Towards a Compositional Approach to the Design and Verification of Distributed Systems[⋆]

Michel Charpentier and K. Mani Chandy

California Institute of Technology
Computer Science Department
m/s 256-80, Pasadena, CA 91125
e-mail: {charpov,mani}@cs.caltech.edu

Technical Report: CS-TR-99-02

Abstract. We are investigating a component-based approach for formal design of distributed systems. In this paper, we introduce the framework we use for specification, composition and communication and we apply it to an example that highlights the different aspects of a compositional design, including top-down and bottom-up phases, proofs of composition, refinement proofs, proofs of program texts, and component reuse.

Key-words: component-based design, distributed systems, formal specification, formal verification, temporal logic, UNITY.

1 A Compositional Approach

1.1 Introduction

Component technology is becoming increasingly popular. Microsoft's COM, JavaSoft's beans, CORBA, and new trade magazines devoted to component technology attest to the growing importance of this area. Component-based software development is having an impact in the development of user interfaces. Such systems often have multiple threads (loci of control) executing in different components that are synchronized with each other.

These systems are examples of reactive systems in which components interact with their environments. Component technology has advantages for reactive systems, but it also poses important challenges including the following.

– How do we specify components? Specifications must deal with both progress and safety, and they must capture the relationship between each component and its environment. What technologies will support large repositories of software components, possibly even world-wide webs of components, such that component implementations can be discovered given component specifications?

⋆ This work is supported by a grant from the Air Force Office of Scientific Research.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **1999**	2. REPORT TYPE	3. DATES COVERED **00-00-1999 to 00-00-1999**

4. TITLE AND SUBTITLE **Towards a Compositional Approach to the Design and Verification of Distributed Systems**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Air Force Office of Scientific Research,875 North Randolph Street Suite 325,Arlington,VA,22203-1768**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**29**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

- Electronic circuit design is an often cited metaphor for building software systems using component technologies. Phrases such as *plug and play*, and *wiring components together*, are used in software design. These approaches to software design will work only if there are systematic methods of proving specifications of composed systems from specifications of components.

  Further, we would like to propose methods in which the proof obligations for the designer who puts components together is made easier at the expense of the component designer. The idea is that component designers add component specifications, implementations, and proofs into a repository. An implementation of a component may, in turn, be a composition of other components. We want the composer's work to become easier by exploiting the effort in specification, implementation and proof that is invested in building the component repository.
- Mechanical proof checkers and theorem provers can play an important role in building high-confidence repositories of software components, though the widespread use of these technologies may be decades away. The challenge is to develop theories of composition that can be supported by mechanical provers.

1.2 Proposition

The basis for our framework is the UNITY formalism which provides a way to describe fair transition systems and which uses a small set of temporal logic operators which appear well suited for many applications. It is extended with a theory of composition which relies on two intuitive forms of interaction: *existential* (a system property holds if it holds in at least one component) and *universal* (a system property holds if it holds in all components). We also add a new temporal operator (*follows*) which allows us to represent asynchronous point-to-point communication at a logical level, while using only monotonic distributed (i.e. writable by one component only) variables.

These choices restrict the expressive power of the framework: we do not deal with *any* temporal specification, *any* form of composition, *any* type of communication. This is a way in which we obtain something manageable. However, we need to know if such a framework can be applied to a wide class of problems and if it leads to simple and intuitive proofs. These two points can be explored by using the framework to design several distributed systems.

Through the example described in this paper, we show that this framework can be used to specify generic and specific components, to describe communication between them, to handle refinement proofs (classical UNITY proofs) related to top-down steps, program text correctness proofs ("almost" classical UNITY proofs) and compositional proofs related to bottom-up steps.

The remainder of the paper is organized as follows. In the next section, we formally define the different aspects of the framework we are using. The following section presents the architecture of a resource allocation example and introduces the required formal steps we have to complete in order to achieve the design.

The next sections are each devoted to a part of this design, namely the high-level decomposition, the clients design, the decomposition of the allocator into a simpler allocator and finally the design of this simpler allocator. One proof of each kind (refinement, composition and program text correctness) is given in the paper. All other proofs are detailed in appendixes.

2 Framework

The framework we use is based on a UNITY-like logic and programming notation [1,6]. The traditional UNITY form of composition (*union*) is used. However, at the logical level, we use the notions of *existential* and *universal* properties. Especially, a *guarantees* operator, that provides existential specifications, is introduced [2,3]. We further extend UNITY with an abstraction of communication, based on temporal operators described in [4,9].

2.1 Basis

A program (describing a component behavior) consists of a set of typed variables, an *initially* predicate, which is a predicate on program states, a finite set of atomic commands C, 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.

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 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 UNITY logical operators as defined in [6]. However, when dealing with composition, one must be careful wether to use the *strong* or the *weak* form of these operators. The strong (sometimes called *inductive*) form is the one obtained from the *wp* quantification *without* using the substitution axiom [1,6]. The weak form is either the one obtained when using the substitution axiom or the one defined from the *strongest invariant* (which are equivalent) [6,8]. Strong operators are subscript with $_s$ and weak operators are subscript with $_w$. No subscript means that either form can be used. The strong form of an operator is logically stronger than the weak form. Note that *transient* has no weak form and *leads-to* has no strong form, and that *always* is the weak form of *invariant* (which is strong). These operators are defined as follows, for any state predicates p and q (SI denotes the strongest invariant of the program):

$$\begin{array}{lcl} \texttt{init}\ p & \triangleq & [\mathit{initially} \Rightarrow p] \\ \texttt{transient}\ p & \triangleq & \langle \exists c : c \in D : p \Rightarrow \mathit{wp}.c.\neg p \rangle \end{array}$$

$$\begin{array}{lcl}
p\ \texttt{next}_S\ q & \triangleq & \langle \forall c : c \in D : p \Rightarrow wp.c.q \rangle \\
\texttt{stable}_S\ p & \triangleq & p\ \texttt{next}_S\ p \\
\texttt{invariant}\ p & \triangleq & (\texttt{init}\ p) \wedge (\texttt{stable}_S\ p) \\
p\ \texttt{next}_W\ q & \triangleq & (SI \wedge p)\ \texttt{next}_S\ q \\
\texttt{stable}_W\ p & \triangleq & p\ \texttt{next}_W\ p \\
\texttt{always}\ p & \triangleq & [SI \Rightarrow p]
\end{array}$$

The *leads-to* operator, denoted by $\mapsto$, is the strongest solution of:

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

$X \cdot F$ means that property X holds in program F. Traditionally, monotonicity is expressed with a set of *stable* properties. In order to avoid the repetition of this set of *stable* properties, we define the shortcut:

$$x \nearrow_{\leqslant} \cdot F \ \triangleq\ (\forall k :: \texttt{stable}_W\ k \leqslant x \cdot F)$$

which means that x never decreases in F in isolation (the weak form of *stable* is used and nothing is said about x when F is composed with other components). When there is no ambiguity, we omit the order relation and simply write $x \nearrow$.

Since we are dealing with distributed systems, we assume no variable can be written by more than one component[1]. We consider three kinds of variables: input ports, output ports and local variables. Input ports can only be read by the component; output ports and local variables can be written by the component and by no other component.

The fact that an output port or a local variable cannot be written by another component is referred to as the *locality* principle. Formally, if variable v is writable by component F and *env* is a possible environment of F ($F * \textit{env}$), then:

$$\forall k :: \texttt{stable}_S\ v = k \cdot \textit{env}$$

i.e., v is left unchanged by F's environment.

2.2 Composition

We use a compositional approach introduced in [2,3], based on *existential* and *universal* characteristics. A property is existential when it holds in any system in which at least *one* component has the property. A property is universal when

[1] Strictly speaking, a component local variable may be read by another component, but we do not use that possibility.

it holds in any system in which *all* components have the property. Of course, any existential property is also universal. We use the formal definition of [5] which is slightly different from the original definition in [3]:

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

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:

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

Properties of type *init*, *transient* and *guarantees* are existential and properties of type $next_S$, $stable_S$ and *invariant* are universal. All other types are neither existential nor universal, but can appear on the right-hand side of a *guarantees* to provide an existential property.

2.3 Communication

All the communication involved in a system is described with input and output ports. Formally, an input (resp. output) port is the *history* of all messages received (resp. sent) through this port. Note that ports are monotonic with respect to the prefix relation. We need to introduce a temporal operator to describe communication delays between these ports, as well as some notations to handle easily finite sequences of messages.

Follows. To represent the unbounded nondeterministic delay introduced by some components (including the underlying network), we use a *follows* temporal operator inspired form [4, 9].

Definition 1 ($\underline{\boxed{\diamond}}$)**.** *For any pair of* state expressions *(in particular variables) x and $‘x$, and an order relation $\leqslant$, we define $‘x \mathrel{\underline{\boxed{\diamond}}} x$ (“ $‘x$* follows x*”) with respect to $\leqslant$:*

$$‘x \mathrel{\underline{\boxed{\diamond}}} x \triangleq (x \nearrow_{\leqslant}) \wedge (‘x \nearrow_{\leqslant}) \wedge (\texttt{always } ‘x \leqslant x) \ \wedge \ (\forall k :: k \leqslant x \mapsto k \leqslant ‘x)$$

Intuitively, $‘x \mathrel{\underline{\boxed{\diamond}}} x$ means that x and $‘x$ are monotonic and $‘x$ is always trailing x (wrt the order $\leqslant$), but that some liveness always works at reducing the gap.

Then, *follows* can be used in conjunction with functions on histories to describe different kinds of transformational components. Deterministic components are described with specifications of the form “$Out \mathrel{\underline{\boxed{\diamond}}} f.In$”, while nondeterministic components use the “$f.Out \mathrel{\underline{\boxed{\diamond}}} In$” form. In particular, network components (wires) are specified by “$Out \mathrel{\underline{\boxed{\diamond}}} In$.”

