Composing Invariants★

Michel Charpentier

Department of Computer Science,
University of New Hampshire,
Durham, NH
charpov@cs.unh.edu

Abstract. We explore the question of the composition of invariance specifications in a context of formal methods applied to concurrent and reactive systems. Depending on how compositionality is stated and how invariants are defined, invariance specifications may or may not be compositional. This paper examines three classic forms of invariants and their compositional properties. After pointing out what we see as deficiencies of these kinds of invariants, a new fourth form is defined and shown to have useful compositional properties that the more classic forms do not enjoy.

Keywords: formal specification, temporal logic, compositional verification, invariants.

1 Introduction

1.1 Motivation

Compositional reasoning, whether it is looked upon as a problem or as a solution, is receiving much attention in the Formal Methods community. As invariance properties are fundamental in the specification and verification of concurrent and reactive systems, it is natural to consider the question of compositional reasoning when invariants are involved. Are invariants *compositional*? The answer is not as straightforward as it may seem, because there are different definitions of what it means for a specification to be compositional and, maybe more surprisingly, there are different definitions of what it means for a specification to be an invariant.

In this paper, we focus our attention on two possible definitions of being compositional. One corresponds to a widespread intuition of what it means to compose invariants, namely that if all components of a system satisfy the same invariant, then this system also satisfies this invariant. The other form of composition, which might look more surprising at first but is indeed used in various formal notations, is to deduce that an invariant is satisfied by a system as long as it is satisfied by at least one component of that system.

★ *12th International Symposium of Formal Methods Europe (FME'2003)*, September 2003, LNCS 2805, pages 401–421.

Depending on how invariants are defined, they satisfy zero, one or both of these composition rules. We consider four possible definitions of invariants in this paper. Three of them are classic and are already used in various formalisms. One of them is introduced here (although it does have a close relative that appears in [30]). We explore the issue of compositional reasoning for each kind of invariant, and discuss what we see as the benefits of the new kind that is defined in this paper.

The context of this paper is one of reactive systems modeled as transition systems, as in Unity or TLA$^+$. The question of invariants in an object-oriented context is briefly discussed in the conclusion. In this paper, we do not consider specifications other than invariance properties. In particular, we do not discuss the case of liveness and progress specifications. It has been argued that liveness specifications are inherently more difficult to compose than safety specifications. This question is also discussed in the conclusion.

The end of this introductory section is dedicated to presenting a basic set of notations that are used throughout this article. The remainder of the paper is organized as follows. Section 2 introduces two forms of invariants that are commonly defined on transition systems, independently from composition issues. Section 3 discusses the compositional properties of these two forms of invariants. The remaining two forms of invariance specifications (one less classic, one new) are defined in section 4, where their compositionality is also discussed. A simple example, initiated in section 2, is carried throughout the paper to illustrate the four kinds of invariants and their compositional properties. Proofs related to the new form of invariants are given in the full report [8]. Proofs related to classic invariants can be found in the literature [29, 27].

1.2 Notations and Terminology

Generic systems and components are denoted by capitals letters like F and G. In this paper, there are no fundamental differences between systems and components, and both words are used interchangeably, although, at an informal level, *component* conveys the idea that the system is used as a part of a larger system, and *system* conveys the idea that a component is itself composite. Parallel composition is denoted by $\|$.

Specifications are point-wise predicates on systems, i.e., functions from systems to booleans. Function application is denoted by a dot, which is assumed to be left associative and to have higher precedence than boolean connectives. For instance, $\mathsf{inv}.p.F$ is $(\mathsf{inv}.p).F$ and means that specification $\mathsf{inv}.p$ is satisfied by system F.

Following [16], we rely on an "*everywhere*" operator denoted by square brackets. Given a predicate S, $[S]$ means that S is everywhere true, i.e., $S.x$ is true for all x (or $S.x.y$ is true for all x and y, or ..., depending on the type of S). $\neg[\neg S]$ means that S is not everywhere false or, equivalently, that S is satisfiable. We also use the usual quantifiers $\exists$ and $\forall$, which are assumed to have low precedence, as in TLA$^+$ [21]. For instance, $\forall x \in S : p \wedge \exists y : r \Rightarrow q$ is $(\forall x \in S : (p \wedge (\exists y : (r \Rightarrow q))))$.

2 Inductive and Operational Invariants

2.1 Transition Systems

We assume the systems that we reason about are modeled as *transition systems*. Typically, such transition systems are represented as tuples that include, at least, a definition of variables, states, transitions and fairness [23]. Fairness constraints (strong, weak, unconditional, ...) are required to reason about *liveness* (and progress) properties. In this paper, we are focusing our attention on invariance specifications, which belong to the class of *safety* properties. As a consequence, fairness issues do not play any role and can be ignored altogether in our discussion. Moreover, when fairness and liveness are not involved, any nonempty set of transitions can be assimilated to a unique (nondeterministic) transition that encompasses all the transitions from the set. Therefore, in this paper, we rely on (unfair) transition systems defined as 5-tuples. A system F is the tuple $(\mathcal{V}.F, \mathcal{L}.F, \Sigma.F, \mathcal{I}.F, \mathcal{W}.F)$ where:

- $\mathcal{V}$ is a finite set of variables, referred to as *state* variables. They are chosen from a universal vocabulary $\mathcal{D}$.
- $\mathcal{L}$ is a subset of $\mathcal{V}$ of *local* variables, which can be read but not written by other systems.
- Σ is a set of *states*. Each state assigns a value to each variable of $\mathcal{V}$.
- $\mathcal{I}$ is an initial condition that defines the possible initial states of the system. $\mathcal{I}$ is a state predicate (free variables in $\mathcal{V}$) which we assume to be satisfiable.
- $\mathcal{W}$ is a predicate transformer (function from state predicates to state predicates) which represents the transition[1] of the system using a *weakest precondition* semantics. In other words, if q is a state predicate, $\mathcal{W}.F.q$ is the (maximal) set of states that have a successor in q. A WP function, similar to our $\mathcal{W}$, is used in [29].

  We assume that the transition of a transition system allows for stuttering: it is always possible for the next state of a computation to be the same as the current state. Formally, this means that, for any system F and any state predicate q, $[\mathcal{W}.F.q \Rightarrow q]$: if the next state is guaranteed to satisfy q and stuttering is possible, the current state has to satisfy q as well. We also assume that the predicate transformer $\mathcal{W}.F$ is universally conjunctive for any system F, like any weakest (liberal) precondition transformer [16].

2.2 Inductive Invariants

A first kind of invariance specification can be defined directly in terms of a transition system. We call these invariants *transition-based* or *inductive* and

[1] If we were to mention individual transitions instead of a global transition, $\mathcal{W}$ would be defined as the conjunction of the weakest preconditions of all transitions. From this point on, we only refer to the system's transition(s) through $\mathcal{W}$.

denote them with $\mathsf{inv}_{\mathcal{T}}$. We define two types of specifications, next and inv[2]:

$$\begin{aligned} \mathsf{next}_{\mathcal{T}}.(p,q).F &\triangleq [p \Rightarrow \mathcal{W}.F.q] \ , \\ \mathsf{inv}_{\mathcal{T}}.p.F &\triangleq \mathsf{next}_{\mathcal{T}}.(p,p).F \ \wedge \ [\mathcal{I}.F \Rightarrow p] \ . \end{aligned}$$

Informally, $\mathsf{next}_{\mathcal{T}}.(p,q)$ means that whenever a transition is fired from a state that satisfies p, the resulting state satisfies q. Similarly, $\mathsf{inv}_{\mathcal{T}}.p$ specifies that p is true in any initial state and is preserved by every atomic transition. Therefore, by induction, p is true in every state. It should be noted that, since $[\mathcal{W}.F.q \Rightarrow q]$ because of possible stuttering, $\mathsf{next}_{\mathcal{T}}.(p,q).F \Rightarrow [p \Rightarrow q]$.

