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
{charpov,mani}@cs.caltech.edu

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?
- 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

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

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 570–589, 1999.

to software design 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. The formalism is extended with a theory of composition which relies on two forms of composition: *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 extend standard UNITY with a new temporal operator (*follows*) which allows us to represent asynchronous point-to-point communication at a logical level.

We obtain a simpler framework by restricting its expressive power: we do not deal with *all* temporal specifications, *all* forms of composition and *all* types of communication. However, we need to know if such a framework can be applied to a wide class of problems and if it can be used to develop simple 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. This paper describes a first step towards developing a simple compositional methodology for reactive systems.

The remainder of the paper is organized as follows. In the next section, we 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. One proof of each kind (refinement, composition and program text correctness) is given in the paper. All other proofs can be found in the technical report [11].

2 Framework

The framework we use is based on a UNITY-like logic and programming notation [5, 28, 27]. The traditional UNITY form of program composition (*union*) is used. However, at the logical level, we use the notions of *existential* and *universal* properties. A *guarantees* operator, that provides existential specifications, is introduced [6, 7]. We further extend UNITY with an abstraction of communication, based on temporal operators described in [12, 9, 13, 37].

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 the boolean: *Programs F and G can be composed.* 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 [28, 27]. However, when dealing with composition, one must be careful whether to use the *strong* or the *weak* form of these operators. The strong (sometimes called *inductive*) form is the one obtained when *wp* calculus is applied to statements *without* using the substitution axiom [5, 28, 27]. The weak form is either the one obtained when the substitution axiom is used or the one defined from the *strongest invariant* (which are equivalent) [34, 28, 27]. Strong operators are subscripted with $_s$ and weak operators are subscripted with $_w$. No subscript means that either form can be used. The strong form of an operator is logically stronger than the weak form. We give the definitions of temporal operators in terms of the program text:

$$
\begin{array}{ll|ll}
\texttt{init}\ p & \triangleq [\mathit{initially} \Rightarrow p] & p\ \texttt{next}_w\ q & \triangleq (SI \wedge p)\ \texttt{next}_s\ q \\
\texttt{transient}\ p & \triangleq \langle \exists c : c \in D : [p \Rightarrow wp.c.\neg p] \rangle & \texttt{stable}_w\ p & \triangleq p\ \texttt{next}_w\ p \\
p\ \texttt{next}_s\ q & \triangleq \langle \forall c : c \in D : [p \Rightarrow wp.c.q] \rangle & \texttt{always}\ p & \triangleq [SI \Rightarrow p] \\
\texttt{stable}_s\ p & \triangleq p\ \texttt{next}_s\ p & & \\
\texttt{invariant}\ p & \triangleq (\texttt{init}\ p) \wedge (\texttt{stable}_s\ p) & &
\end{array}
$$

where SI denotes the strongest invariant of the program (i.e. the conjunction of all invariants).

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

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

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).

$X \cdot F$ 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:

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

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. We consider three kinds of component 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 only by component F and *env* is a possible environment of F ($F * env$), then:

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

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

2.2 Composition

We use a compositional approach introduced in [6, 7], 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 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 [10] which is slightly different from the original definition in [7]:

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

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 any side of *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 [12, 9, 13, 37].

Definition 1 ($\boxleq$). *For any pair of* state expressions *(in particular variables) x and $`x$, and an order relation $\leqslant$, we define $`x \boxleq x$ (" $`x$* follows *x") with respect to $\leqslant$:*

$$`x \boxleq x \triangleq (x \nearrow_{\leqslant}) \wedge (`x \nearrow_{\leqslant}) \wedge (\texttt{always}\ `x \leqslant x) \ \wedge\ (\forall k :: k \leqslant x \ \mapsto\ k \leqslant `x)$$

Intuitively, $`x \boxleq x$ means that $x$ and $`x$ are nondecreasing and $`x$ is always trailing $x$ (wrt the order $\leqslant$), but $`x$ eventually increases (at least) to the current value of x.