The following properties are referred to as “follows theorems”:

$$\begin{array}{ll} ‘x \mathrel{\underline{\boxed{\diamond}}} x \wedge f \textit{ monotonic} & \Rightarrow f.‘x \mathrel{\underline{\boxed{\diamond}}} f.x \\ “x \mathrel{\underline{\boxed{\diamond}}} ‘x \wedge ‘x \mathrel{\underline{\boxed{\diamond}}} x & \Rightarrow “x \mathrel{\underline{\boxed{\diamond}}} x \\ ‘x \mathrel{\underline{\boxed{\diamond}}} x \wedge ‘y \mathrel{\underline{\boxed{\diamond}}} y & \Rightarrow ‘x \cup ‘y \mathrel{\underline{\boxed{\diamond}}} x \cup y \quad \text{(for sets or bags)} \end{array}$$

Notations on Histories.

Definition 2 **(/).** *Given a (finite) sequence Seq and a set S, Seq/S represents the subsequence of Seq for indexes in S:*

$$|Seq/S| \triangleq \mathrm{card}([1..|Seq|] \cap S) \text{ and}$$
$$\langle \forall k : 1 \leqslant k \leqslant |Seq/S| : (Seq/S)[k] \triangleq Seq[\langle \min n : \mathrm{card}(S \cap [1..n]) \geqslant k : n \rangle] \rangle$$

$|Seq|$ denotes the length of sequence Seq. Note that we do not force values in S to be valid indexes of Seq. Actually, the condition under which k is a valid index of Seq/S is $1 \leqslant k \leqslant |Seq| \;\wedge\; k \in S$.

Definition 3 **($\sqsubseteq_{\mathcal{R}}$).** *Given a binary relation $\mathcal{R}$, we define the corresponding weak prefix relationship, denoted by $\sqsubseteq_{\mathcal{R}}$:*

$$Q \sqsubseteq_{\mathcal{R}} Q' \triangleq |Q| \leqslant |Q'| \wedge \langle \forall k : 1 \leqslant k \leqslant |Q| : Q[k] \; \mathcal{R} \; Q'[k] \rangle$$

Note that $\sqsubseteq_{=}$ is the traditional prefix relationship. In the paper, we are only using $\sqsubseteq_{\leqslant}$ and $\sqsubseteq_{\geqslant}$ on sequences of integers.

Definition 4 **($\leqslant$).**
Given S and S' in $2^{\mathbb{N}}$:

$$S \leqslant S' \triangleq S \subset S' \wedge \langle \forall x : x \in S' \wedge x \notin S : \langle \forall y : y \in S : y \leqslant x \rangle \rangle$$

$S \leqslant S'$ strengthens the subset relation $S \subset S'$ by forcing the additional values in S' to be greater than all values in S. An alternative definition could be:

$$S \leqslant S' \triangleq \langle \forall Q :: Q/S \sqsubseteq Q/S' \rangle$$

Definition 5 **($\mathcal{B}$).** *We denote by $\mathcal{B}(Q)$ the* bag *of the values in sequence Q.*

Note that function $\mathcal{B}$ is monotonic.

3 The Resource Allocation Example

3.1 The Different Steps of the Design

We suppose we want to design a resource allocation system: we want some clients to handle correctly some shared resources.

In a first step, we specify formally what the clients are doing with respect to these resources (spec. [3]) and what the correctness constraints of the system are (spec. [2]). We deduce, in a systematical way, how a resource allocator should behave to provide that correctness (spec. [4]). We then pick some generic network specification (spec. [5]) and make a compositional proof to show that if all components satisfy their specifications, the system global correctness is guaranteed (proof C1 sect. 4.2).

Now, we have to design a resource allocator satisfying the previous specification. We come with the idea that such an allocator can be built from a simpler single-client allocator and some generic components (possibly found in a component library). So, we specify how the single-client allocator should behave (spec. [10]), pick a generic merge component (spec. [6]), a generic distributor component (spec. [7]) and connect all these components with a network (spec. [8]). We obtain an allocator that enjoys additional properties, compared to the allocator we need for our system. Since such properties may be reused later in another design, we specify formally this resulting allocator (spec. [9]) and prove that it is actually obtained from the chosen components (proof C2 sect. B.2). We also prove that this allocator implements the allocator we needed (proof R1 sect. 5.3).

To complete the development, we have to design a client program and a single-client allocator program. Starting from the specifications we obtained ([3] and [10]), we write two programs we hope to satisfy the given specifications. We observe that the resulting programs (prog. sect. 6.1 and prog. sect. 7.1) have more properties than requested. Again, since such properties can be reused, we express formally these behaviors (spec. [11] and spec. [12]), prove that the texts satisfy these specifications (proofs T1 sect. 6.3 and T2 sect. D.3) and, of course, that these specifications are stronger than requested (proofs R2 sect. C.2 and R3 sect. C.3).

The different steps of this design are summarized in fig. 1.

3.2 Notations

Resources are described with anonymous tokens. All the messages exchanged between the resource allocator and its clients have the same type: integer (the number of tokens requested, given or released). $\mathit{Tokens}.h$ is the total number of tokens in history h (i.e. $\mathit{Tokens}.h = \sum_{k=1}^{|h|} h[k]$).

All clients have the same generic behavior. They send requests for resources through a port *ask*, receive these resources through a port *giv*, and release the resources through a port *rel*. Clients variables are prefixed with Client_i. The allocator has three arrays of ports, *ask*, *giv* and *rel*, prefixed by *Alloc*. A network is responsible for transporting messages from $\mathit{Client}_i.\mathit{ask}$ to $\mathit{Alloc}.\mathit{ask}_i$, from $\mathit{Alloc}.\mathit{giv}_i$ to $\mathit{Client}_i.\mathit{giv}$, and from $\mathit{Client}_i.\mathit{rel}$ to $\mathit{Alloc}.\mathit{rel}_i$. A valid initial state is a state where all ports histories are empty and where the resource allocator has a stock of NbT tokens.

4 From a Resource Allocation System towards an Allocator and Clients

4.1 Components Specifications

The global correctness we want for the resource allocation system is expressed in a very traditional way. A safety property states that clients never share more

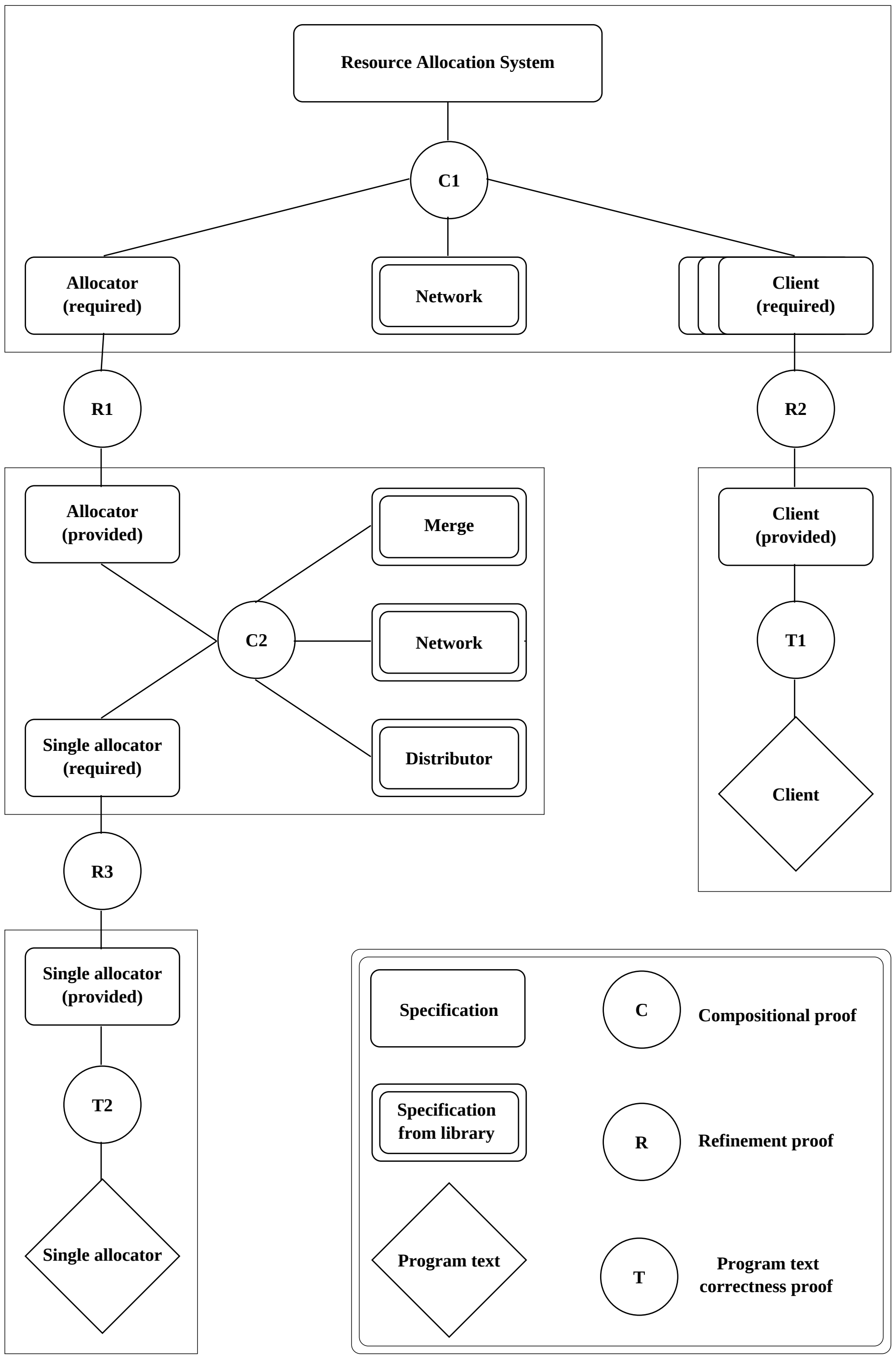


Fig. 1. General design.

tokens that there exists in the system. A liveness property guarantees that all client requests are eventually satisfied (Fig. 2).

$$\texttt{always}\ \sum_i (\mathit{Tokens.Client_i.giv} - \mathit{Tokens.Client_i.rel}) \leqslant \mathit{NbT} \qquad (1)$$

$$\forall i, h :: h \sqsubseteq \mathit{Client_i.ask} \mapsto h \sqsubseteq_{\leqslant} \mathit{Client_i.giv} \qquad (2)$$

Fig. 2. Resource allocation system specification.

