Examples of Program Composition Illustrating the Use of Universal Properties*

Michel Charpentier and K. Mani Chandy

California Institute of Technology
Computer Science Department
m/s 156–80, Pasadena, CA 91125
e-mail; {charpov, mani}@cs.caltech.edu

Abstract. This paper uses a theory of composition based on *existential* and *universal* properties. Universal properties are useful to describe components interactions through shared variables. However, some universal properties do not appear directly in components specifications and they must be constructed to prove the composed system. Coming up with such universal properties often requires creativity. The paper shows through two examples how this construction can be achieved. The principle used is first presented with a toy example and then applied to a more substantial problem.

1 Introduction

A goal of compositional systems development is to support the publication of software modules in a repository such as the Web, where each module is published with its specification, and where new modules can be created by composing existing modules. Hardware vendors publish parts lists with specifications, and other vendors compose these parts to create new parts. Personal computers are manufactured in this fashion. We establish properties of a composed system from the specifications of the components; we do not consider how the components are implemented provided they satisfy their specifications.

Systems can be developed in a compositional way whether the development is bottom-up or top-down or some combination of the two. In all cases, a goal is to specify each component so that the component can be used in a wide variety of environments.

We would like a specification of a software module to name only variables used in that module. We prefer not to specify one module using variables named in other modules with which this module may be composed. The reason for this preference is to allow the widest latitude for the environments of a module. Specifying variables in the environment can over-specify the environment.

A property is a predicate on systems. A specification is a desired property of a system. (Usually, a specification is a desired property which is a conjunction of desired

---

* International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99), April 1999, LNCS 1586, pages 1215–1227.
This work is supported by a grant from the Air Force Office of Scientific Research.

properties.) A local property of a system is a property that names only variables of that system. We would like the specification of a system to be a local property of that system.

When we compose components to get larger systems we may find that, luckily for us, all the properties we desire for the composed system can be obtained in a straightforward way from the specifications of the components. In other cases, we may have to be creative in proving system specifications from their component specifications. This paper is an exploration of how we can prove system properties from local component properties.

This paper uses a theory of composition proposed in [5, 6]. This theory is based on *existential* and *universal* property types. A property type is an existential type when it holds in any system in which at least *one* component has the property. A property is a universal type when it holds in any system in which *all* components have the property. Consider a simple example. Imagine that we are putting pieces together in a jigsaw puzzle. An example of a universal property is "the component is entirely dark colored." If we put entirely dark-colored components together we get entirely dark-colored (larger) components. An example of an existential property is: "the component has a light-colored region." A component has a light-colored region if it has a subcomponent with a light-colored region.

Some properties are neither universal nor existential. The earlier work, however, proposes a theory of composition based on universal and existential properties. Conjunctions of these properties are adequate for specifying most concurrent systems. In particular, existential properties seem to play an important role in the specification of distributed systems [3].

Here, we consider the case of a shared memory system. In such systems, a compositional approach must provide means to describe the way components modify shared variables.

Components specifications do not describe their use of shared variables with existential properties. Specifications about one component must make assumptions on the way other components modify shared variables. One way is to use *universal* properties that specify how all components use shared variables.

However, a universal property of one component referring only to local variables and shared variables of that component and not to other components' local variables will generally not be satisfied by all other components, which modify shared variables according to their local variables. This is because each component has a property defined in terms of its local and shared variables, and so these properties are different for different components.

To cope with this difficulty, one approach is to build, from components specifications, a universal property satisfied by all components, which is then a system property. The paper presents an example of that step.

After presenting the programming model used, we first consider a toy example to highlight the difficulties related to universal properties and the way they can be solved. We then apply the same principles to a more important example: a priority mechanism for conflicting processes.

2 Programming Model

The programming model that we use to illustrate the theory is the model used in [6, 3] which is derived from UNITY [4]. A program consists of a set of typed variables, an *initially* predicate which is a predicate on program states, a finite set C of commands, and a subset D of C of commands subjected to a weak fairness constraint: every command in D must be executed infinitely often. The set C contains at least the command *skip* which leaves the state unchanged.