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 "$Out \boxleq f.In$", while nondeterministic components use the "$f.Out \boxleq In$" form. In particular, network components (unbounded channels) are specified by "$Out \boxleq In$."

The following properties are referred to as "follows theorems":

$$\begin{array}{ll} `x \boxleq x \wedge f \ \textit{increasing function} & \Rightarrow f.`x \boxleq f.x \\ "x \boxleq `x \wedge `x \boxleq x & \Rightarrow "x \boxleq x \\ `x \boxleq x \wedge `y \boxleq y & \Rightarrow `x \cup `y \boxleq x \cup y \quad \text{(for sets or bags)} \end{array}$$

Notations on Histories

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

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

$|Seq|$ 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 between sequences, 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.

3 The Resource Allocation Example

3.1 The Different Steps of the Design

We shall design a resource allocation system. We require that all clients handle shared resources correctly.

In a first step, we specify what the clients are doing with respect to these resources (spec. $\langle 3 \rangle$) and what the correctness constraints of the system are (spec. $\langle 2 \rangle$). We deduce, in a systematic way, how a resource allocator should behave to provide that correctness (spec. $\langle 4 \rangle$). We then use a generic network specification (spec. $\langle 5 \rangle$) and develop a compositional proof to show that if all components satisfy their specifications, then the global system also satisfies its specification (proof C1 sect. 4.2).

Next, we design a resource allocator satisfying the previous specification. We build the general allocator by composing 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. $\langle 10 \rangle$), pick a generic merge component (spec. $\langle 6 \rangle$), a generic distributor component (spec. $\langle 7 \rangle$) and connect all these components with a network (spec. $\langle 8 \rangle$). 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. $\langle 9 \rangle$) and prove that it is actually obtained from the chosen components (proof C2 in [11]). We also prove that this allocator implements the allocator we needed (proof R1 sect. 5.2).

To complete the development, we have to design a client program and a single-client allocator program. Starting from the specifications we obtained ($\langle 3 \rangle$ and $\langle 10 \rangle$), 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. $\langle 11 \rangle$ and spec. $\langle 12 \rangle$), prove that the texts satisfy these specifications (proofs T1 sect. 6.3 and T2 in [11]) and, of course, that these specifications are stronger than requested (proofs R2 and R3 in [11]).

The different steps of this design are summarized in fig. 1. Each specification $\langle n \rangle$ is described in the corresponfing figure fig. n.

