leq x + 1) \triangleright \Box(x \leq y + 1)$ like in the first example. However, it is not possible to establish either $M_x \models \Box(x \leq y + 1) \triangleright \Box(y \leq x + 1)$ or $M_x \models \Box(y \leq x + 1) \triangleright \Box(x \leq y + 1)$. This is due to the fact that both $(x \leq y + 1)$ and $(y \leq x + 1)$ are state formulas that may be falsified by a transition of some environment compatible with M_x (the environment is allowed to change y in an arbitrary way); a module cannot possibly satisfy an assumption-guarantee property if the guarantee part can be falsified by its environment. Analogous arguments applies for M_y.

6 Liveness

To allow liveness properties, we simply strengthen an assumption-guarantee specification by conjoining it with the ordinary implication between the assumption and the guarantee. We consider the extension of a strong assumption-guarantee formula; the extension of a weak one can be done analogously. We will present just one inference rule for composing such properties.

As the generalized definition of a strong assumption-guarantee formula with assumption $A \equiv \Box H_A \wedge L_A$ (in the canonical form) and guarantee $G \equiv \Box H_G \wedge L_G$, we define $A \triangleright G$ as follows:

$$A \triangleright G \stackrel{\Delta}{=} (\Box H_A \triangleright \Box H_G) \wedge (A \rightarrow G)$$

where the $\triangleright$ on the right hand side is as defined in Section 3. This generalized definition is consistent with the definition of $\triangleright$ for safety assumptions and guarantees, since if A and G are safety formulas, the implication $A \rightarrow G$, i.e., $\Box H_A \rightarrow \Box H_G$, will be subsumed by $\Box H_A \triangleright \Box H_G$.

Now that the assumptions may contain liveness properties, we no longer have symmetric composition rules like that in Theorem 2, as mutual dependency

on liveness properties leads to unsound rules. Theorem 2 was generalized in our previous work [16] to permit liveness in the guarantee parts. Below is an asymmetric composition rule for two modules; its proof can be found in the full paper [25].

Theorem 6. *Suppose that* $A_1 \equiv \Box H_{A_1}$, $G_1 \equiv \Box H_{G_1} \wedge L_{G_1}$, $A_2 \equiv \Box H_{A_2} \wedge L_{A_2}$, $G_2 \equiv \Box H_{G_2} \wedge L_{G_2}$, $A \equiv \Box H_A \wedge L_A$, *and* $G \equiv \Box H_G \wedge L_G$, *all in the canonical form. Then,*

$$
\begin{array}{lll}
1.\ (a) & \models & \Box\Big(\boxminus H_A \wedge \boxminus(H_{G_1} \wedge H_{G_2}) \rightarrow H_{A_1} \wedge H_{A_2}\Big) \\
\quad (b) & \models & A \wedge G_1 \rightarrow A_2 \\
2.\ (a) & \models & \Box\Big(\circleddash \boxminus H_A \wedge \boxminus(H_{G_1} \wedge H_{G_2}) \rightarrow H_G\Big) \\
\quad (b) & \models & A \wedge G_1 \wedge G_2 \rightarrow G \\
\hline
 & \models & (A_1 \triangleright G_1) \wedge (A_2 \triangleright G_2) \ \rightarrow \ (A \triangleright G)
\end{array}
$$

Again, take A to be *true* for a closed system.

This composition rule looks unsophisticated and may seem not to be very useful. As a matter of fact, many practical systems exhibit the type of dependency treated by the rule. Take network protocols as an example. An upper-layer protocol relies on the liveness properties of a lower-layer one to ensure liveness in the service that it provides, but the lower-layer protocol does not assume liveness about the upper-layer one. We believe that two modules with more complicated dependency on liveness should be verified as one single module.

7 Discussion: Guidelines of Usage

We give a few guidelines for using the proposed compositional approach.