Clients specification is also very intuitive. The safety part is that clients never ask for more tokens that there exist (such requests would not be satisfiable). The liveness part guarantees that clients return all the tokens they get, when these tokens are satisfying a request (unrequested tokens or tokens in insufficient number may not be returned) (Fig. 3).

$$\mathit{true}\ \textbf{guarantees}\ \mathit{ask} \nearrow\ \wedge\ \mathit{rel} \nearrow \qquad (3)$$

$$\mathit{true}\ \textbf{guarantees}\ \texttt{always}\ \langle \forall k :: \mathit{ask}[k] \leqslant \mathit{NbT} \rangle \qquad (4)$$

$$\mathit{giv} \nearrow\ \textbf{guarantees}\ \forall h :: h \sqsubseteq \mathit{giv} \wedge h \sqsubseteq_{\geqslant} \mathit{ask} \mapsto \mathit{Tokens.rel} \geqslant \mathit{Tokens.h} \qquad (5)$$

Fig. 3. Client specification (required).

The allocator specification is almost derived from the client specification. In particular, there is a strong correspondence between the right-hand sides of clients *guarantees* and the left-hand side of the allocator *guarantees*. The global safety, which is the responsibility (mostly) of the allocator appears also in the specification (Fig. 4).

$$\mathit{true}\ \textbf{guarantees}\ \forall i :: \mathit{giv}_i \nearrow \qquad (6)$$

$$(\forall i :: \mathit{rel}_i \nearrow)\ \textbf{guarantees}\ \texttt{always}\ \sum_i (\mathit{Tokens.giv}_i - \mathit{Tokens.rel}_i) \leqslant \mathit{NbT} \qquad (7)$$

$$\begin{array}{c} \forall i :: \mathit{ask}_i \nearrow\ \wedge\ \mathit{rel}_i \nearrow \\ \wedge\quad \texttt{always}\ \langle \forall i, k :: \mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\ \wedge\ \forall i, h :: h_i \sqsubseteq \mathit{giv}_i \wedge h_i \sqsubseteq_{\geqslant} \mathit{ask}_i \mapsto \mathit{Tokens.rel}_i \geqslant \mathit{Tokens.h}_i \\ \textbf{guarantees} \\ \forall i, h :: h \sqsubseteq \mathit{ask}_i \mapsto h \sqsubseteq_{\leqslant} \mathit{giv}_i \end{array} \qquad (8)$$

Fig. 4. Allocator specification (required).

The network specification relies on the *follows* operator. Output ports are connected to corresponding input ports with $\trianglelefteq$ which provides both safety and liveness (Fig. 5).

$$\begin{array}{lll} & \forall i :: & \\ \mathit{Client}_i.\mathit{ask} \nearrow & \textbf{guarantees} & \mathit{Alloc}.\mathit{ask}_i \trianglelefteq \mathit{Client}_i.\mathit{ask} \\ \mathit{Alloc}.\mathit{giv}_i \nearrow & \textbf{guarantees} & \mathit{Client}_i.\mathit{giv} \trianglelefteq \mathit{Alloc}.\mathit{giv}_i \\ \mathit{Client}_i.\mathit{rel} \nearrow & \textbf{guarantees} & \mathit{Alloc}.\mathit{rel}_i \trianglelefteq \mathit{Client}_i.\mathit{rel} \end{array} \tag{9}$$

Fig. 5. Network specification.

4.2 Composition Proof

Property (1).

Proof. In resource allocation system:

$\{$Specification (3)$\}$
$\forall i :: \mathit{Client}_i.\mathit{rel} \nearrow$
$\Rightarrow \{$Specification (9), follows definition$\}$
$\forall i :: \mathit{Alloc}.\mathit{rel}_i \nearrow$
$\Rightarrow \{$Specification (7)$\}$
$\texttt{always}\ \sum_i(\mathit{Tokens}.\mathit{Alloc}.\mathit{giv}_i - \mathit{Tokens}.\mathit{Alloc}.\mathit{rel}_i) \leqslant \mathit{NbT}$
$\Rightarrow \{$Specifications (6), (3) and (9), follows theorems$\}$
$\texttt{always}\ \sum_i(\mathit{Tokens}.\mathit{Client}_i.\mathit{giv} - \mathit{Tokens}.\mathit{Client}_i.\mathit{rel}) \leqslant \mathit{NbT}$

□

Property (2).

Proof. In resource allocation system:

$\{$Specification (3)$\}$
$\forall i :: \mathit{Client}_i.\mathit{ask} \nearrow \ \wedge\ \mathit{Client}_i.\mathit{rel} \nearrow$
$\Rightarrow \{$Specification (9), follows definition$\}$
$\forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge\ \mathit{Alloc}.\mathit{rel}_i \nearrow$
$\Rightarrow \{$Specification (4)$\}$
$\quad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge\ \mathit{Alloc}.\mathit{rel}_i \nearrow$
$\wedge\ \forall i :: \texttt{always}\ \langle \forall k :: \mathit{Client}_i.\mathit{ask}[k] \leqslant \mathit{NbT} \rangle$
$\Rightarrow \{$Specification (9), follows definition, calculus$\}$
$\quad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge\ \mathit{Alloc}.\mathit{rel}_i \nearrow$
$\wedge\ \texttt{always}\ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle$
$\Rightarrow \{$Specification (6)$\}$
$\quad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge\ \mathit{Alloc}.\mathit{rel}_i \nearrow$
$\wedge\ \texttt{always}\ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle$
$\wedge\ \forall i :: \mathit{Alloc}.\mathit{giv}_i \nearrow$
$\Rightarrow \{$Specification (9), follows definition$\}$

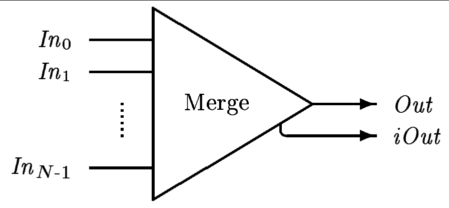
$$
\begin{array}{l}
\quad \forall i :: \mathit{Alloc.ask}_i \nearrow \ \wedge\ \mathit{Alloc.rel}_i \nearrow \\
\wedge\ \texttt{always}\ \langle \forall i, k :: \mathit{Alloc.ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\wedge\ \forall i :: \mathit{Client}_i.\mathit{giv} \nearrow \\
\Rightarrow \{\text{Specification (5)}\} \\
\quad \forall i :: \mathit{Alloc.ask}_i \nearrow \ \wedge\ \mathit{Alloc.rel}_i \nearrow \\
\wedge\ \texttt{always}\ \langle \forall i, k :: \mathit{Alloc.ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\wedge\ \forall i, h :: h_i \sqsubseteq \mathit{Client}_i.\mathit{giv} \wedge h_i \sqsubseteq_{\geqslant} \mathit{Client}_i.\mathit{ask} \mapsto \mathit{Tokens.Client}_i.\mathit{rel} \geqslant \mathit{Tokens}.h_i \\
\Rightarrow \{\text{Specification (9), follows definition, transitivity of } \mapsto \} \\
\quad \forall i :: \mathit{Alloc.ask}_i \nearrow \ \wedge\ \mathit{Alloc.rel}_i \nearrow \\
\wedge\ \texttt{always}\ \langle \forall i, k :: \mathit{Alloc.ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\wedge\ \forall i, h :: h_i \sqsubseteq \mathit{Alloc.giv}_i \wedge h_i \sqsubseteq_{\geqslant} \mathit{Alloc.ask}_i \mapsto \mathit{Tokens.Alloc.rel}_i \geqslant \mathit{Tokens}.h_i \\
\Rightarrow \{\text{Specification (8)}\} \\
\forall i, h :: h \sqsubseteq \mathit{Alloc.ask}_i \mapsto h \sqsubseteq_{\leqslant} \mathit{Alloc.giv}_i \\
\Rightarrow \{\text{Specification (9), follows definition}\} \\
\forall i, h :: h \sqsubseteq \mathit{Client}_i.\mathit{ask} \mapsto h \sqsubseteq_{\leqslant} \mathit{Client}_i.\mathit{giv}
\end{array}
$$

□

5 From a Single-Client Allocator to a General Allocator

5.1 Components Specifications

The first component we use is a fair merge. It merges N input channels (*In*) into one output channel (*Out*). Furthermore, it provides, for each output message, the number of the channel where it comes from (*iOut*) (Fig. 6). This merge component is assumed to be fair: No input channel can be ignored indefinitely. A merge component is nondeterministic.



$$\mathit{true}\ \textbf{guarantees}\ \mathit{Out} \nearrow \ \wedge\ \mathit{iOut} \nearrow \tag{10}$$

$$\mathit{true}\ \textbf{guarantees}\ \texttt{always}\ |\mathit{Out}| = |\mathit{iOut}| \tag{11}$$

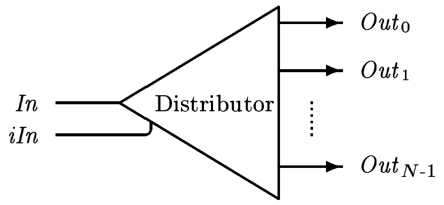
$$\mathit{true}\ \textbf{guarantees}\ \texttt{always}\ \langle \forall k :: 0 \leqslant \mathit{iOut}[k] < N \rangle \tag{12}$$

$$\begin{array}{c} \forall i :: \mathit{In}_i \nearrow \\ \textbf{guarantees} \\ \forall i :: \mathit{Out}/\{k \mid 1 \leqslant k \leqslant |\mathit{iOut}| \wedge \mathit{iOut}[k] = i\} \mathrel{\underline{\boxed{\diamond}}} \mathit{In}_i \end{array} \tag{13}$$

Fig. 6. Merge specification.

The main merge specification is (13). But since this specification only constraints values when indexes are present (and indexes when values are present), we force that there is no value without the corresponding index, and no index without the corresponding value (11). Moreover, specification (13) does not constraint values corresponding to nonvalid indexes. So, we force that there is never a nonvalid index (12). Finally, these specifications force the output to be monotonic *in length* only (some value may still be replaced by another, changing the corresponding index at the same time). Therefore, we add the constraint (10). We obtain the specification in fig. 6.