The program composition is defined to be the union of the sets of variables and the sets C and D of the components, and the conjunction of the *initially* predicates. Such a composition is not always possible. Especially, composition must respect variable locality (a variable declared *local* in a component should not be written by another component) and must provide at least one initial state (the conjunction of initial predicates must be logically consistent). We use $F * G$ to denote that programs F and G can be composed. Then, the system resulting from that composition is denoted by $F[\!]G$.

To specify programs and to reason about their correctness, we use the following properties:

$$\begin{array}{lcl} \texttt{init}\ p & \equiv & \textit{initially} \Rightarrow p \\ \texttt{transient}\ p & \equiv & \langle \exists c : c \in D : p \Rightarrow wp.c.\neg p \rangle \\ p\ \texttt{next}\ q & \equiv & \langle \forall c : c \in C : p \Rightarrow wp.c.q \rangle \\ \texttt{stable}\ p & \equiv & p\ \texttt{next}\ p \\ \texttt{invariant}\ p & \equiv & (\texttt{init}\ p) \wedge (\texttt{stable}\ p) \end{array}$$

We also use the liveness property *leads-to*, denoted by $\mapsto$ and defined by the following rules:

$$\begin{array}{lll} \textit{Transient} & : & [\ \texttt{transient}\ q \ \Rightarrow\ \textit{true} \ \mapsto\ \neg q\ ] \\ \textit{Implication} & : & [p \Rightarrow q] \ \Rightarrow\ [p \ \mapsto\ q] \\ \textit{Disjunction} & : & \textit{For any set of predicates } S\textit{:} \\ & & \quad [\ \langle \forall p : p \in \mathcal{S} : p \ \mapsto\ q \rangle \ \Rightarrow\ \langle \exists p : p \in \mathcal{S} : p \rangle \ \mapsto\ q\ ] \\ \textit{Transitivity} & : & [\ p \ \mapsto\ q \wedge q \ \mapsto\ r \ \Rightarrow\ p \mapsto\ r\ ] \\ \textit{PSP} & : & [\ p \ \mapsto\ q \wedge s\ \texttt{next}\ t \ \Rightarrow\ (p \wedge s) \ \mapsto\ (q \wedge s) \vee (\neg s \wedge t)\ ] \end{array}$$

Note that we use properties with their *inductive* definition and not the definition based on *strongest invariant* [12]. In order to avoid some mix-up, we do not use the *substitution axiom* [4], although we could when dealing with global system properties.

Another element of the theory is the *guarantees* operator, from pairs of properties to properties. Given program properties X and Y, the property X **guarantees** Y is defined by:

$$X\ \textbf{guarantees}\ Y \cdot F \ \triangleq\ \langle \forall G : F * G : (X \cdot F[\!]G) \Rightarrow (Y \cdot F[\!]G) \rangle$$

In this paper, we deal more specifically with universal properties and the *guarantees* operator is not used.

For *existentiality* and *universality*, we use the definition in [3], which is slightly different from the original definition in [6]. For any program property X:

$$\begin{array}{lcl} X \textit{ is existential} & \triangleq & \langle \forall F, G : F * G : X \cdot F \vee X \cdot G \Rightarrow X \cdot F[\!]G \rangle \\ X \textit{ is universal} & \triangleq & \langle \forall F, G : F * G : X \cdot F \wedge X \cdot G \Rightarrow X \cdot F[\!]G \rangle \end{array}$$

Properties of type *init*, *transient* and *guarantees* are existential and properties of type *next*, *stable* and *invariant* are universal. The properties *leads-to* are, in general, neither existential nor universal. However, *leads-to* can appear on the right-hand side of a *guarantees* to obtain existential liveness properties other than *transient* [6, 3].

3 The Toy Example

3.1 Informal Description

We consider a set of components sharing a global counter. Each component also uses a local counter. We are interested in the relationship between the local counters and the global counter.