The approach chosen for this paper is to focus our attention on inv specifications. As we will see in section 3, the part $[\mathcal{I}.F \Rightarrow p]$ in the definition of inv does not involve any difficulty in terms of composition. The next part is where the core of compositional issues turns up. Therefore, although we state most results in terms of inv, our proofs are in terms of next, with inv simply being a special case.

2.3 Operational Invariants

Instead of relying on the definition of a transition system directly, a second kind of invariance specification can be defined in terms of the possible computations of such a system. A transition system F can be associated with a subset $\mathcal{O}.F$ of $(\Sigma.F)^\omega$ of infinite sequences of states defined as follows. An infinite computation $\sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots \rangle$ belongs to the set $\mathcal{O}.F$ if and only if:

1. $\mathcal{I}.F.\sigma_0$,
2. $\forall i \in \mathbb{N} : \mathcal{W}.F.\{\sigma_{i+1}\}.\sigma_i$, where $\{\sigma_{i+1}\}$ is the state predicate that evaluates to *true* for state σ_{i+1} and to *false* for any other state.

Informally, $\mathcal{O}$ consists of those sequences of states that begin with an initial state that satisfies $\mathcal{I}$ and in which each state has a successor in accordance with the transition $\mathcal{W}$. The set $\mathcal{O}$ is nonempty because $\mathcal{I}$ is satisfiable and $\mathcal{W}$ includes stuttering steps.

Once the computations of a transition system are built, next and inv specifications are defined as expected:

$$\begin{aligned} \mathsf{next}_{\mathcal{O}}.(p,q).F &\triangleq \forall \sigma \in \mathcal{O}.F : \forall i \in \mathbb{N} : p.\sigma_i \Rightarrow q.\sigma_{i+1} \ , \\ \mathsf{inv}_{\mathcal{O}}.p.F &\triangleq \forall \sigma \in \mathcal{O}.F : \forall i \in \mathbb{N} : p.\sigma_i \ . \end{aligned}$$

Informally, $\mathsf{next}_{\mathcal{O}}.(p,q)$ means that, in any computation of the system, any state that satisfies p is immediately followed by a state that satisfies q. Although computations include stuttering steps, $\mathsf{next}_{\mathcal{O}}.(p,q)$ does *not* imply that $[p \Rightarrow q]$. In the same way, $\mathsf{inv}_{\mathcal{O}}.p$ means that any state of any computation of a system satisfies p. In linear time temporal logic, $\mathsf{next}_{\mathcal{O}}$ corresponds to $\bigcirc$ and $\mathsf{inv}_{\mathcal{O}}$ corresponds to $\Box$ [23].

Naturally, $\mathsf{next}_{\mathcal{O}}$ and $\mathsf{inv}_{\mathcal{O}}$ are related in a way similar to the relationship between $\mathsf{next}_{\mathcal{T}}$ and $\mathsf{inv}_{\mathcal{T}}$, namely: $\mathsf{inv}_{\mathcal{O}}.p.F \equiv \mathsf{next}_{\mathcal{O}}.(p,p).F \wedge [\mathcal{I}.F \Rightarrow p]$.

[2] In [27], "next" is called "co" and "$\mathsf{next}.(p,p)$" is called "$\mathsf{stable}\ p$".

2.4 Relationship Between Inductive and Operational Invariants

Although they are related, inductive and operational invariants are not equivalent. The use of the word *invariant*, all by itself, without making clear whether inductive or operational invariants are discussed, has been a source of great confusion. A famous example is the case of the UNITY formalism, as it was introduced in [4], in which both $\mathsf{inv}_{\mathcal{T}}$ and $\mathsf{inv}_{\mathcal{O}}$ were used under the same name $\mathsf{invariant}$. This led to inconsistencies that were heavily discussed at the time and solved in various ways, [29] being one of the earliest and cleanest solutions to the problem. Although the relationship between inductive and operational invariants is now well understood, the lack of an agreed upon terminology still makes it difficult to mention invariants without having to resort to stating definitions explicitly, as we have to do in this paper.

In this section, we examine the relationship between inductive and operational invariants in a context of closed systems. Section 3 investigates the question of their relationship when composition is involved.

First, inductive invariants are *stronger* than operational invariants or, equivalently, operational invariants are a consequence of inductive invariants. For any system F and any state predicates p and q:

$$\begin{aligned} \mathsf{next}_{\mathcal{T}}(p,q).F \;&\Rightarrow\; \mathsf{next}_{\mathcal{O}}.(p,q).F \;\;, \\ \mathsf{inv}_{\mathcal{T}}.p.F \;&\Rightarrow\; \mathsf{inv}_{\mathcal{O}}.p.F \;\;. \end{aligned} \tag{1}$$

Along with the following weakening rule (which does not hold for inductive invariants):

$$\mathsf{inv}_{\mathcal{O}}.(p \wedge q).F \Rightarrow \mathsf{inv}_{\mathcal{O}}.p.F \;\;,$$

it provides us with a technique to verify operational invariants on a given transition system. To prove $\mathsf{inv}_{\mathcal{O}}.p.F$, one needs to find a state predicate q such that $\mathsf{inv}_{\mathcal{T}}.(p \wedge q).F$, which itself can be proved directly from $\mathcal{I}.F$ and $\mathcal{W}.F$. From a practical point of view, the difficulty relies in the discovery of predicate q. This technique was known in UNITY as the *substitution axiom* (or in TLA [19] as $\mathsf{INV2}$), but one must keep in mind that the invariant being deduced is operational only.

This proof rule is actually complete [29, 27] in the sense that, when $\mathsf{inv}_{\mathcal{O}}.p.F$ holds, there always exists a predicate q such that $\mathsf{inv}_{\mathcal{T}}.(p \wedge q).F$ is valid:

$$\mathsf{inv}_{\mathcal{O}}.p.F \;\equiv\; \exists q : \mathsf{inv}_{\mathcal{T}}.(p \wedge q).F \;\;. \tag{2}$$

A similar relationship holds between $\mathsf{next}_{\mathcal{O}}$ and $\mathsf{next}_{\mathcal{T}}$:

$$\mathsf{next}_{\mathcal{O}}.(p,q).F \;\equiv\; \exists r : \mathsf{inv}_{\mathcal{T}}.r.F \;\wedge\; \mathsf{next}_{\mathcal{T}}.(p \wedge r, q).F \;\;. \tag{3}$$

From the definition of $\mathsf{inv}_{\mathcal{T}}$ and the fact that $\mathcal{W}.F$ is universally conjunctive, (2) can be reformulated as:

$$\mathsf{inv}_{\mathcal{O}}.p.F \;\equiv\; \mathsf{inv}_{\mathcal{T}}.(p \wedge \forall q : \mathsf{inv}_{\mathcal{T}}.q.F \Rightarrow q).F \;\;.$$

"$\forall q : \mathsf{inv}_{\mathcal{T}}.q.F \Rightarrow q$" is the conjunction of all those state predicates q that are inductively invariant in F. It is itself an inductive invariant of F (conjunctions of inductive invariants are inductive invariants), known as the *strongest invariant* of F [29, 27]. We denote this state predicate by $\mathsf{SI}.F$:

$$[\mathsf{SI}.F \triangleq \forall q : \mathsf{inv}_{\mathcal{T}}.q.F \Rightarrow q] \ . \tag{4}$$

Therefore, (2) can still be rewritten as:

$$\mathsf{inv}_{\mathcal{O}}.p.F \equiv \mathsf{inv}_{\mathcal{T}}.(p \wedge \mathsf{SI}.F) \ . \tag{5}$$

A similar relationship exists between $\mathsf{next}_{\mathcal{O}}$, $\mathsf{next}_{\mathcal{T}}$ and SI:

$$\mathsf{next}_{\mathcal{O}}.(p, q).F \equiv \mathsf{next}_{\mathcal{T}}.(p \wedge \mathsf{SI}.F, q) \ . \tag{6}$$

Because $[\mathsf{SI}.F \Rightarrow p] \equiv [p \wedge \mathsf{SI}.F \equiv \mathsf{SI}.F]$ and $\mathsf{SI}.F$ is the strongest inductive invariant of F, it follows from (5) that:

$$\mathsf{inv}_{\mathcal{O}}.p.F \equiv [\mathsf{SI}.F \Rightarrow p] \ . \tag{7}$$

In other words, p is satisfied by every state of every computation of F if and only if p is a consequence of $\mathsf{SI}.F$. Intuitively, it means that $\mathsf{SI}.F$ characterizes those states that can appear in F's computations, also known as the *reachable states* of F. It should be emphasized that we are referring here to those states that system F can reach *in isolation*, not if F is to become a component of a larger system.