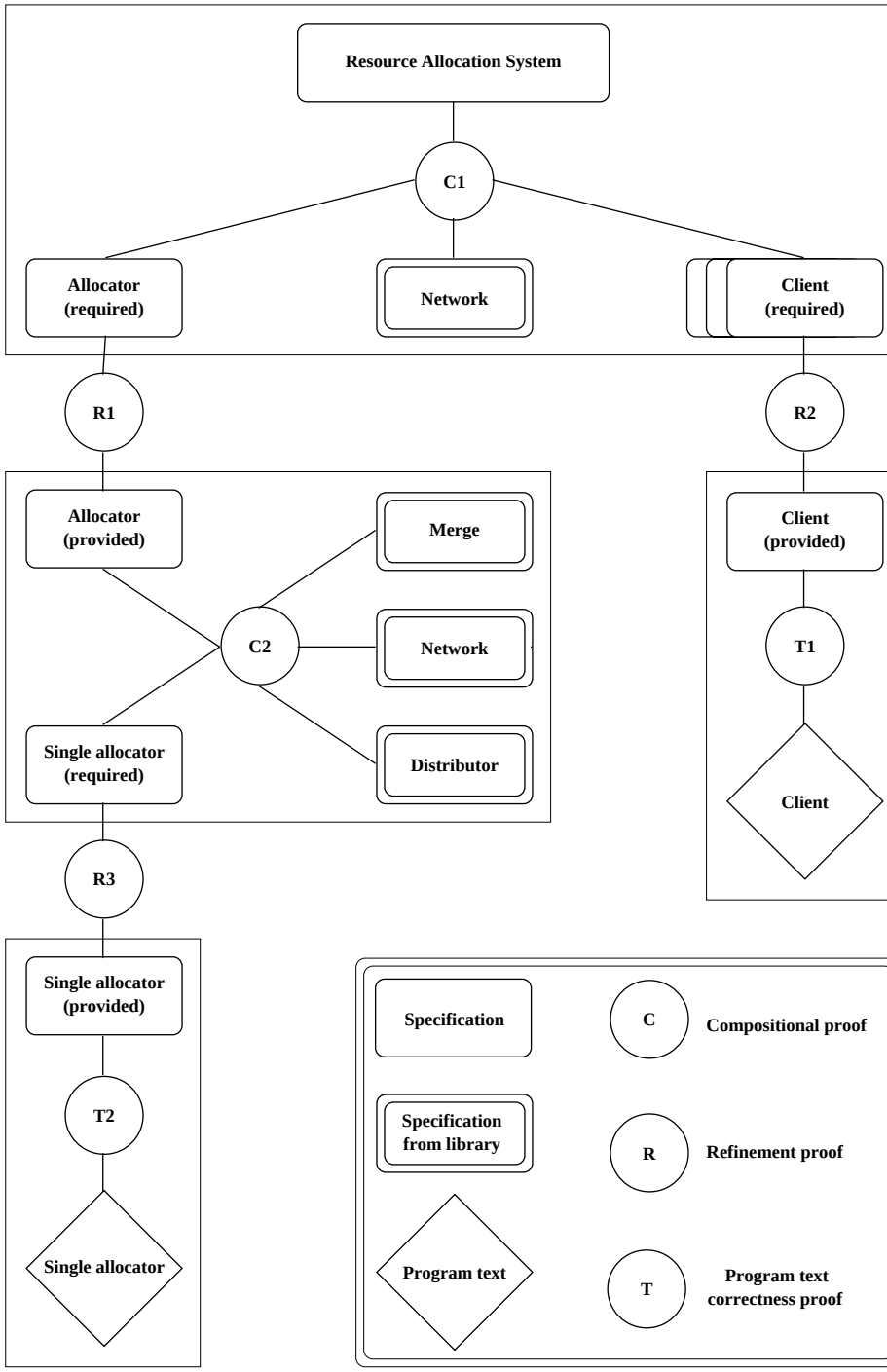


Fig. 1. General design.

3.2 Notations

Resources are modeled as tokens. All tokens are indivisible and identical. Tokens are not created or destroyed. All the messages exchanged between the resource allocator and its clients have the same type: integer (the number of tokens requested, given or released). $Tokens.h$ is the total number of tokens in history h (i.e. $Tokens.h = \sum_{k=1}^{|h|} h[k]$).

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 $Client_i.ask$ to $Alloc.ask_i$, from $Alloc.giv_i$ to $Client_i.giv$, and from $Client_i.rel$ to $Alloc.rel_i$. 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 expect from the resource allocation system is expressed in the traditional way. A safety property states that clients never share more tokens than there exists in the system. A liveness property guarantees that all client requests are eventually satisfied (Fig. 2).

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

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

Fig. 2. Resource allocation system specification.

The client's specification is also intuitive. The safety part is that clients never ask for more tokens than there exist. 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).

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

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

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

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 also appears in the specification (Fig. 4).

$$\mathit{true}\ \textbf{guarantees}\ \forall i :: \mathit{giv}_i \nearrow \tag{6}$$

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

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

Fig. 4. Allocator specification (required).

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

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

Fig. 5. Network specification.