Components increase a global counter C by one each time they perform a certain action a. Therefore, the value of counter C always equals the total number of actions a that have been performed.

In the remainder of the section, we show what difficulties arise when applying a compositional approach to such a problem, and how to solve them. We specify the behavior previously described at the component level, and the correctness of the global system is derived in a compositional way.

3.2 Component Specification

Each component i has a counter c_i of actions a performed. Therefore, it must increase the global counter C each time it increases its counter c_i. The naive specification, corresponding to the case where the system is composed of *one* component i, is:

- **`init`** $C = c_i \cdot Component_i$
- **`stable`** $C = c_i \cdot Component_i$

If all components share this specification, we have two problems:

- The initial condition of the global system is $\langle \forall i :: C = c_i \rangle$, from which we cannot deduce the desired property that C equals the sum of the counters c_i;
- If c_i is local to component i and component j has to modify the shared variable C, the property **`stable`** $C = c_i$ is not satisfied by component j.

To obtain a compositional proof, we have to do a little more work. Initially, C must equal the sum of all the c_i, but expressing this sum is not local to the component. The only way to know the sum, at the component level, is that all c_i are zero (so that the sum does not depend on the number of components[1]). So, the component can have the following local *init* predicate:

$$\texttt{init}\ c_i = 0 \land C = 0 \cdot Component_i$$

[1] We could have **`init`** $C = c_{i_0}$ for the component i_0 and **`init`** $c_i = 0$ for the others, but this would introduce a dissymmetry.

If all other components have the same condition, the initial state will satisfy the condition that C equals the sum of the c_i.

Now, we need the property that component i will always increase C and c_i by the same value. Formally:

$$\forall k, N :: c_i = k \wedge C = N \ \mathtt{\textbf{next}} \ \langle \exists d : d \geqslant 0 : c_i = k + d \wedge C = N + d \rangle \cdot \mathit{Component}_i$$

This is equivalent to:

$$\forall k, N :: C = c_i + N - k \ \mathtt{\textbf{next}} \ C = c_i + N - k \cdot \mathit{Component}_i$$

and since k and N are universally quantified, this is equivalent to:

$$\forall k :: \mathtt{\textbf{stable}} \ C = c_i + k \cdot \mathit{Component}_i$$

The last thing we must specify to obtain a compositional specification is what variables are local and what variables are shared. This is achieved through a *local* declaration that allows to (syntactically) check what compositions are valid:

$$\mathtt{\textbf{local}} \ c_i \cdot \mathit{Component}_i$$

Only variables not declared *local* can be written by other components (here, the only non local variable is C). From this *local* declarations, we deduce logical properties in a generic way:

$$\mathit{For\ all\ variables\ } v, \mathit{\ other\ than\ } c_i \mathit{\ and\ } C, \ \forall k :: \mathtt{\textbf{stable}} \ v = k \cdot \mathit{Component}_i$$

Finally, at a logical level, the specification of $\mathit{Component}_i$ becomes:

$$\mathtt{\textbf{init}} \ (c_i = 0 \wedge C = 0) \tag{1}$$

$$\forall k :: \mathtt{\textbf{stable}} \ C = c_i + k \tag{2}$$

$$\mathit{For\ all\ variables\ } v, \mathit{\ other\ than\ } c_i \mathit{\ and\ } C, \ \forall k :: \mathtt{\textbf{stable}} \ v = k \tag{3}$$

The set of universal properties are still not shared by other components, but we show, in the next section, how a shared universal property can be deduced from them.

3.3 Correctness Proof

First, from *init* and *local* properties, we observe that:

$$\langle \forall i, j : i \neq j : \mathit{Component}_i * \mathit{Component}_j \rangle$$

Therefore, We can consider a system composed of N components, each component satisfying the previous specification:

$$\mathit{System} \ = \ \langle [\!] i : 0 \leqslant i < N : \mathit{Component}_i \rangle$$

The goal here is to prove global system correctness from the component specifications. This desired property is:

$$\mathtt{\textbf{invariant}} \ C = \sum_{i=0}^{N-1} c_i \cdot \mathit{System}$$

Proof.