As a final remark, we can observe that, because of (7), $\mathsf{SI}.F$ is also the conjunction of all the operational invariants of F:

$$[\mathsf{SI}.F \equiv \forall q : \mathsf{inv}_{\mathcal{O}}.q.F \Rightarrow q] \ .$$

In section 4.3, we show how the strongest invariant can be redefined to take into account the interaction of a system with its environment, thus leading to a variety of specifications with good compositional properties that are not enjoyed by $\mathsf{inv}_{\mathcal{O}}$ and $\mathsf{next}_{\mathcal{O}}$.

2.5 Example

To illustrate the relationship between inductive and operational invariants, this section introduces a simple example of a system. This system is reused in sections 3.3 and 4.4 when it becomes a component of a larger system.

At this point, we need a syntactic way of describing a transition system, particularly the transition $\mathcal{W}$. The syntax we rely on in this paper is inspired by UNITY. It is by no mean the best choice to represent transition systems. However, it is easy to read and should be directly accessible to a broad body of readers.

Figure 1 represents a transition system for a Door Manager. This system is responsible for opening the doors of an automatic train after the train has

$$\begin{array}{|l|}\hline \mathcal{V} \triangleq \texttt{doors}, \texttt{checkStop}, \texttt{speed} \\ \mathcal{L} \triangleq \texttt{doors}, \texttt{checkStop} \\ \Sigma \triangleq \texttt{doors}, \texttt{checkStop} \in \mathbb{B}; \texttt{speed} \in \mathbb{N} \\ \mathcal{I} \triangleq \neg\texttt{checkStop} \land \texttt{doors} = \texttt{closed} \\ \mathcal{W} \triangleq \texttt{checkStop} := (\texttt{speed} = 0) \\ \quad \shortparallel \ \mathsf{if}\ \texttt{checkStop}\ \mathsf{then}\ \texttt{doors} := \texttt{open} \\ \quad \shortparallel \ \texttt{doors}, \texttt{checkStop} := \texttt{closed}, \texttt{false} \\ \hline \end{array}$$

Fig. 1. System `DoorManager`

stopped, and for closing the doors before the train starts again. It can check the speed of the train but does not modify it. (The system might not open the doors at all, as we are not interested in progress properties here.)

$\mathcal{W}$ is represented by several state-changing statements separated by $\shortparallel$. This operator $\shortparallel$ represents a nondeterministic choice in which a successor to the current state can be obtained by applying any one of these statements. It corresponds to $[\!]$ in UNITY and $\lor$ in TLA. If a statement's guard is false, the next state is identical to the current state. Stuttering transitions are not represented explicitly and the assignment operator := only modifies those state variables that appear on its left-hand side. `open` and `closed` are aliases for *true* and *false*, respectively.

We consider the following invariant:

$$\mathsf{inv}.(\texttt{doors} = \texttt{open} \ \Rightarrow\ \texttt{speed} = 0)\ .$$

Intuitively, this invariant says that, when the doors of a train are open, this train is stopped. If inv is $\mathsf{inv}_{\mathcal{T}}$, this specification is *not* satisfied by `DoorManager`: from a state where $\texttt{checkStop} = \mathit{true}$, $\texttt{speed} = 1$ and $\texttt{doors} = \texttt{closed}$, $\mathcal{W}$ may lead to a state where $\texttt{speed} = 1$ and $\texttt{doors} = \texttt{open}$ (by choosing the second statement), thus falsifying the requirement that $[q \Rightarrow \mathcal{W}.\texttt{DoorManager}.q]$ for inductive invariants q.

No such state, however, can be reached by this system in isolation: it is impossible to have $\texttt{checkStop} = \mathit{true}$ and $\texttt{speed} = 1$ at the same time in a reachable state. In other words, $[\mathsf{SI}.\texttt{DoorManager} \Rightarrow \neg(\texttt{checkStop} = \mathit{true} \land \texttt{speed} = 1)]$. Indeed, the following specification holds for this system:

$$\mathsf{inv}_{\mathcal{T}}.(\ (\texttt{checkStop} = \mathit{true}\ \Rightarrow\ \texttt{speed} = 0) \land (\texttt{doors} = \texttt{open}\ \Rightarrow\ \texttt{speed} = 0)\ )$$

and, as a consequence, the following holds as well:

$$\mathsf{inv}_{\mathcal{O}}.(\texttt{doors} = \texttt{open}\ \Rightarrow\ \texttt{speed} = 0).\texttt{DoorManager}\ . \tag{8}$$

3 Composition of Invariants

3.1 Parallel Composition of Transition Systems

Parallel composition is defined in the usual way, similar to [4, 27, 2, 18]: set union of variables[3] and transitions and conjunction of initial predicates. More precisely, if F and G are two transition systems, $F \| G$ is defined by:

- $\mathcal{V}.(F\|G) \triangleq \mathcal{V}.F \cup \mathcal{V}.G$.
- $\mathcal{L}.(F\|G) \triangleq \mathcal{L}.F \cup \mathcal{L}.G$.
- $\Sigma.(F\|G)$ maps variables from $\mathcal{V}.F$ to values as in $\Sigma.F$, and variables from $\mathcal{V}.G$ to values as in $\Sigma.G$.
- $\mathcal{I}.(F\|G) \triangleq \mathcal{I}.F \wedge \mathcal{I}.G$.
- $\mathcal{W}.(F\|G) \triangleq \mathcal{W}.F \wedge \mathcal{W}.G$.

Some parallel compositions, however, are impossible and do not lead to transition systems, either because $\mathcal{L}.F$ (resp., $\mathcal{L}.G$) and $\mathcal{W}.G$ (resp., $\mathcal{W}.F$) are not compatible (a component would modify another component's local variable) or because $\mathcal{I}.F$ and $\mathcal{I}.G$ are not compatible (no initial state is suitable for both components). We denote by $F\surd G$ the fact that systems F and G can indeed be composed: $F\surd G$ is true exactly when:

1. $\mathcal{I}.F \wedge \mathcal{I}.G$ is satisfiable, i.e., $\neg[\neg(\mathcal{I}.F \wedge \mathcal{I}.G)]$,
2. For any state predicate q such that all free variables of q are in the set $\mathcal{L}.G$, $[q \Rightarrow \mathcal{W}.F.q]$; in other words, $\mathcal{W}.F$ does not involve modifying variables from $\mathcal{L}.G$,
3. For any state predicate q such that all free variables of q are in the set $\mathcal{L}.F$, $[q \Rightarrow \mathcal{W}.G.q]$.

It should be noted that, according to our definition, the local variables of a system can be read by other systems. A more standard definition would make a distinction between true local variables (which are invisible from outside the system) and "read-only" variables (which can be read but not written by other systems). As a consequence, our $\surd$ allows for composite systems that would otherwise be ruled out. The rationale for this choice is that, although it is important not to allow these compositions from a system design point-of-view, it does not influence our formal treatment of composition of specifications. More precisely, there is no specification in our discussion that would be satisfied if external systems were not allowed to read local variables but would not be satisfied if they were. Consequently, with respect to the results presented in this paper, true local variables and "read-only" variables are equivalent.