4.2 Composition Proof

The following two proofs are built as chains. Because they are existential, *guarantees* properties hold in the global system. Then, applying *follows* theorems to the right-hand side of a *guarantees*, we deduce a property that can be injected into the left-hand side of another *guarantees* property, which allows us to use the right-hand side of this new *guarantees* property, and so on until we reach the desired global correctness property.

Property (1).

Proof. In the resource allocation system:

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

□

Property (2).

Proof. In the resource allocation system:

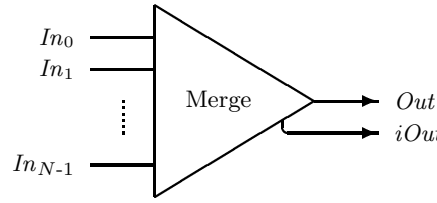
$$
\begin{array}{l}
\quad \{\text{Specification (3)}\} \\
\quad \forall i :: \mathit{Client}_i.\mathit{ask} \nearrow \ \wedge \ \mathit{Client}_i.\mathit{rel} \nearrow \\
\Rightarrow \{\text{Specification (9), follows definition}\} \\
\quad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\Rightarrow \{\text{Specification (4)}\} \\
\qquad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\quad \wedge \ \forall i :: \texttt{always} \ \langle \forall k :: \mathit{Client}_i.\mathit{ask}[k] \leqslant \mathit{NbT} \rangle \\
\Rightarrow \{\text{Specification (9), follows definition, calculus}\} \\
\qquad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\quad \wedge \ \texttt{always} \ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\Rightarrow \{\text{Specification (6)}\} \\
\qquad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\quad \wedge \ \texttt{always} \ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\quad \wedge \ \forall i :: \mathit{Alloc}.\mathit{giv}_i \nearrow \\
\Rightarrow \{\text{Specification (9), follows definition}\} \\
\qquad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\quad \wedge \ \texttt{always} \ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\quad \wedge \ \forall i :: \mathit{Client}_i.\mathit{giv} \nearrow \\
\Rightarrow \{\text{Specification (5)}\} \\
\qquad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\quad \wedge \ \texttt{always} \ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\quad \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}.\mathit{Client}_i.\mathit{rel} \geqslant \\
\qquad \mathit{Tokens}.h_i \\
\Rightarrow \{\text{Specification (9), follows definition, transitivity of } \mapsto \ \} \\
\qquad \forall i :: \mathit{Alloc}.\mathit{ask}_i \nearrow \ \wedge \ \mathit{Alloc}.\mathit{rel}_i \nearrow \\
\quad \wedge \ \texttt{always} \ \langle \forall i, k :: \mathit{Alloc}.\mathit{ask}_i[k] \leqslant \mathit{NbT} \rangle \\
\quad \wedge \ \forall i, h :: h_i \sqsubseteq \mathit{Alloc}.\mathit{giv}_i \wedge h_i \sqsubseteq_{\geqslant} \mathit{Alloc}.\mathit{ask}_i \ \mapsto \ \mathit{Tokens}.\mathit{Alloc}.\mathit{rel}_i \geqslant \\
\qquad \mathit{Tokens}.h_i \\
\Rightarrow \{\text{Specification (8)}\} \\
\quad \forall i, h :: h \sqsubseteq \mathit{Alloc}.\mathit{ask}_i \ \mapsto \ h \sqsubseteq_{\leqslant} \mathit{Alloc}.\mathit{giv}_i \\
\Rightarrow \{\text{Specification (9), follows definition}\} \\
\quad \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 fair: No input channel can be ignored indefinitely. A merge component is nondeterministic.



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

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

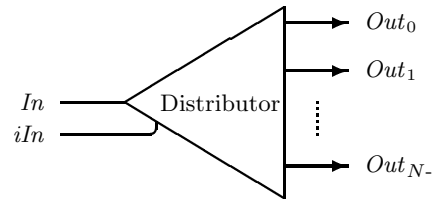
$$\textit{true}\ \textbf{guarantees}\ \texttt{always}\ \langle \forall k :: 0 \leqslant iOut[k] < N \rangle \tag{12}$$

$$\begin{array}{c} \forall i :: In_i \nearrow \\ \textbf{guarantees} \\ \forall i :: Out/\{k \mid 1 \leqslant k \leqslant |iOut| \wedge iOut[k] = i\} \boxleq In_i \end{array} \tag{13}$$

Fig. 6. Merge specification.