$\{$Component specifications, rewriting (2) and (3)$\}$
$\quad$ *For all* i, $\mathtt{\textbf{init}}\ (c_i = 0 \wedge C = 0) \cdot \mathit{Component}_i$
$\wedge$ *For all* $i, \forall k_1, k_2, \cdots, k_n :: \mathtt{\textbf{stable}}\ C = c_i + \sum_{j \neq i} k_j \cdot \mathit{Component}_i$
$\wedge$ *For all* $i, \forall k_1, k_2, \cdots, k_n :: \mathtt{\textbf{stable}}\ \langle \forall j : j \neq i : c_j = k_j \rangle \cdot \mathit{Component}_i$
$\Rightarrow$ $\{$Conjunction of *stable* properties, removing unused dummies$\}$
$\quad$ *For all* i, $\mathtt{\textbf{init}}\ (c_i = 0 \wedge C = 0) \cdot \mathit{Component}_i$
$\wedge$ *For all* i, $\mathtt{\textbf{stable}}\ C = \sum_j c_j \cdot \mathit{Component}_i$
$\Rightarrow$ $\{$*init* properties are existential, *stable* properties are universal$\}$
$\quad$ $\mathtt{\textbf{init}}\ \langle \forall i :: (c_i = 0 \wedge C = 0) \rangle \cdot \mathit{System}$
$\wedge$ $\mathtt{\textbf{stable}}\ C = \sum_j c_j \cdot \mathit{System}$
$\Rightarrow$ $\{$Predicate calculus$\}$
$\quad$ $\mathtt{\textbf{init}}\ C = \sum_j c_j \cdot \mathit{System}$
$\wedge$ $\mathtt{\textbf{stable}}\ C = \sum_j c_j \cdot \mathit{System}$
$\Rightarrow$ $\{$Definition of *invariant*$\}$
$\mathtt{\textbf{invariant}}\ C = \sum_j c_j \cdot \mathit{System}$

$\square$

3.4 Lessons from the Toy Example

The proposal of local properties (1) and (2) of $\mathit{Component}_i$ was obtained from an analysis of the kinds of systems in which we expected to embed the component. In this sense, we took a top-down approach to get a local component specification from an anticipated system specification. However, we can now use our local component specification in a variety of systems including those that we have not anticipated.

4 The Priority Mechanism

4.1 Description

We suppose a set of perpetually conflicting components: Each component always wants to perform an action that requires it to have higher priority than all its neighbors. These conflicts are solved by a *priority mechanism*. Such a mechanism should:

- never give priority at the same time to two conflicting components (8);
- give priority to each component in turn (9).

We uses a principle presented in [4]. We give an orientation to the graph of conflicts so that it always remains acyclic, and we use this graph as a priority graph.

Then, a component should:

- wait until it has priority over its neighbors (4);
- yield priority to its neighbors in finite time after receiving priority (5);
- not introduce cycles in the graph (6).

A way not to introduce cycles is that an active node (with a higher priority than its neighbors in the graph), when changing priorities, always gets a lower priority than *all* its neighbors.

4.2 Component Specification

We call $\mathcal{P}$ the (nonoriented) finite graph of neighborhood. Unless explicitly specified, a graph property *prop* is to be understood as $\mathcal{P}.prop$. The graph $\mathcal{P}$ is described by variables $N[i]$:

$$N[i] \triangleq \textit{set of the neighbors of } Component_i$$

We require that $\langle \forall i :: i \notin N[i] \rangle$ (no node is conflicting with itself). We assume $\langle \forall i, j :: i \in N[j] \equiv j \in N[i] \rangle$ is invariant in the system (implementation of variables $N[i]$).