3.2 Composition of Inductive Invariants

Inductive invariants are compositional in the following way: If two systems F and G satisfy an inductive invariant $\mathsf{inv}_{\mathcal{T}}.p$, their parallel composition $F\|G$ satisfies

[3] We do not consider here the issue of variable renaming.

this inductive invariant as well. This rule also holds for transition-based next specifications. Formally, for any systems F and G such that $F\surd G$ and any state predicates p and q:

$$\mathsf{next}_{\mathcal{T}}.(p,q).F \wedge \mathsf{next}_{\mathcal{T}}.(p,q).G \;\Rightarrow\; \mathsf{next}_{\mathcal{T}}.(p,q).(F\|G) \;, \tag{9}$$

$$\mathsf{inv}_{\mathcal{T}}.p.F \wedge \mathsf{inv}_{\mathcal{T}}.p.G \;\Rightarrow\; \mathsf{inv}_{\mathcal{T}}.p.(F\|G) \;. \tag{10}$$

This is a direct consequence of the definition of inductive invariants and the definition of parallel composition of transition systems:

– From $[\mathcal{I}.(F\|G) \equiv \mathcal{I}.F \wedge \mathcal{I}.G]$, if $[\mathcal{I}.F \Rightarrow p]$ and $[\mathcal{I}.G \Rightarrow p]$, then $[\mathcal{I}.(F\|G) \Rightarrow p]$.
– From $[\mathcal{W}.(F\|G) \equiv \mathcal{W}.F \wedge \mathcal{W}.G]$, if $[p \Rightarrow \mathcal{W}.F.q]$ and $[p \Rightarrow \mathcal{W}.G.q]$, then $[p \Rightarrow \mathcal{W}.(F\|G).q]$.

3.3 Composition of Operational Invariants

Operational invariants, on the other hand, do *not* compose when they are not inductive. There is no equivalent to (9) and (10) for $\mathsf{next}_{\mathcal{O}}$ and $\mathsf{inv}_{\mathcal{O}}$ because these specifications are defined in terms of $\mathcal{O}$, the set of computations of a closed system, and interactions with a possible environment are not taken into account at all in the definition of $\mathcal{O}$.

As an example, let us consider the Engine system of figure 2. This component is responsible for starting and stopping a train. In this simple implementation, a train can accelerate (and, in particular, begin to move) only when its doors are closed. It can decelerate at any time, as long as it is not already stopped. This system satisfies the following specification:

$$\mathsf{inv}_{\mathcal{T}}.(\mathtt{doors} = \mathtt{open} \;\Rightarrow\; \mathtt{speed} = 0).\mathtt{Engine} \;. \tag{11}$$

If operational invariants were compositional, the system $\mathtt{DoorManager}\|\mathtt{Engine}$ would satisfy $\mathsf{inv}_{\mathcal{O}}.(\mathtt{doors} = \mathtt{open} \;\Rightarrow\; \mathtt{speed} = 0)$, since both components satisfy it. Obviously, this is not the case, as $\mathtt{DoorManager}\|\mathtt{Engine}$ admits computations in which a train starts to accelerate *after* variable `checkStop` is set to *true*, thus allowing the composed system to open the doors while the train is already in motion. So, $\mathtt{DoorManager}\|\mathtt{Engine}$ illustrates the case of a system in which all components satisfy a desirable safety specification (a train must be halted for its doors to be open) but the system as a whole does not satisfy this specification.

$$\boxed{\begin{array}{l}
\mathcal{V} \triangleq \mathtt{doors}, \mathtt{speed}\\
\mathcal{L} \triangleq \mathtt{speed}\\
\Sigma \triangleq \mathtt{doors} \in \mathbb{B}; \mathtt{speed} \in \mathbb{N}\\
\mathcal{I} \triangleq \mathtt{speed} = 0\\
\mathcal{W} \triangleq \mathsf{if}\ \mathtt{doors} = \mathtt{closed}\ \mathsf{then}\ \mathtt{speed} := \mathtt{speed} + 1\\
\quad\;\, \| \ \mathsf{if}\ \mathtt{speed} > 0\ \mathsf{then}\ \mathtt{speed} := \mathtt{speed} - 1
\end{array}}$$

Fig. 2. System `Engine`

3.4 Discussion

One possible approach to deal with the difficulty that was illustrated in the previous section is to focus on inductive invariants and ignore operational specifications altogether as soon as composition issues are involved. One might think that, as long as inductive invariants can be composed, there is no need for another form of compositional invariants. Inductive invariants, however, are not well suited for specification. They are a reasonable tool in proofs, thanks to weakening rules such as (1), and they are indeed necessary to verify operational invariants. But they are not high-level enough to be used as logical specifications.

Component specifications have to be high-level enough to substantially abstract from component implementations. This constraint is what makes compositional reasoning such a difficult problem. If specifications are too low-level, there is not much benefit in using compositional techniques. There is an overhead cost in specifying independent components, because a component is always more difficult to verify than a closed system, including a closed systems made of several components [20]. That cost can be balanced by the fact that a verified component is reused in building different systems, and therefore that the effort that went into its verification is reused as well [17, 6]. If component specifications are too low-level, however, the part of the verification effort that is reused is minimal, and the main verification effort is to combine those low-level specifications to deduce a (presumably high-level) system specification. On the other hand, if specifications are too high-level, they hide too much of a component's implementation to guarantee the correctness fo a composed system.

Inductive invariants are too low-level. They do not hide much of a component's implementation, and it does not make sense to have to provide a component's implementation to specify it. Operational invariants, on the other hand, are very high-level. To force an operational invariant on a component still leaves much freedom regarding the ways this component can be implemented. The $\mathtt{Engine}'$ component of figure 5, for instance, relies on a very different implementation than `Engine`. One problem is that operational invariants are indeed too high-level. They hide so much of a component's implementation that they cease to enjoy reasonable compositional properties. In section 4, we present a form of invariants that are higher-level than inductive invariants (and lower-level than operational invariants) but still compose according to a rule identical to (10).

A different strategy has been proposed to deal with composition, which is used in particular in [13, 2, 18]. The idea is that the *system* `DoorManager` and the *component* `DoorManager` are two different things and should be specified using two different transition systems. The difference between these two systems lies in the way stuttering is implemented. When a closed system is considered, stuttering steps are defined to leave *all* variables unchanged. In our `DoorManager` system from figure 1 for instance, there is an implicit stuttering transition equivalent to $\mathtt{doors}, \mathtt{checkStop}, \mathtt{speed} := \mathtt{doors}, \mathtt{checkStop}, \mathtt{speed}$. If `DoorManager` is a *component* of a larger system instead, the stuttering transition becomes $\mathtt{doors}, \mathtt{checkStop}, \mathtt{speed} := \mathtt{doors}, \mathtt{checkStop}, \langle ? \rangle$: the shared variable `speed` is now free to take any value. `DoorManager` can be used as a component only if it is implemented using the second form. If the first form is used, it is a closed system and composition is not possible.

When such an approach is used, a component is associated with a transition system in such a way that the resulting computations are not computations of the component, but instead computations of any system that contains this component. An operational specification is not a specification of the component anymore. It is already a specification of a system that uses this component. Not too surprisingly, many difficulties related to composition then disappear naturally. One could even argue that there is no composition involved, as all component specifications are always referring to the global system that contains these components. In this case, even operational invariants are compositional, since they are defined in terms of the computations of a global system. In formalisms that follow this approach, any specification Spec is compositional in the following way. For any systems F and G such that $F\sqrt{}G$:

$$\mathsf{Spec}.F \;\Rightarrow\; \mathsf{Spec}.(F\|G) \ . \tag{12}$$

In particular, any operational invariant of F is an operational invariant of $F\|G$. It should be emphasized, though, that (12) is of a different form than (9) or (10). In section 4.1, we introduce a terminology that helps make a distinction between these two definitions of "being compositional".