The main merge specification is (13). This specification only constrains values when indexes are present (and indexes when values are present): it requires that there is no value without the corresponding index, and no index without the corresponding value (11). Moreover, specification (13) does not constrain values corresponding to nonvalid indexes. So, we require that there are no nonvalid indexes (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 dual of the merge: If the k-th value in iIn is j, then the distributor outputs the k-th value of In to Out_j, provided that j is a correct index, i.e. $0 \leqslant j < N$. A distributor component is fair: If indexes are present, the input channel cannot be ignored indefinitely. A distributor component is deterministic.



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

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.

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).

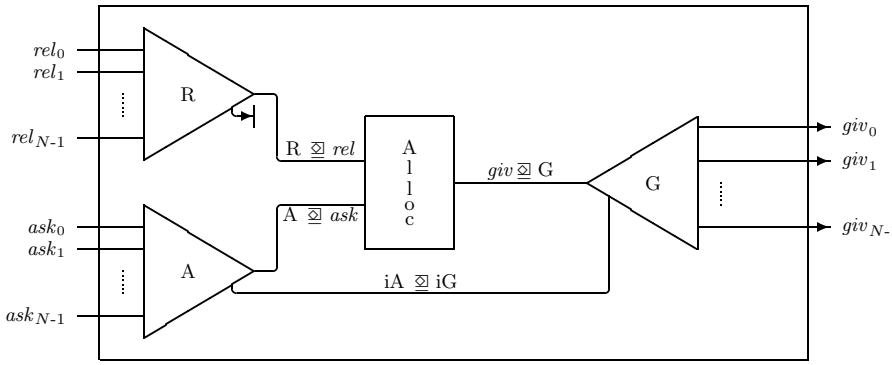


Fig. 8. General allocator architecture and network specification.

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 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).

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

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

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

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).

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

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

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

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

5.2 Refinement Proof

We need to show that the provided specification ⟨9⟩ is stronger than 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 than the left-hand-side of (17).

Proof.

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

□

The composition proof, i.e. the proof that the components connected as described in fig. 8 provide the general allocator specification ⟨9⟩ is given in [11].

6 Clients

6.1 Model

A client has a variable T which represents the size of the next request. We do not specify how T is chosen. However, its value must always be in the range $1..NbT$. In the following program, we use the nondeterminism of UNITY programs to generate a random value for T. 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 ($\langle 11 \rangle \Rightarrow \langle 3 \rangle$) is given in [11].

$$true \textbf{ guarantees } ask \nearrow \; \wedge \; rel \nearrow \tag{21}$$

$$true \textbf{ guarantees } \texttt{always} \; \langle \forall k :: ask[k] \leqslant NbT \rangle \tag{22}$$

$$giv \nearrow \textbf{ guarantees } \forall h :: h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \;\mapsto\; h \sqsubseteq rel \tag{23}$$

Fig. 11. Client specification (provided).

6.3 Correctness Proof

Property (21). Properties in the right-hand side of the *guarantees* are strongly satisfied by the client text (i.e. the corresponding strong property holds). Because they only refer to local variables, they also hold strongly in other components. Then, since strong *stable* properties are universal, they hold in the global system.

Property (22). The same argument is applied, using 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 (23).