The graph orientation is defined by the arrow $\rightarrow$. The notation $(i \rightarrow j)$ means that component i has priority over component j. This is a boolean value. It can be modified both by i and j and by no other node. Any change must respect the (implementation) invariant: $\langle \forall i, j : j \in N[i] : (i \rightarrow j) \equiv \neg(j \rightarrow i) \rangle$.

The function $Priority.i$ is used to represent the priority of a node i over all its neighbors:

$$Priority.i \triangleq \langle \forall j : j \in N[i] : (i \rightarrow j) \rangle$$

The three properties of component i become:

$$\forall b, j :: j \in N[i] \wedge (i \rightarrow j) = b \wedge \neg Priority.i \ \mathtt{next}\ (i \rightarrow j) = b \cdot Component_i \quad (4)$$

$$\mathtt{transient}\ Priority.i \cdot Component_i \quad (5)$$

$$Priority.i\ \mathtt{next}\ Priority.i \vee \langle \forall j : j \in N[i] : (j \rightarrow i) \rangle \cdot Component_i \quad (6)$$

As previously, we add a locality constraint: A component cannot change edges other than its incoming and outcoming edges:

$$\forall b, j, j' :: j \neq i \wedge j' \neq i \wedge (j \rightarrow j') = b\ \mathtt{next}\ (j \rightarrow j') = b \cdot Component_i \quad (7)$$

4.3 System Specification

Here, we express formally the system specification previously informally stated:

– safety:

$$\mathtt{invariant}\ \langle \forall i :: Priority.i \Rightarrow \langle \forall j : j \in N[i] : \neg Priority.j \rangle \rangle \cdot System \quad (8)$$

– liveness:

$$\forall i :: true \mapsto Priority.i \cdot System \quad (9)$$

The proof of safety is trivial. To prove the liveness part, we use the fact that the graph always remains acyclic, and therefore that there is always a node which has the priority. To achieve that, we have to build a global universal property, satisfied by all components, from which we can deduce the graph acyclicity. It corresponds to the step presented in the toy example to obtain the property $\mathtt{invariant}\ C = \sum_i c_i$. However, here, the property is more tricky (see property (13)).

4.4 Notations

In order to express this acyclicity, we define the functions[2] $R.i$ and $A.i$:

$$\begin{array}{rcl} R.i & = & \{j : j \in N[i] : (i \to j)\} \\ A.i & = & \{j : j \in N[i] : (j \to i)\} \end{array}$$

and a kind of (nonreflexive) transitive closure $R^*.i$ and $A^*.i$:

$$R^1.i = R.i \qquad \forall n,\ R^{n+1}.i = R^n.i \cup \bigcup_{j \in R^n.i} R.j \qquad R^*.i = \bigcup_{n>0} R^n.i$$

$R^*.i$ is the set of nodes reachable from node i following the graph's edges. $A^*.i$ is defined in the same way and is the set of nodes from which the node i is reachable.
We use the following property for all i and j:

$$[i \in R^*.j \ \equiv\ j \in A^*.i] \tag{10}$$

Then the graph acyclicity is defined by:

$$\begin{array}{rcl} Acyclicity & \triangleq & \langle \forall i :: i \notin R^*.i \rangle \\ & \equiv & \langle \forall i :: i \notin A^*.i \rangle \end{array}$$

We also use the equivalent definition of $Priority.i$:

$$Priority.i \ \equiv\ A^*.i = \emptyset \tag{11}$$

4.5 Construction of a Universal Property

Definition 1. *Let G and G' be two graphs differing only by edge orientation. We say that G' is derived from G through node i_0 if and only if all the edges of i_0 are outcoming in G and incoming in G', all other edges being equal in G and G'.*

$$G \stackrel{i_0}{\leadsto} G' \ \equiv$$
$$\langle \forall k, k' : k, k' \neq i_0 : G.(k \to k') = G'.(k \to k') \rangle \wedge G.A^*.i_0 = \emptyset \wedge G'.R^*.i_0 = \emptyset$$