The distributor component is the symmetric of the merge: Given a valid index in *iIn* and a value in *In*, it outputs the value in the right output queue Out_i (Fig. 7). A distributor component is assumed to be fair: If indexes are present, the input channel cannot be ignored indefinitely. A distributor component is deterministic.



$$\begin{gathered} In \nearrow \ \wedge\ iIn \nearrow \ \wedge\ \texttt{always}\ \langle \forall k :: 0 \leqslant iIn[k] < N \rangle \\ \textbf{guarantees} \\ \forall i :: Out_i \boxtimes In/\{k \mid 1 \leqslant k \leqslant |iIn| \wedge iIn[k] = i\} \end{gathered} \tag{14}$$

Fig. 7. Distributor specification.

The distributor main specification is similar to the corresponding merge specification (Fig. 7). One important difference is the left-hand side of the *guarantees*: We have to assume that the indexes provided in *iIn* are correct. The difficulties we had with the merge, leading us to add several specifications, do not appear here. In particular, the monotonicity of outputs is a theorem (22).

We now combine these merge and distributor components with a simple allocator that only deals with a unique client. The requests are merged towards this simple allocator. Their origins are transfered directly to a distributor which is responsible for sending the tokens given by the allocator to the right addresses. Releases are also merged towards the simple allocator. Their origins are not used. All four components are connected via a network described with follows relations (Fig. 8).

The allocator we build from this composition provides a specification stronger than the required specification [4]. The difference is that the origin of releases is completely ignored. Therefore, the resulting allocator only cares about the *total* number of tokens coming back. This allows clients to exchange tokens and

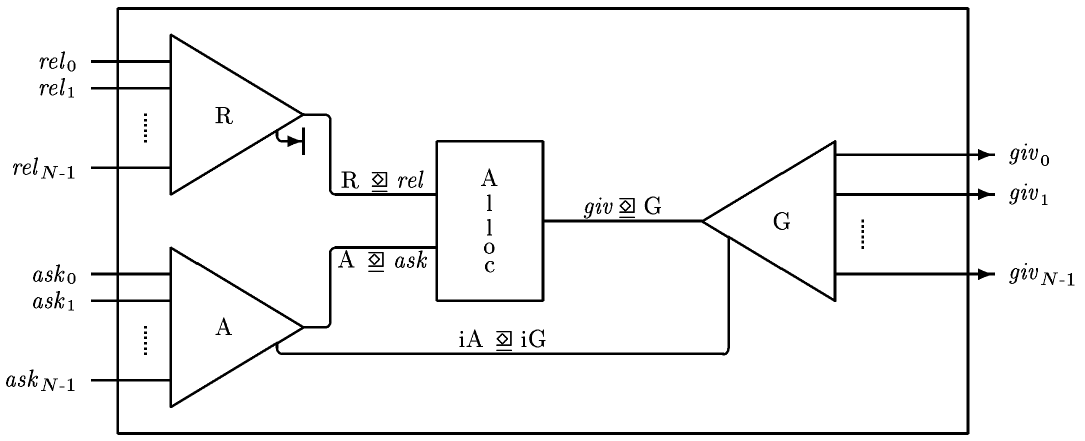


Fig. 8. General allocator architecture and network specification.

a client can return tokens received by another client. This leads to a weaker assumption in the left-hand side of the liveness property (Fig. 9).

$$\textit{true}\ \mathbf{guarantees}\ \forall i :: \mathit{giv}_i \nearrow \tag{15}$$

$$(\forall i :: \mathit{rel}_i \nearrow)\ \mathbf{guarantees}\ \texttt{always}\ \textstyle\sum_i(\mathit{Tokens}.\mathit{giv}_i - \mathit{Tokens}.\mathit{rel}_i) \leqslant \mathit{NbT} \tag{16}$$

$$\begin{array}{ll} & \forall i :: \mathit{ask}_i \nearrow\ \wedge\ \mathit{rel}_i \nearrow \\ \wedge & \texttt{always}\ \langle \forall i,k :: \mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\ \wedge & \forall h :: \langle \forall i :: h_i \sqsubseteq \mathit{giv}_i \wedge h_i \sqsubseteq_{\geqslant} \mathit{ask}_i \rangle \mapsto \sum_i \mathit{Tokens}.\mathit{rel}_i \geqslant \sum_i \mathit{Tokens}.h_i \\ & \mathbf{guarantees} \\ & \forall i,h :: h \sqsubseteq \mathit{ask}_i \mapsto h \sqsubseteq_{\leqslant} \mathit{giv}_i \end{array} \tag{17}$$

Fig. 9. Allocator specification (provided).

The previous construction relies on a single-client allocator. Its specification is derived from specification [9] for $N = 1$ (Fig. 10).

5.2 Basic Properties

In this section, we state some basic properties satisfied by the merge and distributor components. We can either consider them as lemmas in the compositional proof, or as additional specifications provided, for instance, by a previous use of these components (see conclusions).

Merge Component. The following formula states that the output (in terms of *bags*) *follows* the input of a merge. In other words, there are always less messages

$$\textit{true}\ \textbf{guarantees}\ \textit{giv} \nearrow \tag{18}$$

$$\textit{rel} \nearrow\ \textbf{guarantees}\ \texttt{always}\ \textit{Tokens.giv} - \textit{Tokens.rel} \leqslant \textit{NbT} \tag{19}$$

$$\begin{array}{c} \textit{ask} \nearrow \wedge\ \textit{rel} \nearrow \\ \wedge \quad \texttt{always}\ \langle \forall k :: \textit{ask}[k] \leqslant \textit{NbT} \rangle \\ \wedge\ \forall h :: h \sqsubseteq \textit{giv} \wedge h \sqsubseteq_{\geqslant} \textit{ask} \mapsto \textit{Tokens.rel} \geqslant \textit{Tokens}.h \\ \textbf{guarantees} \\ \forall h :: h \sqsubseteq \textit{ask} \mapsto h \sqsubseteq_{\leqslant} \textit{giv} \end{array} \tag{20}$$

Fig. 10. Single-client allocator specification (required).

on the right of a merge than on the left, but some liveness works at reducing that difference.

$$(\forall i :: \textit{In}_i \nearrow)\ \textbf{guarantees}\ \mathcal{B}(\textit{Out}) \boxtimes \bigcup_i \mathcal{B}(\textit{In}_i) \tag{21}$$

Proof. See appendix A.1 □

Distributor Component. This property states the monotonicity of outputs:

$$\begin{array}{c} \textit{In} \nearrow\ \wedge\ \textit{iIn} \nearrow\ \wedge\ \texttt{always}\ \langle \forall k :: 0 \leqslant \textit{iIn}[k] < N \rangle \\ \textbf{guarantees} \\ \forall i :: \textit{Out}_i \nearrow \end{array} \tag{22}$$

Proof. See appendix A.2 □

The following property corresponds to property (21) for the merge:

$$\begin{array}{c} \textit{In} \nearrow\ \wedge\ \textit{iIn} \nearrow\ \wedge\ \texttt{always}\ \langle \forall k :: 0 \leqslant \textit{iIn}[k] < N \rangle \\ \textbf{guarantees} \\ \bigcup_i \mathcal{B}(\textit{Out}_i) \boxtimes \mathcal{B}(\textit{In}/\{k \mid 1 \leqslant k \leqslant |\textit{iIn}|\}) \end{array} \tag{23}$$

Proof. See appendix A.2 □

5.3 Refinement Proof

We need to show that the provided specification [9] is stronger that the required specification [4]. The only proof obligation is that (17) ⇒ (8). Since the right-hand sides are the same, we have to prove that the left-hand side of (8) is stronger that the left-hand-side of (17).

Proof.

$\{\text{lhs of (8)}\}$
$\forall i, h :: h_i \sqsubseteq giv_i \wedge h_i \sqsubseteq_{\geqslant} ask_i \mapsto Tokens.rel_i \geqslant Tokens.h_i$
$\Rightarrow \{rel_i \nearrow,\ Tokens.rel_i \nearrow,\ \text{PSP}\}$
$\forall h :: \langle \forall i :: h_i \sqsubseteq giv_i \wedge h_i \sqsubseteq_{\geqslant} ask_i \rangle \mapsto \langle \forall i :: Tokens.rel_i \geqslant Tokens.h_i \rangle$
$\Rightarrow \{\text{calculus}\}$
$\forall h :: \langle \forall i :: h_i \sqsubseteq giv_i \wedge h_i \sqsubseteq_{\geqslant} ask_i \rangle \mapsto \sum_i Tokens.rel_i \geqslant \sum_i Tokens.h_i$
$\{\text{lhs of (17)}\}$

□

The composition proof is given in appendix B.2

6 Clients

6.1 Model

A client handles a variable T which is randomly chosen between 1 and NbT and which represents the size of the next request. Requests for tokens are built by appending the value of T to the history of requests. There is exactly one *rel* message produced for each *giv* message received that satisfies the condition (enough tokens to serve the request). Such a client can send several requests before one request is answered, and can receive several answers before it releases one.

```
Program Client
Declare
    $giv$ : $input\ history$;
    $ask, rel$ : $output\ history$;
    $T$ : $bag\ of\ colors$;
Initially
    $1 \leqslant T \leqslant NbT$;
Assign (weak fairness)
    $rel := rel \bullet giv[|rel|+1]$ if $|rel| < |giv| \wedge giv[|rel|+1] \geqslant ask[|rel|+1]$
Assign (no fairness)
   [] $T := (T \bmod NbT) + 1$
   [] $ask := ask \bullet T$
End
```

6.2 Provided Specification

The client model provides a different (and stronger) liveness than requested. It is only requested that clients return the right *total* number of tokens. However, this client always return *all* the tokens corresponding to a request in a single message (Fig. 11).