Although there are many interesting specifications for which a composition rule such as (12) makes sense, including *assumption-commitment* specifications [5, 10, 12], there are also cases where they do not fit very well [9]. Even if operational invariants can be composed according to (12), specification (8) cannot be used to specify the desired safety property of `DoorManager` for instance. The reason is that it is impossible to implement a Door Manager that satisfies (8) in a formalism where (12) is valid without requiring that this component also has exclusive control over the speed of the train, which would break down compositionality. In a symmetric way, the `Engine` component cannot guarantee specification (8) as well, since it does not have control of the doors. There are ways of dealing with this problem within notations that rely on (12) for composition, but they involve using constructs that are more complicated than simple invariants ($\xrightarrow{+}$ in [2], $\Rightarrow$ combined with $\ominus$ in [18]). In this paper, we advocate instead the use of a high-level form of invariants that compose according to rules similar to those of low-level inductive invariants.[4]

[4] The rule for composing invariants in [13], translated into this paper's notations, is:

$$\mathsf{inv}.p.F \wedge \mathsf{inv}.q.G \;\Rightarrow\; \mathsf{inv}.(p \wedge q).(F\|G) \ .$$

Although it has the flavor of rule (10) for inductive invariants, it is actually a consequence of (12) (which holds for invariants in [13]) and a simple rule about conjunctions of invariants being invariants. It is as if one feels that invariants *should* compose according to (10), even in formalisms in which all specifications already compose according to (12).

4 High-Level Compositional Invariants

4.1 Existential and Universal Specifications

In this section and in the remainder of the paper, we rely on a terminology that was introduced in [5] to help make a distinction between specifications that compose according to rules similar to (10) and those that compose according to (12). Specifications that follow (12) (for any systems F and G) are called *existential*: an existential specification holds in any system that contains at least one component that satisfies the specification. Specifications that follow rules such as (10) are called *universal*: a universal specification holds in any system in which all components satisfy the specification. Of course, any existential specification is also universal.

According to this terminology, the discussion in the previous section can be summarized as follows:

- Inductive invariants are universal. However, they are not very high-level.
- Operational invariants are higher-level, but they are not universal in general.
- Some formalisms choose to tailor their semantics so that every operational specification becomes existential. In such formalisms, there is no simple way to express universal specifications that are not existential.

As mentioned earlier, it is the next part of the definition of invariants that is interesting with respect to composition. The part $[\mathcal{I}.F \Rightarrow p]$ is existential and does not present any difficulty.

4.2 Specification Transformers

Many important component specifications are neither existential nor universal. A challenge for component and system designers is to derive compositional (existential or universal) specifications from non-compositional ones. To help study this fundamental question, we have defined a collection of predicate transformers that indeed transform a given specification into a corresponding compositional specification [11]. In this paper, we rely on one of these transformers, namely $\mathbb{WE}$, defined in [12].

It can be shown that for any specification Spec, there exists a (unique) weakest existential specification stronger that Spec, which we denote by $\mathbb{WE}.\mathsf{Spec}$. $\mathbb{WE}.\mathsf{Spec}$ characterizes those components that "bring" Spec into any system: if a component satisfies $\mathbb{WE}.\mathsf{Spec}$, then any system that uses this component will satisfy Spec. Conversely, if a component is such that any system that uses it satisfies Spec, this component satisfies $\mathbb{WE}.\mathsf{Spec}$. Given two specifications X and Y, $\mathbb{WE}.(X \Rightarrow Y)$ provides us with a powerful form of *assumption-commitment* specifications [5, 10, 3, 12].

The fact that $\mathbb{WE}.\mathsf{Spec}$ is defined to be the *weakest* existential specification stronger than Spec relates to the discussion of the previous section regarding low-level and high-level specifications. Let us suppose that we are interested in designing a component so that systems that use this component satisfy specification Spec. If Spec is existential, we simply design a component that satisfies Spec. But if Spec is not existential, it is not enough for a component to satisfy Spec. The component must be designed to satisfy a stronger specification, namely $\mathbb{WE}.\mathsf{Spec}$. However, we only want to force on this component what is necessary to make systems that use this component satisfy Spec. In particular, we do not want this specification to include too many details about the component's implementation. We want those details to be hidden to a user of this component, and much freedom given to the implementer of such a component.

For some specifications Spec, there does not exist a weakest universal specification stronger than Spec. As a consequence, there is no transformer $\mathbb{WU}$ for universal composition that achieves what $\mathbb{WE}$ does for existential composition.

4.3 Strongest Invariant for Composition

Operational invariants do not compose existentially or universally. However, the transformer $\mathbb{WE}$ can be applied to $\mathsf{inv}_{\mathcal{O}}$ to obtain a form of invariant that composes existentially:

$$[\mathsf{inv}_{\mathcal{E}}.p \ \triangleq\ \mathbb{WE}.(\mathsf{inv}_{\mathcal{O}}.p)] \ .$$

From the definition of $\mathbb{WE}$, $\mathsf{inv}_{\mathcal{E}}$ is (existentially) compositional and stronger than $\mathsf{inv}_{\mathcal{O}}$. For any state predicate p and any systems F and G such that $F\surd G$:

$$\begin{aligned} \mathsf{inv}_{\mathcal{E}}.p.F \ &\Rightarrow\ \mathsf{inv}_{\mathcal{E}}.p.(F \| G) \ , \\ \mathsf{inv}_{\mathcal{E}}.p.F \ &\Rightarrow\ \mathsf{inv}_{\mathcal{O}}.p.F \ . \end{aligned}$$

Specifications $\mathsf{inv}_{\mathcal{E}}.p$ are actually equivalent to operational invariants in those formalisms where all specifications are existential.

In section 2.4, we defined the strongest invariant of a system to be the conjunction of all (inductive or operational) invariants of that system. In the same way, we define the *strongest invariant for composition* (SIC) of a system F to be the conjunction of all existential invariants of that system:

$$[\mathsf{SIC}.F \ \triangleq\ \forall q : \mathsf{inv}_{\mathcal{E}}.q.F \Rightarrow q] \ . \tag{4'}$$

Because conjunctions of invariants are invariants and the transformer $\mathbb{WE}$ is universally conjunctive [12], it follows that $\mathsf{SIC}.F$ is an existential invariant of F:

$$\mathsf{inv}_{\mathcal{E}}.(\mathsf{SIC}.F).F \ . \tag{13}$$

As a consequence, $\mathsf{SIC}.F$ is an operational[5] invariant of F and $[\mathsf{SI}.F \Rightarrow \mathsf{SIC}.F]$.

In the same way as the strongest invariant is used to characterize operational invariants using (7), SIC can be used to characterize existential invariants. It can be shown that, for any state predicate p and any system F:

$$\mathsf{inv}_{\mathcal{E}}.p.F \ \equiv\ [\mathsf{SIC}.F \Rightarrow p] \ . \tag{7'}$$

[5] As it happens, $\mathsf{SIC}.F$ is actually an inductive invariant of F.

The following property shows that $\mathsf{SIC}.F$ is the union of all the predicates $\mathsf{SI}.(F \| G)$ for all the systems G that are compatible with F:

$$[\mathsf{SIC}.F \;\equiv\; \exists G : (F \surd G) \wedge \mathsf{SI}.(F \| G)] \ . \tag{14}$$

Since SI characterizes the reachable states of a system in isolation, it follows that $\mathsf{SIC}.F$ characterizes those states that can be reached by system F *when F interacts with some compatible environment G*. Equivalently, states outside of $\mathsf{SIC}.F$ *cannot* be reached by systems in which F is a component (Fig. 3).