Lemma 24.

$$\forall h, k :: \texttt{transient}\ rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \cdot Client \tag{24}$$

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

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

□

Proof (specification (23)). In any composed system:

$$\begin{array}{l}
\quad \{\text{From lemma (24), existentiality of } \textit{transient}\} \\
\qquad\qquad (rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask) \\
\qquad \mapsto \ \neg(rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask) \\
\Rightarrow \{giv \nearrow \text{ from lhs, } ask \nearrow\text{, PSP}\} \\
\quad rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ rel \neq k \\
\Rightarrow \{rel \nearrow\text{, PSP}\} \\
\quad rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ k \sqsubset rel \\
\Rightarrow \{\text{Induction on } |h| - |k|\} \\
\quad rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ rel = h \\
\Rightarrow \{\text{Weakening}\} \\
\quad rel = k \wedge k \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ h \sqsubseteq rel \\
\Rightarrow \{\text{Disjunction over } k\} \\
\quad rel \sqsubset h \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ h \sqsubseteq rel \\
\Rightarrow \{\text{Disjunction } [(p \wedge r \ \mapsto \ q) \Rightarrow ((p \vee q) \wedge r \ \mapsto \ q)]\} \\
\quad (rel \sqsubset h \vee h \sqsubseteq rel) \wedge h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ h \sqsubseteq rel \\
\Rightarrow \{\texttt{always}\ rel \sqsubseteq giv \text{ holds in system, hence } h \sqsubseteq giv \Rightarrow (rel \sqsubset h \vee h \sqsubseteq rel)\} \\
\quad h \sqsubseteq giv \wedge h \sqsubseteq_{\geqslant} ask \ \mapsto \ h \sqsubseteq rel
\end{array}$$

□

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 are 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 $\langle 10 \rangle$.

The main difference is that this allocator waits for a request before it sends tokens (27). This is not explicitly required in specification $\langle 10 \rangle$. 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 (28). 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.

$$true \ \textbf{guarantees} \ giv \nearrow \tag{25}$$

$$rel \nearrow \ \textbf{guarantees} \ \texttt{always} \ Tokens.giv - Tokens.rel \leqslant NbT \tag{26}$$

$$ask \nearrow \ \textbf{guarantees} \ \texttt{always} \ giv \sqsubseteq ask \tag{27}$$

$$\begin{array}{c} ask \nearrow \ \wedge \ rel \nearrow \\ \wedge \qquad \texttt{always} \ \langle \forall k :: ask[k] \leqslant NbT \rangle \\ \wedge \ \forall k :: Tokens.giv \geqslant k \ \mapsto \ Tokens.rel \geqslant k \\ \textbf{guarantees} \\ \forall h :: h \sqsubseteq ask \ \mapsto \ h \sqsubseteq giv \end{array} \tag{28}$$

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

The refinement proof ($\langle 12 \rangle \Rightarrow \langle 10 \rangle$) and the program text correctness proof are given in [11].

8 Conclusions

The allocator example illustrates the need, when adopting a component-based design, to switch between top-down and bottom-up approaches.

Designers are given the specification of the global system that they are designing. They propose components which can be composed to obtain the desired system. Some of the components may have been implemented earlier and may be available in a component library. Other components must be designed. These components can be implemented as compositions of other components or directly as programs. This approach is the traditional top-down approach. While building these components (by programming, or by composition), designers adopt the point of view of providers: they look at what they get and express it logically. This corresponds to a bottom-up phase.

Provided specifications and required specifications 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. 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.

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 a communication abstraction (*follows*), is able to handle those three types of proofs, while remaining in the minimalist spirit of UNITY.

Our goal is to develop methods for building distributed systems from components. It is important to be able to specify components in terms of input and output, and to be able to connect them formally through simple proofs. Combining a temporal operator like *follows* with some basic sequence properties seems

an interesting approach. We are currently investigating how far this approach can go. 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 [10].

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* [33] 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.

9 Related Work

Many people have studied methods for designing systems by composition of components [1, 2] and by systematic refinement [3, 4, 16, 35]. A significant amount of work has been done on compositional methods within the UNITY framework [14, 15, 28, 27]. Composition using rely/guarantee properties and systematic specifications of interfaces have been proposed by Jones [19, 21, 22, 23, 36]. This paper differs from most of the earlier work in developing the theory of universal and existential properties and the *guarantees* operator proposed in [7].