Lemma 1. *If a graph G' is derived from a graph G through node i_0, then the reachability of nodes in G' cannot be greater than the union of what they are in G and the singleton $\{i_0\}$.*

$$[G \stackrel{i_0}{\leadsto} G' \ \Rightarrow\ \langle \forall i :: G'.R^*.i \subset G.R^*.i \cup \{i_0\} \rangle]$$

Proof. From graph theory. □

Property 12 *The only changes a component i can make in the priority graph are governed by the relation $\stackrel{i}{\leadsto}$.*

$$\forall G :: \mathcal{P} = G \ \texttt{next}\ \mathcal{P} = G \vee G \stackrel{i}{\leadsto} \mathcal{P} \cdot Component_i \tag{12}$$

[2] Defining $R.i$ and $A.i$ as functions instead of relations allows the use of set operators to simplify the writing of some formulas.

Proof. Trivial from the specifications (4), (6) and (7) of component i. □

Property 13 (Universal system property)

$$\forall G :: \mathcal{P} = G \;\mathbf{next}\; \mathcal{P} = G \vee \langle \exists i_0 :: G \overset{i_0}{\leadsto} \mathcal{P} \rangle \cdot \mathit{System} \tag{13}$$

Proof. From (12), the property is satisfied by every component. Since *next* is universal, this is a system property. □

4.6 Proof of the Liveness Property (9)

Property 14 *A component cannot enter any reachability set before it has priority.*

$$\forall i, j :: A^*.i \neq \emptyset \wedge i \notin R^*.j \;\mathbf{next}\; i \notin R^*.j \cdot \mathit{System} \tag{14}$$

Proof. From lemma 1 and (13):

$$\begin{array}{l} \forall G, r, i, j :: \\ \qquad\qquad \mathcal{P} = G \wedge R^*.j = r \wedge A^*.i \neq \emptyset \wedge i \notin r \\ \qquad\qquad\qquad\qquad\quad \mathbf{next} \\ \mathcal{P} = G \vee \langle \exists i_0 :: G \overset{i_0}{\leadsto} \mathcal{P} \wedge R^*.j \subset r \cup \{i_0\} \wedge i \notin r \rangle \cdot \mathit{System} \end{array}$$

If $\mathcal{P} = G$, then $R^*.j = r$ and then $i \notin R^*.j$. If not, from $G \overset{i_0}{\leadsto} \mathcal{P}$, we know that $G.A^*.i_0 = \emptyset$. Therefore, $i_0 \neq i$, and since $i \notin r$, we deduce that $i \notin R^*.i$. Using disjunction over G and r, we obtain (14). □

Property 15 *A component with priority will keep its reachability set or its above set empty.*

$$\forall i :: A^*.i = \emptyset \;\mathbf{next}\; A^*.i = \emptyset \vee R^*.i = \emptyset \cdot \mathit{System} \tag{15}$$

Proof. If a component has priority, its neighbors cannot have priority and, thanks to (4) and (7), cannot change any edge. Therefore, its neighbors cannot set its own priority to *false*. That means that (6) is satisfied by all components. Since it is universal, it is a system property:

$$\forall i :: \mathit{Priority}.i \;\mathbf{next}\; \mathit{Priority}.i \vee \langle \forall j : j \in N[i] :: (j \to i) \rangle \cdot \mathit{System}$$

Then, just rewriting using $R^*.i$ and $A^*.i$, we obtain exactly (15). □

Property 16 *If it is acyclic initially, the priority graph remains acyclic.*

$$\mathit{Acyclicity} \;\mathbf{next}\; \mathit{Acyclicity} \cdot \mathit{System} \tag{16}$$

Proof. From (14), choosing $i = j$, we have:

$$\forall i :: A^*.i \neq \emptyset \wedge i \notin R^*.i \;\mathbf{next}\; i \notin R^*.i \cdot \mathit{System}$$

From (15), using $i \in R^*.i \equiv i \in A^*.i$:

$$\forall i :: A^*.i = \emptyset \;\mathbf{next}\; i \notin R^*.i \cdot \mathit{System}$$

