An Approach to Composition Motivated by *wp*[⋆]

Michel Charpentier

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

Abstract. We consider the question of composition in system design, a fundamental issue in engineering. More precisely, we are interested in deducing system properties from components properties and vice-versa. This requires system and component specifications to be "*compositional*" in some sense. Depending on what systems are and how they are composed, this problem is satisfactorily solved (e.g., sequential composition of terminating programs) or remains a hot research topic (e.g., concurrent composition of reactive systems). In this paper, we aim at providing a logical framework in which composition issues can be reasoned about independently from the kind of systems and the laws of composition under consideration. We show that many composition related statements can be expressed in terms of predicate transformers in a way that presents interesting similarities with program semantics descriptions based on weakest precondition calculus.

1 Motivation

System designers, whether those systems are software, hardware, planes or buildings, are all faced with a common issue: How to compose systems from components and how to partition systems into components. Compositional design offers the hope of managing complexity by avoiding unnecessary details: Systems designers prove properties of systems given properties, but not detailed implementations, of components.

As fundamental as it is, the question of composition is far from being solved in a fully satisfactory way. What does it mean for a formal framework to be "*compositional*"? Are there "*good*" and "*bad*" compositional frameworks? What characterizes "*good*" compositional specifications? Should system designers and components providers use the same kind of specifications? Are there fundamental laws of composition that are common to most areas of engineering? How are composition and reuse issues related? We believe these to be important questions one must (at least partially) answer before suitable solutions to the composition problem are found.

⋆ *5th International Conference on Fundamental Approaches to Software Engineering (FASE'2002)*, April 2002, LNCS 2306, pages 1–14.

In this paper, we aim at providing a basis that can be used to reason about composition. We seek a logical framework in which many of the questions mentioned above can be formulated and reasoned about. We try to keep the context of our approach as general as possible. Especially, we don't want to decide from the start what systems are, how they can be composed and what specification languages are used to describe them. Instead, we assume generic binary laws of composition and we later introduce specific properties, such as for instance associativity, symmetry or idempotency. In other words, the basis of our framework can be applied to almost any form of composition and any type of system, but more relevant and useful properties can be deduced when additional hypotheses are introduced.

We restrict ourselves to logical properties on systems. More precisely, system properties are predicates on these systems. We use the terms *property* and *specification* interchangeably, the only slight nuance being that specifications are usually provided while properties may have to be deduced.

Since in this context components and systems are seen only through their logical properties, it is convenient to identify systems and system properties and to describe composition at a purely logical level, where everything is expressed in terms of specifications without mentioning components and systems. These specifications behave in different ways when they are composed (compositional versus non compositional specifications). In other words, specifications are *transformed* through composition. They may also be transformed beforehand to give "better" compositional specifications. As a consequence, our approach relies heavily on *predicate transformers* (functions from predicates to predicates). Our transformers present interesting similarities with Dijkstra's *weakest precondition* and *strongest postcondition*, from which we can inherit useful intuition.

The remainder of the paper is organized as follows. Section 2 introduces the basic elements of our framework, namely components, composition, systems and system properties. What it means for specifications to be compositional is precisely stated in section 3. In section 4, we define a family of "*strongest transformers*" which can be used to describe how specifications are transformed when systems are composed. At this point, it is explained how systems and their specifications can be identified. Section 5 introduces another family of predicate transformers ("*weakest transformers*") which are used to transform non compositional specifications into compositional ones. In section 6, we compare our transformers with *strongest postcondition* and *weakest precondition* transformers. We show in section 7 how to switch between the bottom-up and top-down aspects of the composition problem by using conjugates of predicate transformers. Section 8 instantiates previously defined notions with a more specific context (a single law of composition over a monoid structure). Finally, section 9 shows how this approach to composition can be applied in the context of concurrent and reactive systems.

2 Components, Composition, Systems and System Properties

We assume the existence of atomic[1] components as well as laws of composition. When applied to components, these laws lead to systems. Systems, in turn, can be composed to provide us with new systems. We only consider systems that involve a finite number of applications of the laws of composition. However, that number may be zero and, by extension, single components are also considered to be systems.