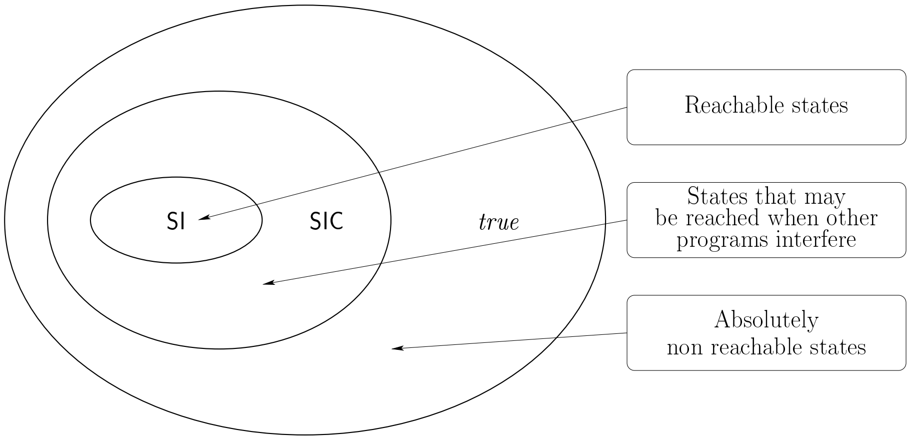

Fig. 3. State reachability

Intuitively, if SIC replaces SI in proof rules (5) and (6) for inv and next, the composition problem illustrated in section 3.3 disappears: no new reachable state can interfere with the proof of an invariant when a system is composed. the following definitions of a new kind of next and inv specifications are based on this idea:

$$\mathsf{next}_{\mathcal{U}}.(p,q).F \;\triangleq\; \mathsf{next}_{\mathcal{T}}.(p \wedge \mathsf{SIC}.F, q).F \ , \tag{6'}$$

$$\mathsf{inv}_{\mathcal{U}}.p.F \;\triangleq\; \mathsf{inv}_{\mathcal{T}}.(p \wedge \mathsf{SIC}.F).F \ . \tag{5'}$$

Because SIC is defined to take the interaction of a system with its environment into account, contrary to SI, $\mathsf{next}_{\mathcal{U}}$ and $\mathsf{inv}_{\mathcal{U}}$ enjoy compositional properties that $\mathsf{next}_{\mathcal{O}}$ and $\mathsf{inv}_{\mathcal{O}}$ do not share. More precisely:

Proposition 1. *For any state predicates p and q, $\mathsf{next}_{\mathcal{U}}.(p,q)$ and $\mathsf{inv}_{\mathcal{U}}.p$ are universal specifications.*

Proposition 1 follows from the fact that $\mathsf{SIC}.(F \| G)$ can be related to $\mathsf{SIC}.F$ and $\mathsf{SIC}.G$ in a simple way:[6]

$$[\mathsf{SIC}.(F \| G) \;\Rightarrow\; \mathsf{SIC}.F \wedge \mathsf{SIC}.G] \ . \tag{15}$$

[6] Implication from right to left does not hold.

No such simple relationship exist between $\mathsf{SI}.(F\|G)$, $\mathsf{SI}.F$ and $\mathsf{SI}.G$. Intuitively, (15) means that any state reachable by a system in which $F\|G$ is a component is also reachable by a system in which F is a component (and a system in which G is a component), which is straightforward. It is not true of the reachable states of closed systems: a state can be reachable in $F\|G$ that was neither reachable in F nor in G.

Because SIC lies between SI and *true* (Fig. 3), $\mathsf{inv}_{\mathcal{U}}$ lies between $\mathsf{inv}_{\mathcal{O}}$ and $\mathsf{inv}_{\mathcal{T}}$. If a system F does not have any local variables, then $\mathsf{SIC}.F$ reduces to *true* (every state is reachable when F is composed) and $\mathsf{inv}_{\mathcal{U}}$ is equivalent to $\mathsf{inv}_{\mathcal{T}}$. But when systems have local variables, $\mathsf{inv}_{\mathcal{U}}$ becomes strictly weaker than $\mathsf{inv}_{\mathcal{T}}$, while still being universal. This possibility, of course, is the interesting case: a component satisfies $\mathsf{inv}_{\mathcal{U}}.p$ (and, therefore, is suitable for composition) but does not satisfy $\mathsf{inv}_{\mathcal{T}}.p$ (and, therefore, would have to be ruled out if $\mathsf{inv}_{\mathcal{T}}$ were used in its specification instead of $\mathsf{inv}_{\mathcal{U}}$). This situation is illustrated in the following section.

We conclude this section with a discussion of how to prove $\mathsf{next}_{\mathcal{U}}$ and $\mathsf{inv}_{\mathcal{U}}$ specifications on a given system. $\mathsf{next}_{\mathcal{U}}$ and $\mathsf{inv}_{\mathcal{U}}$ specifications can be verified without computing SIC explicitly, as $\mathsf{next}_{\mathcal{O}}$ and $\mathsf{inv}_{\mathcal{O}}$ do not require the explicit knowledge of SI. For any system F and any predicates p and q:

$$\mathsf{next}_{\mathcal{U}}.(p,q).F \;\equiv\; \exists r : \mathsf{inv}_{\mathcal{E}}.r.F \;\wedge\; \mathsf{next}_{\mathcal{T}}.(p \wedge r, q).F \;, \tag{3'}$$

$$\mathsf{inv}_{\mathcal{U}}.p.F \;\equiv\; \exists q : \mathsf{inv}_{\mathcal{E}}.q.F \;\wedge\; \mathsf{inv}_{\mathcal{T}}.(p \wedge q).F \;. \tag{2'}$$

Equivalences (3′) and (2′) are direct consequences of the definitions of SIC, $\mathsf{next}_{\mathcal{U}}$ and $\mathsf{inv}_{\mathcal{U}}$. They correspond to (3) and (2) for operational specifications.

Consequently, all that is needed to prove $\mathsf{next}_{\mathcal{U}}$ and $\mathsf{inv}_{\mathcal{U}}$ specifications is a way to prove $\mathsf{inv}_{\mathcal{E}}.p$ on a transition system, which is achieved through the following proposition:

Proposition 2. *Given a system F and a state predicate p, $\mathsf{inv}_{\mathcal{E}}.p.F$ is true if there exists a state predicate q such that:*

1. $\mathsf{inv}_{\mathcal{T}}.q.F$,
2. $[q \Rightarrow p]$,
3. *the free variables of q are a subset of $\mathfrak{L}.F$.*

4.4 Example

The component `DoorManager` from figure 1, which is incorrect, is now replaced with `DoorManager`′ from figure 4. This new component manages a set of lights in addition to opening and closing the doors of the train. More precisely, it turns the lights on as long as the doors are open and turns them off when the doors are closed. `on` and `off` are aliases for *true* and *false*, respectively.