The refinement proof ([11] ⇒ [3]) is given in appendix C.2

$$true\ \mathbf{guarantees}\ ask \nearrow \wedge rel \nearrow \tag{24}$$
$$true\ \mathbf{guarantees}\ \texttt{always}\ \langle \forall k :: ask[k] \leqslant NbT \rangle \tag{25}$$
$$giv \nearrow\ \mathbf{guarantees}\ \forall h :: h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto h \sqsubseteq rel \tag{26}$$

Fig. 11. Client specification (provided).

6.3 Correctness Proof

Property (24). Inductive and local.

Property (25). The program satisfies the following inductive invariant:

$$\texttt{invariant}\ \langle \forall k :: ask[k] \leqslant NbT \rangle \wedge (T \leqslant NbT)$$

which is local and stronger than the required *always* property.

Property (26).

Lemma 27.

$$\forall h, k :: \texttt{transient}\ rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \cdot Client \tag{27}$$

Proof. We use the fact that $(\texttt{transient}\ q) \wedge [p \Rightarrow q] \Rightarrow (\texttt{transient}\ p)$:

$rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask$
$\Rightarrow$ {Definition of $\sqsubseteq_{\geqslant}$, calculus}
$rel = k \wedge |rel| < |h| \leqslant |giv| \wedge \langle \forall n : 1 \leqslant n \leqslant |h| : giv[n] \geqslant ask[n] \rangle$
$\Rightarrow$ {Calculus}
$rel = k \wedge |rel| \leqslant |giv| \wedge \langle \forall n : 1 \leqslant n \leqslant |rel| + 1 : giv[n] \geqslant ask[n] \rangle$
$\Rightarrow$ {Choose $n = |rel| + 1$}
$rel = k \wedge |rel| \leqslant |giv| \wedge giv[|rel| + 1] \geqslant ask[|rel| + 1]$
{From the first program statement}
is *transient* for any k

□

Proof (specification (26)). In any composed system:

{From lemma (27), existentiality of *transient*}
$(rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask)$
$\mapsto \neg(rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask)$
$\Rightarrow$ {$giv \nearrow$ from lhs, $ask \nearrow$, PSP}
$rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto rel \neq k$
$\Rightarrow$ {$rel \nearrow$, PSP}
$rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto k \sqsubset rel$

$\Rightarrow$ {Induction on $|h| - |k|$}
$rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto rel = h$
$\Rightarrow$ {Weakening}
$rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto h \sqsubseteq rel$
$\Rightarrow$ {Disjunction over k}
$rel \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto h \sqsubseteq rel$
$\Rightarrow$ {Disjunction $[(p \wedge r \mapsto q) \Rightarrow ((p \vee q) \wedge r \mapsto q)]$}
$(rel \sqsubset h \vee h \sqsubseteq rel) \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto h \sqsubseteq rel$
$\Rightarrow$ {`always` $rel \sqsubseteq giv$ holds in system, hence $h \sqsubseteq giv \Rightarrow (rel \sqsubset h \vee h \sqsubseteq rel)$}
$h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto h \sqsubseteq rel$

□

7 The Single-Client Allocator

7.1 Model

The allocator uses a variable T to store the number of available tokens. It simply answers an unsatisfied request if there is enough tokens in T. The allocator also looks into its release port and "consumes" messages to increase T. It keeps track of the number of consumed messages in NbR.

```
Program Alloc
Declare
    $ask, rel$ : input history;
    $giv$ : output history;
    $T$ : bag of colors;
    $NbR$ : int;
Initially
    $T = NbT \wedge NbR = 0$
Assign (weak fairness)
    $giv, T := giv \bullet ask[|giv|+1], T - ask[|giv|+1]$
        if $|ask| > |giv| \wedge T \geqslant ask[|giv|+1]$
  [] $T, NbR := T + rel[NbR+1], NbR+1$ if $|rel| > NbR$
End
```

7.2 Provided Specification

The previous model provides a specification for the single client allocator stronger than the required specification [10].

The main difference is that this allocator waits for a request before it sends tokens (30). This is not explicitly required in specification [10]. Especially, we can imagine an allocator able to *guess* some clients requests, using, for instance, some knowledge that several clients have exactly the same behavior (client i asked for n tokens, therefore client j will also ask for n tokens.). The only constraint is that tokens given to a client correspond (possibly in the future) to a request from that client. Then, since this allocator does not send tokens without requests, it can expect the return of *all* the tokens it sent, which changes the left-hand side of the liveness property (31). Intuitively, this leads to a stronger specification: Never sending tokens without request and expecting the return of all tokens is stronger than only expecting the return of tokens sent in response to a request.

The second (minor) difference is that this allocator always gives *exactly* the right number of tokens. The resulting specification is summarized in fig. 12.

$$\mathit{true}\ \textbf{guarantees}\ \mathit{giv} \nearrow \tag{28}$$

$$\mathit{rel} \nearrow\ \textbf{guarantees}\ \texttt{always}\ \mathit{Tokens.giv} - \mathit{Tokens.rel} \leqslant \mathit{NbT} \tag{29}$$

$$\mathit{ask} \nearrow\ \textbf{guarantees}\ \texttt{always}\ \mathit{giv} \sqsubseteq \mathit{ask} \tag{30}$$

$$\begin{array}{c} \mathit{ask} \nearrow \wedge\ \mathit{rel} \nearrow \\ \wedge \quad \texttt{always}\ \langle \forall k :: \mathit{ask}[k] \leqslant \mathit{NbT} \rangle \\ \wedge\ \forall k :: \mathit{Tokens.giv} \geqslant k \mapsto \mathit{Tokens.rel} \geqslant k \\ \textbf{guarantees} \\ \forall h :: h \sqsubseteq \mathit{ask} \mapsto h \sqsubseteq \mathit{giv} \end{array} \tag{31}$$

Fig. 12. Single-client allocator specification (provided).

The refinement proof ([12] ⇒ [10]) is given in appendix C.3. The program text correctness proof is given in appendix D.3.

8 Conclusions

The allocator example illustrates the need, when adopting a component-based design, to switch between top-down and bottom-up approaches: A designer has in mind the global (at his level) system he wants to obtain. He deduces some expected components behaviors. Among these components, some are generic and he can expect to find them in some repository. However, he will have to design some other components by himself. Such a design can be compositional again (Sect. 5), or he can just program them in a traditional way (Sect. 6 and 7).

While building these components (by programming, or by further decomposition), he adopts the provider point of view: he looks at what he gets and expresses it logically. The provided specification and the required specification need not be the same and, in general, they are different. This is because when specifying a required behavior, one does not want to demand too much; and on the other hand, when programming a model, one does not try to obtain the weakest possible solution. If the user and the provider are forced to share some average common specification, they remain unsatisfied: “Why should I ask for things I don’t need?”, “Why should I hide some properties my program has?”. Because we hope for reusability, a component should be finally published with its provided specification, as well as with different weaker specifications (Fig. 13). One advantage in publishing these weak specifications is that they may allow a reuse of the component while avoiding either another refinement proof or a complex compositional proof (using directly the provided specification). These weaker specifications can be obtained from previous uses of that component. Another possibility is that the component implementor invests effort in proving several specifications of his component, and hence reduces the work of the composer who can use the most convenient specification.

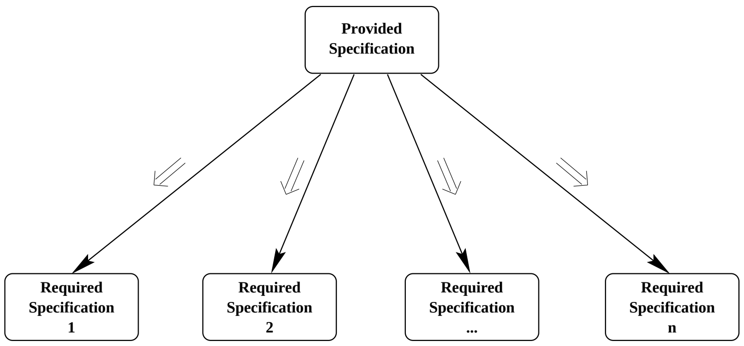


Fig. 13. Multiple use of a component.

As we see on the example, this approach requires three kinds of proofs: *compositional proofs*, to deduce the correctness of a system (or sub-system) from components correctness; *refinement proofs* to check that bottom-up phases provides components stronger than requested during top-down phases; *program text correctness proofs* to relate, at the bottom, program texts to logical specifications. The framework we are using, based on UNITY, extended with composition operators (*guarantees*) and some communication abstraction (*follows*), is able to handle those three types of proofs, while remaining in the minimalism spirit of UNITY.

Our goal being to build distributed systems from components, it is very important to have the possibility to specify components in terms of input and outputs, and to be able to connect them formally through simple proofs. Mixing a temporal operator like *follows* with some basic sequences properties seems an interesting approach. We are currently investigating how far this approach can go. Especially, we would like to express both traditional functional behaviors and useful nondeterministic behaviors (like *merge*) with a common notation that would be able to handle their interaction nicely.

Another area of interest is *universal* properties. Throughout the allocator example, all composition aspects are handled with *existential* properties. Although it it easier to use existential properties than universal properties, it seems that they can become insufficient when dealing with global complex safety properties. We are currently investigating examples involving such global complex safety properties to learn more about universally-based composition [5].

The motivation for this research is the development of large repositories of software components. Designers can discover implementations of components and use relatively simple compositional structures to create useful software systems. Widespread deployment of such repositories may be years away. Nevertheless, we believe that research to support such repositories is interesting both because it offers intellectually stimulating problems and because it is useful.

We have been working on composition in which shared variables are modified only by one component and read by others. Further, proofs are simplified if these shared variables have a monotonic structure. Existential and universal property types make proof obligations very clear, and these property types yield nice proof rules that appear, at least at this early stage in our investigation, to be well suited for mechanical theorem provers. We have started a collaboration with Larry Paulson who has successfully used *Isabelle* [7] to prove the correctness of such systems.

Much work remains to be done to achieve our goal of large repositories of software components with their proofs of correctness. This is a step in that direction.