From the disjunction of the two above, strengthening the left-hand side of the *next*:

$$\forall i :: i \notin R^*.i \;\texttt{next}\; i \notin R^*.i \cdot \mathit{System}$$

which, from the definition of $\mathit{Acyclicity}$, is exactly (16). □

Lemma 2. *There is at least one maximal node in any nonempty above set of a finite acyclic graph.*

$$[\mathit{Acyclicity} \Rightarrow \langle \forall i : A^*.i \neq \emptyset : \langle \exists j : j \in A^*.i : A^*.j = \emptyset \rangle\rangle]$$

Proof. From graph theory. □

Property 17 *Any nonpriority component has always a priority component above it.*

$$\forall i :: \texttt{invariant}\; \mathit{Acyclicity} \wedge (A^*.i \neq \emptyset \Rightarrow \langle \exists j : j \in A^*.i : A^*.j = \emptyset \rangle) \cdot \mathit{System} \tag{17}$$

Proof. From lemma 2 and (16). □

Property 18 *Any component with priority eventually escapes every above set.*

$$\forall i, j :: A^*.i = \emptyset \;\mapsto\; i \notin A^*.j \cdot \mathit{System} \tag{18}$$

Proof. From the existential characteristics of (5), we have:

$$\forall i :: \texttt{transient}\; \mathit{Priority}.i \cdot \mathit{System}$$

that rewrites:

$$\forall i :: A^*.i = \emptyset \;\mapsto\; A^*.i \neq \emptyset \cdot \mathit{System}$$

From (15) and the above, using PSP:

$$\forall i :: A^*.i = \emptyset \;\mapsto\; R^*.i = \emptyset \cdot \mathit{System}$$

Since $i \in A^*.j \equiv j \in R^*.i$:

$$\forall i :: A^*.i = \emptyset \;\mapsto\; \langle \forall j :: i \notin A^*.j \rangle \cdot \mathit{System}$$

which is stronger than the required property (18). □

Finally, we prove the liveness correctness (9), which is equivalent to the following property:

Property 19

$$\forall i :: \mathit{true} \;\mapsto\; A^*.i = \emptyset \cdot \mathit{System} \tag{19}$$

Property (14) is equivalent to:

$$\forall i, j :: A^*.i \neq \emptyset \wedge j \notin A^*.i \ \mathbf{next}\ j \notin A^*.i \cdot \mathit{System}$$

which in turn is equivalent to:

$$\forall a, i :: A^*.i = a \neq \emptyset \ \mathbf{next}\ A^*.i \subset a \cdot \mathit{System}$$

In the same way, from (18) we have:

$$\forall a, i, j :: A^*.i = a \wedge j \in A^*.i \wedge A^*.j = \emptyset \ \mapsto\ j \notin a \cdot \mathit{System}$$

Applying PSP to the two above, we obtain:

$$\forall a, i, j :: A^*.i = a \wedge j \in A^*.i \wedge A^*.j = \emptyset \ \mapsto\ A^*.i \subsetneq a \cdot \mathit{System}$$

Using *leads-to* disjunction over j, it becomes:

$$\forall a, i :: A^*.i = a \wedge \langle \exists j : j \in A^*.i : A^*.j = \emptyset \rangle \ \mapsto\ A^*.i \subsetneq a \cdot \mathit{System}$$

From the invariant (17), $A^*.i \neq \emptyset \Rightarrow \langle \exists j : j \in A^*.i : A^*.j = \emptyset \rangle$, and therefore, the previous formula implies:

$$\forall a, i :: A^*.i = a \neq \emptyset \ \mapsto\ A^*.i \subsetneq a \cdot \mathit{System}$$

Through induction on the cardinality of $A^*.i$, this gives the liveness correctness (9).

5 Conclusions

This paper explores a methodology for compositional development of systems. The methodology attempts to work with two types of system properties: universal and existential. A goal of the methodology is to specify components using only local properties. In some cases, system properties can be obtained in a straightforward fashion from local component properties. In other cases, creativity is required to derive system properties from local properties.