- *What type of systems can be treated with the compositional approach?*
  Our approach works for a system where each shared variable is owned and can be modified by exactly one of its modules. This is partly due to the fact that only this type of systems allow parallel composition to be conveniently modeled as conjunction in LTL. There certainly are ways to circumvent this limitation; introducing the notion of agents is one possibility [1]. However, we do not think that compositional verification should be applied to two modules that may change a same shared variable. Sharing variables in such a manner indicates that the two modules are tightly coupled and are best treated as one single module.
- *How does one decide which form of assumption-guarantee specification should be used?*
  The desired property of a system gives much hint on what the guarantee parts should look like, as seen from the examples in Section 5. Here is the first thing to check: does the guarantee part involve a variable owned by the environment? For instance, in Example 1 the guarantee part in each of the two assumption-guarantee properties does not involve a variable owned by

the environment. A module M cannot possibly satisfy a property $A \triangleright G$ if some environment compatible with M is capable of falsifying G, which is more likely to happen when G involves a variable owned by the environment. If the environment is capable of falsifying G, then one may try to find a suitable A' such that in falsifying G the environment also has to pay the price of falsifying A'. $A' \trianglerighteq G$ could turn out to be a property of M useful for proving the desired property of the system.

– *What changes are needed to the approach if one prefers using "+"-superscribed (or primed) rather than "−"-superscribed variables in expressing transitions of a system (like in TLA)?*
We have opted for using "−"-superscribed variables, as it leads to a more succinct formulation of assumption-guarantee specifications and rules for composing such specifications. The required changes are quite straightforward. If $A \equiv \mathit{Init}_A \wedge \Box N_A$ and $G \equiv \mathit{Init}_G \wedge \Box N_G$, where N_A and N_G are transition formulas using primed variables, then $A \triangleright G$ translates into

$$\mathit{Init}_G \wedge (\mathit{Init}_A \rightarrow N_G) \wedge \Box(\boxminus((\mathit{first} \rightarrow \mathit{Init}_A) \wedge N_A) \rightarrow \bigcirc N_G).$$

Note that $\Box(\boxminus((\mathit{first} \rightarrow \mathit{Init}_A) \wedge N_A) \rightarrow \bigcirc N_G)$ is a safety formula, though not in the canonical form. The composition rules can be changed accordingly.

We have omitted the treatment of hiding, i.e., assumptions and guarantees with existentially quantified variables, to focus on showing the complementary roles of strong and weak assumption-guarantee specifications. Hiding is a powerful means of expressiveness. The formulation of strong assumption-guarantee specifications with hiding have been considered in our previous work [16]; the same technique applies to weak assumption-guarantee specifications.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] M. Abadi and G.D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.

[4] R. Alur and T.A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In *Computer Aided Verification, Proceedings of the 7th International Conference, LNCS 939*, pages 166–179, 1995.

[5] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. In *Computer Aided Verification, LNCS 818*, pages 324–337, June 1994.

[6] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 35–61. Springer-Verlag, 1984.

[7] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[8] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 353–362, 1989.

[9] P. Collette. Application of the composition principle to Unity-like specifications. In *TAPSOFT '93: Theory and Practice of Software Development, LNCS 668*, pages 230–242. Springer-Verlag, 1993.

[10] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Guarantee Specifications — Application to UNITY.* PhD thesis, Université Catholique de Louvain, June 1994.

[11] W.-P. de Roever, H. Langmåck, and A. Pnueli. *Compositionality: The Significant Difference.* Springer-Verlag, 1998. Lecture Notes in Computer Science 1536.

[12] P. Grønning, T.Q. Nielsen, and H.H. Løvengreen. Refinement and composition of transition-based rely-guarantee specifications with auxiliary variables. In K.V. Nori and C.E. Veni Madhavan, editors, *Foundations of Software Technology and Theoretical Computer Science, LNCS 472*, pages 332–348. Springer-Verlag, 1991.

[13] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[14] C.A.R. Hoare. An axiomatic basis for computer programs. *Communications of the ACM*, 12(8):576–580, 1969.

[15] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[16] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167:47–72, October 1996. An extended abstract appeared earlier in TAPSOFT '95, LNCS 915.

[17] O. Kupferman and M.Y. Vardi. Module checking. In O. Grumberg, editor, *Computer-Aided Verification, CAV '96, LNCS 1102*, pages 75–86. Springer-Verlag, August 1996.

[18] O. Kupferman and M.Y. Vardi. Module checking revisited. In *Computer-Aided Verification, CAV '97, LNCS 1254*. Springer-Verlag, June 1997.

[19] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[20] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[22] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[23] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[24] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1982.

[25] Y.-K. Tsay. Compositional verification in linear-time temporal logic (the full version). Send requests to `tsay@im.ntu.edu.tw`.

[26] M.Y. Vardi. On the complexity of modular model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, pages 101–111, June 1995.

[27] Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory, LNCS 836*, pages 267–282. Springer-Verlag, 1994.