References

1. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
2. K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.
3. 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.
4. Michel Charpentier. *Assistance à la Répartition de Systèmes Réactifs.* PhD thesis, Institut National Polytechnique de Toulouse, France, November 1997.
5. Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, Lecture Notes in Computer Science, Puerto Rico, April 1999. Springer-Verlag.
6. Jayadev Misra. *A Logic for Concurrent Programming.* Technical Report ("New UNITY"), Department of Computer Science, University of Texas at Austin, 1994.
7. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.
8. Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.
9. Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems.* PhD thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997.

A Basic Properties Proofs

A.1 Merge

Property (21).

$$\langle \forall i :: In_i \nearrow \rangle \ \textbf{guarantees}\ \mathcal{B}(Out) \trianglelefteq \bigcup_i \mathcal{B}(In_i) \qquad (21)$$

Proof.

$\quad \mathcal{B}(Out)$
$=\ \{$From specification (11), $|Out| = |iOut|\}$
$\quad \mathcal{B}(Out/\{k \mid 1 \leqslant k \leqslant |iOut|\})$
$=\ \{$From specification (12), for any k:
$\qquad \langle \exists i : 0 \leqslant i < N : iOut[k] = i \rangle \equiv true\}$
$\quad \mathcal{B}(Out/\{k \mid 1 \leqslant k \leqslant |iOut| \wedge \langle \exists i : 0 \leqslant i < N : iOut[k] = i \rangle\})$
$=\ \{$Calculus$\}$
$\quad \mathcal{B}(Out/\bigcup_{i=0}^{N-1}\{k \mid 1 \leqslant k \leqslant |iOut| \wedge iOut[k] = i\})$
$=\ \{$Sets are disjoint, lemma (42)$\}$
$\quad \bigcup_{i=0}^{N-1} \mathcal{B}(Out/\{k \mid 1 \leqslant k \leqslant |iOut| \wedge iOut[k] = i\})$
$\trianglelefteq\ \{$From specification (13), follows theorems$\}$
$\quad \bigcup_{i=0}^{N-1} \mathcal{B}(In_i)$

□

A.2 Distributor

Property (22).

$$\begin{array}{c} In \nearrow \ \wedge\ iIn \nearrow \ \wedge\ \texttt{always}\ \langle \forall k :: 0 \leqslant iIn[k] < N \rangle \\ \textbf{guarantees} \\ \forall i :: Out_i \nearrow \end{array} \qquad (22)$$

Proof. From specification (14), any follows' right-hand side being monotonic.

□

Property (23).

$$\begin{array}{c} In \nearrow \ \wedge\ iIn \nearrow \ \wedge\ \texttt{always}\ \langle \forall k :: 0 \leqslant iIn[k] < N \rangle \\ \textbf{guarantees} \\ \bigcup_i \mathcal{B}(Out_i) \trianglelefteq \mathcal{B}(In/\{k \mid 1 \leqslant k \leqslant |iIn|\}) \end{array} \qquad (23)$$

Proof.

$\quad \bigcup_{i=0}^{N-1} \mathcal{B}(Out_i)$
$\trianglelefteq\ \{$From specification (14) and follows theorems$\}$
$\quad \bigcup_{i=0}^{N-1} \mathcal{B}(In/\{k \mid 1 \leqslant k \leqslant |iIn| \wedge iIn[k] = i\})$
$=\ \{$Sets are disjoint, lemma (42)$\}$
$\quad \mathcal{B}(In/\bigcup_{i=0}^{N-1}\{k \mid 1 \leqslant k \leqslant |iIn| \wedge iIn[k] = i\})$
$=\ \{$Calculus$\}$
$\quad \mathcal{B}(In/\{k \mid 1 \leqslant k \leqslant |iIn| \wedge \langle \exists i : 0 \leqslant i < N : iIn[k] = i \rangle\})$
$=\ \{$From lhs, $\langle \exists i : 0 \leqslant i < N : iIn[k] = i \rangle \equiv true\}$
$\quad \mathcal{B}(In/\{k \mid 1 \leqslant k \leqslant |iIn|\})$

□

B Compositional Proofs

B.1 C1

See sect. 4.2.

B.2 C2

Lemmas and Corollaries. The following lemmas and corollaries are used in the proof. For any binary relation $\mathcal{R}$, any sequences Q, $`Q$, $Q'$, $`Q'$, Q'', Q''', any sets S, S', and any predicate P:

Lemma 32.
$$[Q \sqsubseteq Q' \sqsubseteq_{\mathcal{R}} Q'' \sqsubseteq Q''' \;\Rightarrow\; Q \sqsubseteq_{\mathcal{R}} Q'''] \tag{32}$$

Lemma 33. *If $\mathcal{R}$ is an order relation (reflexive, transitive, antisymmetric), then $\sqsubseteq_{\mathcal{R}}$ is an order relation.*

Lemma 34. *If $\mathcal{R}$ is an order relation, then:*
$$`Q \,\underline{\boxdiamond}\, Q \; wrt \sqsubseteq \;\Rightarrow\; `Q \,\underline{\boxdiamond}\, Q \; wrt \sqsubseteq_{\mathcal{R}} \tag{34}$$

Lemma 35.
$$[Q \sqsubseteq_{\mathcal{R}} Q' \wedge S \leqslant S' \;\Rightarrow\; Q/S \sqsubseteq_{\mathcal{R}} Q'/S'] \tag{35}$$

Corollary 36.
$$Q \nearrow_{\sqsubseteq} \; \wedge S \nearrow_{\leqslant} \Rightarrow\; Q/S \nearrow_{\sqsubseteq} \tag{36}$$

Lemma 37.
$$[Q \sqsubseteq Q' \;\Rightarrow\; \{k \mid 1 \leqslant k \leqslant |Q| \wedge P.Q[k]\} \leqslant \{k \mid 1 \leqslant k \leqslant |Q'| \wedge P.Q'[k]\}] \tag{37}$$

Corollary 38.
$$Q \nearrow_{\sqsubseteq} \Rightarrow\; \{k \mid 1 \leqslant k \leqslant |Q| \wedge P.Q[k]\} \nearrow_{\leqslant} \tag{38}$$

Corollary 39.
$$Q \nearrow_{\sqsubseteq} \; \wedge Q' \nearrow_{\sqsubseteq} \Rightarrow\; Q/\{k \mid 1 \leqslant k \leqslant |Q'| \wedge P.Q'[k]\} \nearrow_{\sqsubseteq} \tag{39}$$

Lemma 40.
$$Q \nearrow_{\sqsubseteq} \; \wedge S \nearrow_{\leqslant} \; \wedge `Q \,\underline{\boxdiamond}\, Q \;\wedge\; `S \,\underline{\boxdiamond}\, S \;\Rightarrow\; `Q/`S \,\underline{\boxdiamond}\, Q/S \tag{40}$$

Corollary 41.
$$\begin{gathered} Q \nearrow_{\sqsubseteq} \; \wedge Q' \nearrow_{\sqsubseteq} \; \wedge `Q \,\underline{\boxdiamond}\, Q \;\wedge\; `Q' \,\underline{\boxdiamond}\, Q' \\ \Rightarrow \\ `Q/\{k \mid 1 \leqslant k \leqslant |`Q'| \wedge P.`Q'[k]\} \,\underline{\boxdiamond}\, Q/\{k \mid 1 \leqslant k \leqslant |Q'| \wedge P.Q'[k]\} \end{gathered} \tag{41}$$

Lemma 42.

$$[S \cap S' = \emptyset \ \Rightarrow\ \mathcal{B}(Q/S \cup S') = \mathcal{B}(Q/S) \cup \mathcal{B}(Q/S')] \tag{42}$$

The following lemma relates the bags of histories to the number of tokens in these histories:

Lemma 43 (bags and tokens relationship). *For any queues Q, Q' and Q'':*

$$\begin{array}{c} [\mathcal{B}(Q) = \mathcal{B}(Q') \ \Rightarrow\ \mathit{Tokens}.Q = \mathit{Tokens}.Q'] \\ [\mathcal{B}(Q) = \mathcal{B}(Q') \cup \mathcal{B}(Q'') \ \Rightarrow\ \mathit{Tokens}.Q = \mathit{Tokens}.Q' + \mathit{Tokens}.Q''] \end{array} \tag{43}$$

In the remainder of the proof, variable names are not prefixed with component name. We use instead the names of fig. 8. Properties are stated for the global system.

Proof of (15).

Proof.

$\{$Inner allocator specification (6), merge specifications (10) and (12)$\}$
$\mathit{giv} \nearrow \ \wedge\, iA \nearrow \ \wedge\, \texttt{always}\ \langle \forall k :: 0 \leqslant iA[k] < N \rangle$
$\Rightarrow$ $\{$Follows definition, weakening $\boxtimes$ into $\sqsubseteq\}$
$G \nearrow \ \wedge\, iG \nearrow \ \wedge\, \texttt{always}\ \langle \forall k :: 0 \leqslant iG[k] < N \rangle$
$\Rightarrow$ $\{$Distributor specification (22)$\}$
$\forall i :: \mathit{giv}_i \nearrow$

□

Proof of (16).

Proof. From merge specifications and follows definition, *rel* $\nearrow$ and specification (7) is applicable. Then:

$\sum_i \mathit{Tokens}.\mathit{giv}_i - \sum_i \mathit{Tokens}.\mathit{rel}_i$
$\leqslant$ $\{$From distributor property (23), using lemma (43)$\}$
$\mathit{Tokens}.G - \sum_i \mathit{Tokens}.\mathit{rel}_i$
$\leqslant$ $\{$Weakening $G \boxtimes \mathit{giv}$ into subbag relation, using lemma (43)$\}$
$\mathit{Tokens}.\mathit{giv} - \sum_i \mathit{Tokens}.\mathit{rel}_i$
$\leqslant$ $\{$Same work on *rel*, using merge property (21)$\}$
$\mathit{Tokens}.\mathit{giv} - \mathit{Tokens}.\mathit{rel}$
$\leqslant$ $\{$From inner allocator specification (7)$\}$
NbT

□

Proof of (17).

Lemma 44. *The inner allocator is live (it eventually answers any request).*

$$\forall h :: h \sqsubseteq \mathit{ask} \mapsto h \sqsubseteq_{\leqslant} \mathit{giv} \tag{44}$$

Proof. We show that the lhs of (8) is satisfied:

1. $$ask \nearrow \wedge\ rel \nearrow$$
   from merge specification and follows definition.