This case study exposes the use of three kinds of compositional properties:

- an existential property (5);
- a universal property, shared by all components (6);
- a universal property, not shared by other components (4).

The first two types of property are easy to use: they simply hold in the global system when components are gathered. The third one, however, requires creativity and additional work to become useful in the composition step. The case studies help us in exploring compositional steps that appear to be almost mechanical in contrast to steps that require some ingenuity.

The specification of the conflict resolution solution included the property that the graph of the priority relation is an acyclic graph. We could have specified the components in terms of such acyclic graphs, but this would have resulted in component specifications being nonlocal. The specification of one component would include properties about the priority relationship between completely different components. If we specify components using only local properties, then we have to bridge the gap between local properties and the global system property about acyclic graphs. We found no mechanical way of bridging this gap.

The principle used to build a universal shared property is to weaken the component properties so that all components can share the weakened property. This transformation requires some knowledge on how shared variables are modified by other components. This knowledge is provided by other components (universal) specification.

Note that this step leads to *weaken* a property, and is not exactly a *refinement* step in the strict sense of the word [2, 11]. Such transformations, introducing some nondeterminism, seem to appear frequently when dealing with distributed programs [9, 7].

Universal properties seem to be closely related to global safety. In the priority example, the safety correctness is trivial, but we need a strong safety property to prove the liveness part. In [3], a resource allocator example is derived. In that example, all the safety points are local to components and, actually, the example only makes use of existential properties.

Another point worth being noticed is that, in both the toy example and the priority mechanism example, we only make use of *statement* properties (*transient* or *inductive* safety properties). Properties like "always true" are avoided. The theory provides a *guarantees* operator to deal with non *transient* existential properties. However, for universal properties, nothing more than inductiveness is used.

We are currently investigating such questions, both from the theoretical point of view and by applying the theory of composition to a collection of examples. In particular, we are working on developing a theory based on the traditional rely–guarantee approach [8, 13] and relating it to other theories of composition [10, 1].

The vision that drives us is that of modularity at the level used by manufacturers of personal computers, cars and airplanes. Such systems are complex with large numbers of parts. We should be able to compose certain kinds of software modules in the same way.

Just as there have been many generations of airplanes we now are moving towards many generations of user interfaces, and the compositional technologies that the community has learned in building airplanes over many generations are now being used to build user interfaces and other software systems. The trend towards plug-and-play, object systems, and component systems such as Java Beans and Microsoft's DCOM are examples of steps in this direction.

Formal theories that support compositional development of concurrent systems have been proposed. This work is an exploration of a theory based on specifications using only local properties and two types of properties: universal and existential. We believe that this theory is worthy of further investigation because of the extreme simplicity of its foundation and the successful case studies of its use.

References

1. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
2. R.J.R. Back. Refinement of parallel and reactive programs. Technical report, Marktoberdorf Summer School on Programming Logics, 1992.
3. K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, April 1999. Accepted for publication.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
5. K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.
6. K. Mani Chandy and Beverly A. Sanders. Reasoning about program composition. Technical Report 96-035, University of Florida, Department of Computer and Information Science and Engineering, 1996.
7. Michel Charpentier. A UNITY mapping operator for distributed programs. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Fourth International Symposium of Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 665–684. Springer-Verlag, September 1997.
8. 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.
9. Mamoun Filali, Philippe Mauran, and Gérard Padiou. Raffiner pour répartir. In *Actes des quatrièmes Rencontres francophones du Parallélisme (RenPar'4)*, Villeneuve D'Ascq, France, 1992.
10. S. S. Lam and A. U. Shankar. A theory of interfaces and modules 1: Composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.
11. C. Morgan, P. Gardiner, K. Robinson, and T. Vickers. *On the Refinement Calculus*. FACIT. Springer-Verlag, 1994.
12. Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.
13. Rob T. Udink. *Program Refinement in* UNITY*-like Environments*. PhD thesis, Utrecht University, September 1995.