This component does *not* satisfy:

$$\mathsf{inv}_{\mathcal{T}}.(\mathtt{doors} = \mathtt{open} \;\Rightarrow\; \mathtt{speed} = 0) \;. \tag{16}$$

$$
\begin{array}{|l|}
\hline
\mathcal{V} \triangleq \texttt{doors}, \texttt{speed}, \texttt{lights} \\
\mathcal{L} \triangleq \texttt{doors}, \texttt{lights} \\
\Sigma \triangleq \texttt{doors}, \texttt{lights} \in \mathbb{B}; \texttt{speed} \in \mathbb{N} \\
\mathcal{I} \triangleq \texttt{doors} = \texttt{closed} \wedge \texttt{lights} = \texttt{off} \\
\mathcal{W} \triangleq \textsf{if } \texttt{speed} = 0 \vee \texttt{lights} = \texttt{on} \textsf{ then } \texttt{doors}, \texttt{lights} := \neg\texttt{doors}, \neg\texttt{lights} \\
\hline
\end{array}
$$

Fig. 4. System $\texttt{DoorManager}'$

However, it satisfies:

$$\textsf{inv}_{\mathcal{T}}.(\texttt{lights} = \texttt{on} \;\equiv\; \texttt{doors} = \texttt{open}) \;,$$
$$\textsf{inv}_{\mathcal{T}}.(\;(\texttt{lights} = \texttt{on} \;\equiv\; \texttt{doors} = \texttt{open}) \wedge (\texttt{doors} = \texttt{open} \;\Rightarrow\; \texttt{speed} = 0)\;) \;.$$

Because `lights` and `doors` are local to $\texttt{DoorManager}'$, from proposition 2 and ($2'$), it follows that:

$$\textsf{inv}_{\mathcal{U}}.(\texttt{doors} = \texttt{open} \;\Rightarrow\; \texttt{speed} = 0).\texttt{DoorManager}'$$

Since this specification is also satisfied by `Engine` (which satisfies it inductively as (11)), and it is universal from proposition 1, it holds for $\texttt{DoorManager}' \| \texttt{Engine}$ (and, therefore, the weaker operational specification holds as well in the global system).

To complete our train example, we consider the component $\texttt{Engine}'$ as described in figure 5. In this new implementation of the Engine, the speed does not necessarily increase or decrease by one, but varies according to an integer variable `acceleration`. As a consequence, this new component does not satisfy the inductive specification (16). However, it satisfies:

$$\textsf{inv}_{\mathcal{E}}.(\texttt{acceleration} > 0 \;\Rightarrow\; \texttt{speed} > 0).\texttt{Engine}'$$

and, consequently,

$$\textsf{inv}_{\mathcal{U}}.(\texttt{doors} = \texttt{open} \;\Rightarrow\; \texttt{speed} = 0).\texttt{Engine}'$$

from which the correctness of the composed system can be deduced as before.

This example illustrates two important points concerning $\textsf{next}_{\mathcal{U}}$ and $\textsf{inv}_{\mathcal{U}}$ specifications:

- When a component is specified in terms of $\textsf{next}_{\mathcal{U}}$ and $\textsf{inv}_{\mathcal{U}}$, there is a freedom in the way this component can be implemented. This freedom is not available when a component is specified in terms of $\textsf{next}_{\mathcal{T}}$ and $\textsf{inv}_{\mathcal{T}}$, because $\textsf{next}_{\mathcal{T}}$ and $\textsf{inv}_{\mathcal{T}}$ are lower-level than $\textsf{next}_{\mathcal{U}}$ and $\textsf{inv}_{\mathcal{U}}$.
- To prove a $\textsf{next}_{\mathcal{U}}$ or an $\textsf{inv}_{\mathcal{U}}$ specification on a given system is not different in nature than to prove a $\textsf{next}_{\mathcal{O}}$ or an $\textsf{inv}_{\mathcal{O}}$ specification. It reduces to finding a suitable inductive invariant. The benefit is that, if some locality constraints hold for this inductive invariant, a universal specification can be claimed instead of a non-compositional specification.

$$
\begin{aligned}
\mathcal{V} &\triangleq \texttt{doors}, \texttt{speed}, \texttt{acceleration} \\
\mathcal{L} &\triangleq \texttt{speed}, \texttt{acceleration} \\
\Sigma &\triangleq \texttt{doors} \in \mathbb{B}; \texttt{speed}, \texttt{acceleration} \in \mathbb{N} \\
\mathcal{I} &\triangleq \texttt{speed} = 0 \wedge \texttt{acceleration} = 0 \\
\mathcal{W} &\triangleq \textsf{if } \texttt{doors} = \texttt{closed} \textsf{ then } \texttt{speed} := 1 \\
&\;\; \parallel \texttt{speed} := \texttt{speed} + \texttt{acceleration} \\
&\;\; \parallel \textsf{if } \texttt{speed} > \texttt{acceleration} \textsf{ then } \texttt{speed} := \texttt{speed} - \texttt{acceleration} \\
&\;\; \parallel \textsf{if } \texttt{speed} > 0 \textsf{ then } \texttt{acceleration} := \texttt{acceleration} + 1 \\
&\;\; \parallel \textsf{if } \texttt{acceleration} > 0 \textsf{ then } \texttt{acceleration} := \texttt{acceleration} - 1 \\
&\;\; \parallel \textsf{if } \texttt{speed} = 1 \textsf{ then } \texttt{speed}, \texttt{acceleration} := 0, 0
\end{aligned}
$$

Fig. 5. System $\texttt{Engine}'$

5 Summary and Discussion

Compositional reasoning has been defined as the capacity to deduce the correctness of a system from the specifications (as opposed to implementations) of its components [14]. Although it can be argued that there is more to compositional reasoning than just this bottom-up deduction [11, 7], this question is indeed an important issue that must be dealt with. However, for the problem to be fully defined, one needs to add the important additional constraint, which too often is left implicit, that we expect specifications to be of a higher level than implementations. Without this constraint, the problem could be solved easily by choosing a low-level specification language close to implementation languages. After all, CSP processes, UNITY programs and TLA$^+$ formulas are formally defined and compose very naturally. The real issue, however, it to define *high-level* specifications such that the behavior of a system can be analyzed from the specifications of its components. Only in this case is compositional reasoning worthwhile, because substantial verification efforts are embedded in components and *reused* when components are reused.

A recent trend in compositional reasoning is to focus exclusively on existential specifications [1, 13, 30, 2, 18]. Although we acknowledge the importance of existential specifications and have indeed based entire case studies on them [10, 3], we also feel that other forms of composition can lead to better specifications in some cases. Universal specifications, in particular, seem to be especially important [9]. Universal composition is indeed what first comes to mind when composition of invariants is discussed.

In this paper, we have introduced a form of high-level invariant specifications, called $\mathsf{inv}_{\mathcal{U}}$, that are compositional for universal composition. The definition of $\mathsf{inv}_{\mathcal{U}}$ specifications is obtained through a simple modification of the notion of strongest invariant. Proof rules for $\mathsf{inv}_{\mathcal{U}}$ and for the usual high-level (non-compositional) invariants are very similar. There are indeed a number of theorems related to $\mathsf{next}_{\mathcal{U}}$ and $\mathsf{inv}_{\mathcal{U}}$ that are identical to their counterparts for $\mathsf{next}_{\mathcal{O}}$ and $\mathsf{inv}_{\mathcal{O}}$ and that were not mentioned in this paper. To prove an $\mathsf{inv}_{\mathcal{U}}$ specification is not different from proving an $\mathsf{inv}_{\mathcal{O}}$ specification. Actually, one might say that high-level compositional invariants are proved routinely, but that their compositional nature is often lost for lack of a suitable name.

In [13], a form of strongest invariant similar to our SIC is defined. However, it is only used to define a variety of invariants that correspond to our $\mathsf{inv}_{\mathcal{E}}$ (using a rule similar to (7′)). There are no universal invariants. In [30], two forms of invariants are defined that correspond to our $\mathsf{inv}_{\mathcal{E}}$ and $\mathsf{inv}_{\mathcal{U}}$, but not in terms of a strongest invariant. Moreover, the universal form of invariant is defined only as a step in the construction of the existential form, which is the only form used to specify components. Both works are mixing these invariant specifications with complex assumption-commitment rules and, in the case of [30], with refinement rules. It is interesting to note that, in [13, 30], the starting point in the definition of compositional specifications is a transition system and its computations. Our method is different, as transitions and computations are completely absent from our definition of $\mathsf{inv}_{\mathcal{U}}$ (although they are used, obviously, in the definition of $\mathsf{inv}_{\mathcal{T}}$ and $\mathsf{inv}_{\mathcal{O}}$). Our approach to making invariant specifications compositional is purely semantic and relies on a predicate transformer $\mathbb{WE}$ that is defined in a more general context (i.e., independent from transition systems and temporal logic).