2. $$\texttt{always}\ \langle \forall n :: ask[n] \leqslant NbT \rangle$$

   $\{$From lhs of (17)$\}$
   $\texttt{always}\ \langle \forall i, n :: ask_i[n] \leqslant NbT \rangle$
   $\Rightarrow$ $\{$Definition of $\mathcal{B}(ask_i)\}$
   $\texttt{always}\ \langle \forall i, x : x \in \mathcal{B}(ask_i) : x \leqslant NbT \rangle$
   $\Rightarrow$ $\{\mathcal{B}(A) \mathrel{\underline{\boxdiamond}} \bigcup_i \mathcal{B}(ask_i)$ from (21), calculus$\}$
   $\texttt{always}\ \langle \forall x : x \in \mathcal{B}(A) : x \leqslant NbT \rangle$
   $\Rightarrow$ $\{$Weakening $ask \mathrel{\underline{\boxdiamond}} A$ into subbag relation$\}$
   $\texttt{always}\ \langle \forall x : x \in \mathcal{B}(ask) : x \leqslant NbT \rangle$
   $\Rightarrow$ $\{$Definition of $\mathcal{B}(ask)\}$
   $\texttt{always}\ \langle \forall n :: ask[n] \leqslant NbT \rangle$

3. $$\forall h :: h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \mapsto Tokens.rel \geqslant Tokens.h$$

   $h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask$
   $\Rightarrow$ $\{$Choice of $a\}$
   $h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \wedge iA = a$
   $\mapsto$ $\{$From $iG \mathrel{\underline{\boxdiamond}} iA$ and monotonicity of histories$\}$
   $h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \wedge a \sqsubseteq iA \wedge a \sqsubseteq iG$
   $\Rightarrow$ $\{$Notation: $h_i = h/\{k \mid 1 \leqslant k \leqslant |h| \wedge a[k] = i\}$, lemma (35)$\}$
   $\langle \forall i :: h_i \sqsubseteq giv/\{k \mid 1 \leqslant k \leqslant |h| \wedge a[k] = i\}$
   $\quad\wedge\quad h_i \sqsubseteq_{\geqslant} ask/\{k \mid 1 \leqslant k \leqslant |h| \wedge a[k] = i\}\rangle$
   $\quad\wedge\quad a \sqsubseteq iA \wedge a \sqsubseteq iG$
   $\Rightarrow$ $\{$From merge specification (11), $|h| \leqslant |a|$, then lemma (35)$\}$
   $\langle \forall i :: h_i \sqsubseteq giv/\{k \mid 1 \leqslant k \leqslant |a| \wedge a[k] = i\}$
   $\quad\wedge\quad h_i \sqsubseteq_{\geqslant} ask/\{k \mid 1 \leqslant k \leqslant |a| \wedge a[k] = i\}\rangle$
   $\quad\wedge\quad a \sqsubseteq iA \wedge a \sqsubseteq iG$
   $\Rightarrow$ $\{$Using $a \sqsubseteq iA$ and $a \sqsubseteq iG$, lemmas (37) and (35)$\}$
   $\langle \forall i :: h_i \sqsubseteq giv/\{k \mid 1 \leqslant k \leqslant |iG| \wedge iG[k] = i\}$
   $\quad\wedge\quad h_i \sqsubseteq_{\geqslant} ask/\{k \mid 1 \leqslant k \leqslant |iA| \wedge iA[k] = i\}\rangle$
   $\Rightarrow$ $\{$Merge specification (13), follows liveness$\}$
   $\langle \forall i :: h_i \sqsubseteq giv_i$
   $\quad\wedge\quad h_i \sqsubseteq_{\geqslant} ask/\{k \mid 1 \leqslant k \leqslant |iA| \wedge iA[k] = i\}\rangle$
   $\Rightarrow$ $\{$Merge specification (13), weakening $\mathrel{\underline{\boxdiamond}}$ into $\sqsubseteq_{\geqslant}\}$
   $\langle \forall i :: h_i \sqsubseteq giv_i \wedge h_i \sqsubseteq_{\geqslant} ask_i \rangle$
   $\mapsto$ $\{$From lhs of (17)$\}$
   $\sum_i Tokens.rel_i \geqslant \sum_i Tokens.h_i$
   $\mapsto$ $\{$From merge property (21) and lemma (43)$\}$
   $Tokens.rel \geqslant \sum_i Tokens.h_i$
   $\Rightarrow$ $\{$Using $\sum_i Tokens.h_i \ = \ Tokens.h\}$
   $Tokens.rel \geqslant Tokens.h$

$\square$

Lemma 45.

$$S \nearrow_{\leqslant} \;\Rightarrow\; (\forall h, S :: h \sqsubseteq ask/S \mapsto h \sqsubseteq_{\leqslant} giv/S) \tag{45}$$

Proof.

$$\begin{array}{ll}
 & \{\text{Lemma } (44)\} \\
 & \forall k :: k \sqsubseteq ask \mapsto k \sqsubseteq_{\leqslant} giv \\
\Rightarrow & \{\text{Choosing } k = ask\text{, strengthening lhs, } k/S \nearrow \text{ from corollary } (36)\} \\
 & \forall h :: h \sqsubseteq ask/S \wedge ask = k \mapsto h \sqsubseteq k/S \wedge k \sqsubseteq_{\leqslant} giv \\
\Rightarrow & \{\text{Lemma } (35)\} \\
 & \forall h :: h \sqsubseteq ask/S \wedge ask = k \mapsto h \sqsubseteq k/S \wedge k/S \sqsubseteq_{\leqslant} giv/S \\
\Rightarrow & \{\text{Lemma } (32)\} \\
 & \forall h :: h \sqsubseteq ask/S \wedge ask = k \mapsto h \sqsubseteq_{\leqslant} giv/S \\
\Rightarrow & \{\text{Disjunction}\} \\
 & \forall h :: h \sqsubseteq ask/S \mapsto h \sqsubseteq_{\leqslant} giv/S
\end{array}$$

□

Proof (specification (17)).

$$\begin{array}{ll}
 & h \sqsubseteq ask_i \\
\mapsto & \{ask_i \nearrow\text{, merge specification } (13)\text{, follows liveness}\} \\
 & h \sqsubseteq A/\{k \mid 1 \leqslant k \leqslant |iA| \wedge iA[k] = i\} \\
\mapsto & \{ask \boxtimes A,\ iG \boxtimes iA\text{, corollary } (41)\text{, follows liveness}\} \\
 & h \sqsubseteq ask/\{k \mid 1 \leqslant k \leqslant |iG| \wedge iG[k] = i\} \\
\mapsto & \{\text{Lemma } (45)\} \\
 & h \sqsubseteq_{\leqslant} giv/\{k \mid 1 \leqslant k \leqslant |iG| \wedge iG[k] = i\} \\
\mapsto & \{\text{From distributor specification } (14)\text{, lemma } (34)\text{, follows liveness}\} \\
 & h \sqsubseteq_{\leqslant} giv_i
\end{array}$$

□

C Refinement Proofs

C.1 R1

See sect. 5.3.

C.2 R2

We need to show that the provided specification [11] is stronger that the required specification [3]. The only proof obligation is that (26) $\Rightarrow$ (5), which is trivial since

$$[h \sqsubseteq rel \;\Rightarrow\; Tokens.rel \geqslant Tokens.h]$$

C.3 R3

We need to show that the provided specification [12] is stronger that the required specification [10]. The only proof obligation is that [12] $\Rightarrow$ (20).

Lemma 46.

$$[12] \land \mathit{lhs\ of}\ (20)\ \Rightarrow\ (\forall k :: \mathit{Tokens.giv} \geqslant k \mapsto \mathit{Tokens.rel} \geqslant k) \tag{46}$$

Proof.

$$\begin{array}{ll}
 & \mathit{Tokens.giv} \geqslant k \\
\Rightarrow & \{\text{Choice of } h\} \\
 & \mathit{Tokens.giv} \geqslant k \land \mathit{giv} = h \\
\Rightarrow & \{\mathit{ask} \nearrow \text{ from lhs of (20), use specification (30)}\} \\
 & \mathit{Tokens.giv} \geqslant k \land \mathit{giv} = h \land h \sqsubseteq \mathit{ask} \\
\mapsto & \{\text{From lhs of (20)}\} \\
 & \mathit{Tokens.rel} \geqslant \mathit{Tokens.h} \\
\Rightarrow & \{\text{From } \mathit{Tokens.h} \geqslant k\} \\
 & \mathit{Tokens.rel} \geqslant k
\end{array}$$

□

Proof (specification (20)). Assuming [12]:

$$\begin{array}{ll}
 & \mathit{lhs\ of}\ (20) \\
\Rightarrow & \{\text{Lemma (46)}\} \\
 & \mathit{lhs\ of}\ (20) \land (\forall k :: \mathit{Tokens.giv} \geqslant k \mapsto \mathit{Tokens.rel} \geqslant k) \\
\Rightarrow & \{\text{Calculus}\} \\
 & \mathit{lhs\ of}\ (31) \\
\textbf{guarantees} & \{\text{Using (31) from [12]}\} \\
 & \forall h :: h \sqsubseteq \mathit{ask} \mapsto h \sqsubseteq \mathit{giv} \\
\Rightarrow & \{\text{Weakening } \sqsubseteq \text{ into } \sqsubseteq_{\geqslant}\} \\
 & \mathit{rhs\ of}\ (20)
\end{array}$$

□

D Text Correctness Proofs

D.1 Lemma for Proving Guarantees

The following lemma is useful when proving some *guarantees* properties from a component program text. It allows us to derive a (weak) system property (right-hand side of a *guarantees*) from a strong local property (from the program text), a weak system property (left-hand side of the *guarantees*) and a strong system property (from locality hypotheses).

Lemma 47. *Given programs F and G, and predicate P, the following rule holds:*

$$\frac{\begin{array}{c}\mathtt{stable}_S\ P.x.y \cdot F \\ \forall k :: \mathtt{stable}_W\ P.k.y \cdot F [\!] G \\ \forall k :: \mathtt{stable}_S\ x = k \cdot G\end{array}}{\mathtt{stable}_W\ P.x.y \cdot F [\!] G} \tag{47}$$

Proof.