Likewise, compositional software architectures have been covered in great detail with the software engineering community. Excellent overviews of the software process [24], technology [39] and the academic state of the art [31] are available. Existing solutions primarily focus on software engineering processes (e.g. semi-formal interface specifications [25, 18]) and language- and architecture-based approaches ([26, 30] and [8, 32, 38], respectively). Only recently has work begun to incorporate theoretically grounded compositional concepts into practice [29, 17, 37, 20].

References

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

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

[3] R. Back. Refinement calculus, Part I: Sequential nondeterministic programs. In *REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1989.

[4] R. Back. Refinement calculus, Part II: Parallel and reactive programs. In *REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1989.

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

[6] K. M. Chandy and B. A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.

[7] K. M. Chandy and B. A. Sanders. Reasoning about program composition. Technical Report 96-035, University of Florida, Department of Computer and Information Science and Engineering, 1996.

[8] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[9] M. Charpentier. *Assistance à la Répartition de Systèmes Réactifs*. PhD thesis, Institut National Polytechnique de Toulouse, France, Nov. 1997.

[10] M. Charpentier and K. M. Chandy. Examples of program composition illustrating the use of universal properties. In J. Rolim, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1215–1227. Springer-Verlag, Apr. 1999.

[11] M. Charpentier and K. M. Chandy. Towards a compositional approach to the design and verification of distributed systems. Technical Report CS-TR-99-02, California Institute of Technology, Jan. 1999. 29 pages.

[12] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Abstracting communication to reason about distributed algorithms. In Ö. Babaoğlu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms (WDAG'96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, October 1996.

[13] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Tailoring UNITY to distributed program design. In J. Rolim, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98)*, volume 1388 of *Lecture Notes in Computer Science*, pages 820–832. Springer-Verlag, April 1998.

[14] P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, 1994.

[15] P. 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.

[16] P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[17] D. Garlan. Higher-order connectors. In *Proceedings of Workshop on Compositional Software Architectures*, Monterey, California, Jan. 1998.

[18] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *European Conference on Object-Oriented Programming/ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 25/10, pages 169–180, 1990.

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

[20] J. R. Kiniry. CDL: A component description language. In *Proceedings of the COOTS '99 Advanced Topics Workshop on Validating the Composition/Execution of Component-Based Systems*, 1999.

[21] S. Lam and A. Shankar. Specifying modules to satisfy interfaces–a state transition approach. *Distributed Computing*, 6(1):39–63, July 1992.

[22] S. Lam and A. Shankar. A theory of interfaces and modules 1: Composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, Jan. 1994.

[23] R. Manohar and P. Sivilotti. Composing processes using modified rely-guarantee specifications. Technical Report CS-TR-96-22, California Institute of Technology, 1996.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 2nd edition, 1988.

[25] B. Meyer. Applying design by contract. *IEEE Computer*, Oct. 1992.

[26] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[27] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[28] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[29] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, Sept. 1992.

[30] O. Nierstrasz and T. Meijler. Requirements for a composition language. In *ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, pages 147–161. Springer–Verlag, 1995.

[31] O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, Inc., 1995.

[32] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0*. Object Management Group (OMG), 2.0 edition.

[33] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[34] B. A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.

[35] B. A. Sanders. Data refinement of mixed specification: A generalization of UNITY. *Acta Informatica*, 35(2):91–129, 1998.

[36] N. Shankar. Lazy compositional verification. In *Compositionality: The Significant Difference. International Symposium, COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[37] P. 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, Dec. 1997.

[38] Sun Microsystems, Inc. JavaBeans API specification, version 1.01. Technical report, Sun Microsystems, Inc., July 1997.

[39] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, 1997.