The family of specifications $\mathsf{inv}_{\mathcal{E}}$ is defined by direct application of $\mathbb{WE}$ to $\mathsf{inv}_{\mathcal{O}}$. As a result, we know that $\mathsf{inv}_{\mathcal{E}}.p$ is the weakest existential specification stronger than $\mathsf{inv}_{\mathcal{O}}.p$. $\mathsf{inv}_{\mathcal{U}}$, on the other hand, is not the result of applying a transformer $\mathbb{WU}$ to $\mathsf{inv}_{\mathcal{O}}$. The reason is that we do not have such a transformer, because the weakest universal specification stronger than a given specification does not always exist. The absence of a transformer $\mathbb{WU}$ in the definition of $\mathsf{inv}_{\mathcal{U}}$ raises an interesting question: Is $\mathsf{inv}_{\mathcal{U}}.p$ the weakest universal specification stronger than $\mathsf{inv}_{\mathcal{O}}.p$? This question is unanswered. Actually, we do not know whether a weakest universal specification stronger than $\mathsf{inv}_{\mathcal{O}}$ exists at all.

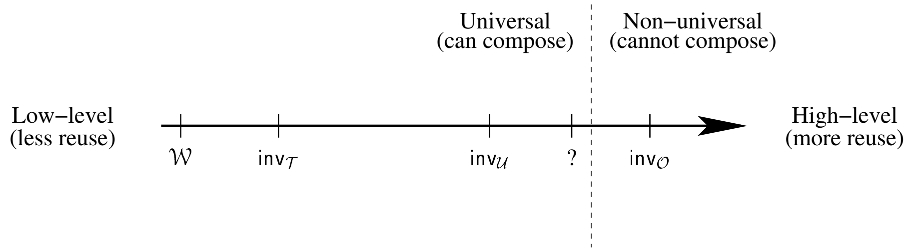

Fig. 6. Weakest universal invariant?

As illustrated in figure 6, $\mathsf{inv}_{\mathcal{U}}$ is indeed weaker than $\mathsf{inv}_{\mathcal{T}}$[7], stronger than $\mathsf{inv}_{\mathcal{O}}$, and encompasses enough information about a component to be universal. What we would like to know, is whether there is a better (weaker, higher-level) form of invariant that is also universal.

[7] It can be shown that it is also weaker than $\mathsf{inv}_{\mathcal{E}}$.

A key idea in compositional reasoning is the notion of non-interference in proofs, as found in the so called Owicki-Gries method [28] and applied to compositional reasoning [25, 31]. Intuitively, new components in the environment of a system should not interfere with the way a specification is proved on that system to ensure that this specification is not compromised. Non-interference with the specification itself is not enough in general. Our definition of $\mathsf{next}_{\mathcal{U}}$ can be seen as a translation, at a semantic level, of this principle. A $\mathsf{next}_{\mathcal{U}}$ specification is proved using an existential invariant $\mathsf{inv}_{\mathcal{E}}$ which is not sensitive to interaction with an external environment. Specifications $\mathsf{inv}_{\mathcal{O}}$ are not compositional because their proof relies on an invariant that can be challenged by the environment.

The approach to building compositional specifications that is presented in this paper applies directly to inv and next specifications. A large class of useful safety specifications can be expressed in terms of next, including stability properties and *unless* (or weak until) specifications. Liveness and progress specifications, on the other hand, have not been considered at all. The argument has been made that liveness specifications are more difficult to compose than safety specifications. However, although leads-to specifications are notoriously difficult to compose [2, 24], a property like transient [26] composes very well (it is existential) and is the basis for the definition of liveness specifications in UNITY. However, transient is lower-level than leads-to. What makes leads-to difficult to compose may be that it is a high-level specification. For instance, the specification of a component responsible for resource allocation might include a property of fair access to resources (expressed as a leads-to). But it could also be specified in terms of a property of absence of deadlock (expressed as an invariant), from which fair access to resources can be derived after composition. The absence of deadlock specification is lower level than the fair access specification: The leads-to specification allows a resource allocator to be implemented through a mechanism that resolves deadlocks *a posteriori* while the invariant-based specification does not. Therefore, the main challenge might well be to compose high-level specifications, whether they are safety or liveness properties.

Our paper also focuses on invariants in the context of reactive systems and temporal logic. Another question is to consider the case of invariants in an object-oriented programming context (class invariants). Both notions of inductive invariants (true before and after each method call, considered individually and atomically) and operational invariants (true of the data fields of an object in between method calls at any point during its lifetime) still exist. They can still be related by a form of strongest invariant that characterizes the reachable states of an object. The argument has been made that the first kind of invariant is preferable because they can be composed more easily [22]. A counter argument is that they are too low-level to be a good specification tool. The laws of composition that are involved there, however, can be more complex than parallel composition (interleaving of atomic transitions). It would be interesting to investigate the generalization of the approach followed in this paper for this new context.

References

1. Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.
2. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
3. K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, January 2002.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
5. K. Mani Chandy and Beverly Sanders. Reasoning about program composition. http://www.cise.ufl.edu/~sanders/pubs/composition.ps.
6. Michel Charpentier. Reasoning about composition: A predicate transformer approach. In *Specification and Verification of Component-Based Systems (SAVCBS'2001)*, pages 42–49. Workshop at OOPSLA'2001, October 2001.
7. Michel Charpentier. An approach to composition motivated by `wp`. In R.D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE'2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, April 2002.
8. Michel Charpentier. Composing invariants. Technical Report 03-10, University of New Hampshire, May 2003.
9. Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In J. Rolim, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1215–1227. Springer-Verlag, April 1999.
10. Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM'99), (Vol. I)*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer-Verlag, September 1999.
11. Michel Charpentier and K. Mani Chandy. Reasoning about composition using property transformers and their conjugates. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics (IFIP-TCS'2000)*, volume 1872 of *Lecture Notes in Computer Science*, pages 580–595. Springer-Verlag, August 2000.
12. Michel Charpentier and K. Mani Chandy. Theorems about composition. In R. Backhouse and J. Nuno Oliveira, editors, *International Conference on Mathematics of Program Construction (MPC'2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 167–186. Springer-Verlag, July 2000.
13. Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to* UNITY. Doctoral thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, June 1994.
14. Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, , and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Cambridge University Press, November 2001.
15. Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference. International Symposium, COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1997.

16. Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Texts and monographs in computer science. Springer-Verlag, 1990.
17. J.L. Fiadeiro and T. Maibaum. Verifying for reuse: foundations of object-oriented system verification. In I. Makie C. Hankin and R. Nagarajan, editors, *Theory and Formal Methods*, pages 235–257. World Scientific Publishing Company, 1995.
18. Bernd Finkbeiner, Zohar Manna, and Henry B. Sipma. Deductive verification of modular systems. In de Roever et al. [15], pages 239–275.
19. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
20. Leslie Lamport. Composition: A way to make proofs harder. In de Roever et al. [15], pages 402–423.
21. Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, July 2002.
22. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual*. Compaq Systems Research Center, October 200.
23. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
24. David Meier and Beverly Sanders. Composing leads-to properties. *Theoretical Computer Science*, 243(1–2):339–361, 2000.
25. Jayadev Misra. The importance of ensuring. In *Notes on Unity*. Department of Computer Science, University of Texas at Austin, 1990. Note 11.
26. Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.
27. Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.
28. Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
29. Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.
30. Rob T. Udink. *Program Refinement in* UNITY*-like Environments*. PhD thesis, Utrecht University, September 1995.
31. Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.