$(\texttt{stable}_S\ P.x.y \cdot F) \wedge$
$(\forall k :: \texttt{stable}_W\ P.k.y \cdot F[\![G) \wedge$
$(\forall k :: \texttt{stable}_S\ x = k \cdot G)$
$\Rightarrow$ {Translating *stable* into *next*}
$(P.x.y\ \texttt{next}_S\ P.x.y \cdot F) \wedge$
$(\forall k :: P.k.y\ \texttt{next}_W\ P.k.y \cdot F[\![G) \wedge$
$(\forall k :: x = k\ \texttt{next}_S\ x = k \cdot G)$
$\Rightarrow$ {lhs strengthening, rhs weakening of *next*}
$(\forall k :: x = k \wedge P.x.y\ \texttt{next}_S\ x = k \vee P.x.y \cdot F) \wedge$
$(\forall k :: P.k.y\ \texttt{next}_W\ P.k.y \cdot F[\![G) \wedge$
$(\forall k :: x = k \wedge P.x.y\ \texttt{next}_S\ x = k \vee P.x.y \cdot G)$
$\Rightarrow$ {Universality of $next_S$}
$(\forall k :: x = k \wedge P.x.y\ \texttt{next}_S\ x = k \vee P.x.y \cdot F[\![G) \wedge$
$(\forall k :: P.k.y\ \texttt{next}_W\ P.k.y \cdot F[\![G)$
$\Rightarrow$ {Conjunctivity of *next*}
$\forall k :: x = k \wedge P.x.y \wedge P.k.y\ \texttt{next}_W\ (x = k \vee P.x.y) \wedge P.k.y \cdot F[\![G$
$\Rightarrow$ {Predicate calculus: $lhs \equiv (x = k \wedge P.x.y)$, $rhs \Rightarrow P.x.y$}
$\forall k :: x = k \wedge P.x.y\ \texttt{next}_W\ P.x.y \cdot F[\![G$
$\Rightarrow$ {Disjunctivity of *next*}
$\langle \exists k :: x = k \wedge P.x.y \rangle\ \texttt{next}_W\ P.x.y \cdot F[\![G$
$\Rightarrow$ {Predicate calculus}
$P.x.y\ \texttt{next}_W\ P.x.y \cdot F[\![G$
$\Rightarrow$ {Translating *next* into *stable*}
$\texttt{stable}_W\ P.x.y \cdot F[\![G$

□

D.2 T1

See sect. 6.3.

D.3 T2

Property (28). Inductive and local.

Property (29).

Proof.

{From program text}
$\texttt{stable}_S\ T \geqslant 0 \quad \cdot Alloc$
$\texttt{stable}_S\ NbR \leqslant |rel| \quad \cdot Alloc$
$\texttt{stable}_S\ T = NbT - Tokens.giv + \sum_{i=1}^{NbR} rel[i] \cdot Alloc$
$\Rightarrow$ {Let $x = (T, NbR, giv)$ and $y = rel$ and
$$P.x.y = (T \geqslant 0) \wedge (NbR \leqslant |rel|) \wedge (T = NbT - Tokens.giv + \sum_{i=1}^{NbR} rel[i])\}$$

$\texttt{stable}_S\ P.x.y \cdot \mathit{Alloc}$

$\Rightarrow$ {From locality and lhs of (29)}

$\quad\texttt{stable}_S\ P.x.y \cdot \mathit{Alloc}$

$\wedge\ (\forall k :: \texttt{stable}_W\ P.k.y \cdot \mathit{Alloc} \| \mathit{env})\ \wedge\ (\forall k :: \texttt{stable}_S\ x = k \cdot \mathit{env})$

$\Rightarrow$ {Lemma (47)}

$\texttt{stable}_W\ P.x.y \cdot \mathit{Alloc} \| \mathit{env}$

$\Rightarrow$ {Histories are empty in initial state, assume $\mathit{NbT} \geqslant 0$}

$\texttt{always}\ P.x.y \cdot \mathit{Alloc} \| \mathit{env}$

$\Rightarrow$ {Expanding predicate P}

$$\begin{array}{c}\texttt{always}\\ (T \geqslant 0) \wedge (\mathit{NbR} \leqslant |\mathit{rel}|) \wedge (T = \mathit{NbT} - \mathit{Tokens.giv} + \sum_{i=1}^{\mathit{NbR}} \mathit{rel}[i])\\ \cdot \mathit{Alloc} \| \mathit{env}\end{array} \tag{48}$$

$\Rightarrow$ {From $[\mathit{NbR} \leqslant |\mathit{rel}| \Rightarrow \sum_{i=1}^{\mathit{NbR}} \mathit{rel}[i] \leqslant \mathit{Tokens.rel}]$}

$\texttt{always}\ \mathit{Tokens.giv} - \mathit{Tokens.rel} \leqslant \mathit{NbT} \cdot \mathit{Alloc} \| \mathit{env}$

□

Property (30).

Proof. Let $x = \mathit{giv}$, $y = \mathit{ask}$ and $P.x.y \equiv x \sqsubseteq y$.

{From program text}

$\texttt{stable}_S\ P.x.y \cdot \mathit{Alloc}$

$\Rightarrow$ {From locality and lhs of (30)}

$\quad\texttt{stable}_S\ P.x.y \cdot \mathit{Alloc}$

$\wedge\ (\forall k :: \texttt{stable}_W\ P.k.y \cdot \mathit{Alloc} \| \mathit{env})\ \wedge\ (\forall k :: \texttt{stable}_S\ x = k \cdot \mathit{env})$

$\Rightarrow$ {Lemma (47)}

$\texttt{stable}_W\ P.x.y \cdot \mathit{Alloc} \| \mathit{env}$

$\Rightarrow$ {Histories are empty in initial state}

$\texttt{always}\ P.x.y \cdot \mathit{Alloc} \| \mathit{env}$

$\Rightarrow$ {Expanding predicate P}

$\texttt{always}\ \mathit{giv} \sqsubseteq \mathit{ask} \cdot \mathit{Alloc} \| \mathit{env}$

□

Property (31). All the following properties are stated for $\mathit{Alloc} \| \mathit{env}$, including *transient* properties that come from the corresponding *transient* property in *Alloc* alone.

Lemma 49.

$$\forall k :: |\mathit{rel}| \geqslant k \mapsto \mathit{NbR} \geqslant k \tag{49}$$

Proof.

$\{\text{From program text}\}$
$\forall k :: \texttt{transient}\ |rel| \geqslant k \wedge NbR = k - 1$
$\Rightarrow \{\text{Using } \texttt{stable}\ |rel| \geqslant k \text{ (from lhs of } guarantees\text{), PSP}\}$
$\forall k :: |rel| \geqslant k \wedge NbR = k - 1 \mapsto NbR \neq k - 1$
$\Rightarrow \{\text{Using } \texttt{stable}\ NbR \geqslant k \text{ (from program text), PSP}\}$
$\forall k :: |rel| \geqslant k \wedge NbR = k - 1 \mapsto NbR \geqslant k$
$\Rightarrow \{\text{Induction over } n\text{, using the previous property}\}$
$\forall n, k :: |rel| \geqslant k \wedge NbR = n \mapsto NbR \geqslant k$
$\Rightarrow \{\text{Disjunction over } n\}$
$\forall k :: |rel| \geqslant k \mapsto NbR \geqslant k$

□

Lemma 50.

$$\forall k :: T \geqslant ask[|giv| + 1] \wedge |ask| > |giv| = k \mapsto |giv| > k \tag{50}$$

Proof. Similar as for (49). □

Lemma 51.

$$\texttt{always} \sum_{i=1}^{NbR} rel[i] \geqslant Tokens.giv \Rightarrow T \geqslant NbT \tag{51}$$

Proof. From $ask \nearrow$ and $rel \nearrow$ in the left-hand side of the *guarantees*, we can use the right-hand side of (48) (proved under the hypothesis $rel \nearrow$), from which the required property follows trivially. □

Proof (property (31)).

$\{\text{From lhs of } guarantees,\ giv \nearrow, \text{PSP, predicate calculus}\}$
$|giv| = n \wedge Tokens.giv = k \mapsto (|giv| = n \wedge Tokens.giv = k \wedge Tokens.rel \geqslant k) \vee |giv| > n$
$\Rightarrow \{\text{Lemma (49)}\}$
$|giv| = n \wedge Tokens.giv = k \mapsto (|giv| = n \wedge Tokens.giv = k \wedge \sum_{i=1}^{NbR} rel[i] \geqslant k) \vee |giv| > n$
$\Rightarrow \{\text{Lemma (51)}\}$
$|giv| = n \wedge Tokens.giv = k \mapsto (|giv| = n \wedge T \geqslant NbT) \vee |giv| > n$
$\Rightarrow \{\text{Disjunction over } k\}$
$|giv| = n \mapsto (|giv| = n \wedge T \geqslant NbT) \vee |giv| > n$
$\Rightarrow \{\text{lhs strengthening, } ask \nearrow, \text{PSP}\}$
$|ask| = k \wedge |giv| = n < k \mapsto (T \geqslant NbT \wedge |ask| > |giv| = n) \vee |giv| > n$
$\Rightarrow \{ask[|giv| + 1] \leqslant NbT, \text{ from lhs of } guarantees\}$
$|ask| = k \wedge |giv| = n < k \mapsto (T \geqslant ask[|giv| + 1] \wedge |ask| > |giv| = n) \vee |giv| > n$
$\Rightarrow \{\text{From lemma (50)}\}$
$|ask| = k \wedge |giv| = n < k \mapsto |giv| > n$
$\Rightarrow \{\text{Induction over } n\}$
$|ask| = k \mapsto |giv| \geqslant k$
$\Rightarrow \{\text{Disjunction}\}$
$|ask| \geqslant k \mapsto |giv| \geqslant k$
$\Rightarrow \{\text{Using (30)}\}$
$\forall h :: h \sqsubseteq ask \mapsto h \sqsubseteq giv$

□