We denote components and systems with capital letters F, G, H, etc. Composition laws are represented by infix binary operators such as $\circ$ or $\star$. We do not assume any property on these operators, such as (mutual) associativity, symmetry, idempotency or distributivity. For example, if F, G and H are systems, so are $F{\circ}G$, $G{\star}H$ and $(F{\circ}G){\star}H$.

Some compositions may not be possible (i.e., the resulting system may not exist). For a given law of composition, different definitions of "possible" may be used. For instance, we can allow parallel composition of processes that share variables, or forbid it and request that processes send and receive messages instead. In both cases, the nature of composition is similar (say, fair interleaving of atomic transitions) and it would be awkward to rely on two different laws of composition. Our approach to this situation is to use a compatibility operator to denote that a given composition is possible. The same law of composition, such as concurrent composition of processes, can be coupled to different compatibility operators to allow shared variables or not, for instance. In order not to clutter up our notations, we do not use compatibility operators explicitly here (but we do in [15, 14]). Instead, it is implicitly understood that systems that are referred to exist. For instance, in a formula of the form $\langle \forall F, G : \cdots : \cdots F{\circ}G \cdots \rangle$, the range of the universal quantification is implicitly reduced to those systems F and G for which $F{\circ}G$ exists.

System properties are point-wise predicates on systems (i.e., functions from systems to booleans) or, equivalently, sets of systems (although, in this paper, we stick to predicate notations). We denote them with capital letters X, Y, Z, S, T, etc. Function application is denoted with a dot (.) which has higher precedence than boolean operators but lower precedence than composition operators and which associates to the left (i.e., $a.b.c = (a.b).c$). For example, if F and G are systems and X is a property, $X.F{\circ}G$ is the boolean: "property X holds for system $F{\circ}G$". A system that satisfies property X is called an "X-system".

3 Compositional Specifications

A core issue in compositional design is to relate systems properties and subsystems properties. Designers should be able to deduce a system property from

[1] *Atomic* doesn't mean that a component cannot be decomposed. It simply means that its internal structure is not considered here.

subsystems properties (i.e., use components) and subsystems properties from a system property (i.e., develop/find components). The ability to deduce subsystems properties from a system specification is an aspect of the problem that is sometimes ignored. However, we believe it to be as important an issue as the other aspect, since system designers have to identify suitable components in order to build their system. In other words they have to deduce relevant component properties from the given system specification.

In this paper, we focus our attention on the problem of deducing systems properties from subsystems properties, except in section 7 where we show how the other hand of the question can be reached through conjugates of predicate transformers. This way, we avoid presenting symmetric but similar arguments all along the paper but instead we explain how top-down arguments can be deduced in a systematic way from corresponding bottom-up arguments.

With the bottom-up view in mind, we say that a list of properties is *compositional* (with respect to a specific pattern of composition such as $\_\circ(\_\star\_)$) if the last property of the list holds in any system built (by applying the given pattern) from components that satisfy the other properties of the list. For instance, if we consider systems of the form $(F{\circ}G){\star}H$[2], the list of properties (X, Y, Z, T) is compositional if and only if:

$$\langle \forall F, G, H :: X.F \wedge Y.G \wedge Z.H \Rightarrow T.(F{\circ}G){\star}H \rangle \ . \tag{1}$$

Compositional lists of properties allow us to deduce system properties from subsystem properties. In some sense, they are a schematic representation of the composition problem.

We do not introduce a specific syntax to represent compositional lists of properties. Instead, we show in the next section how formula (1) can be expressed without making reference to systems and a property-only syntax is introduced.

4 Components and Systems as Predicates

We consider the following equation in predicates, where S is the unknown:

$$S \ : \ \langle \forall F, G, H :: X.F \wedge Y.G \wedge Z.H \Rightarrow S.(F{\circ}G){\star}H \rangle \ . \tag{2}$$

It is clear that any (finite or infinite) conjunction of solutions of (2) is itself a solution of (2). Therefore, given properties X, Y and Z, equation (2) has a strongest solution [20]. We denote this solution by $(X{\circ}Y){\star}Z$. This notation generalizes to other forms of composition and $X{\circ}Y$ or $(X{\star}Y){\circ}(Z{\star}T)$, for instance, are also properties.

These properties, defined in terms of the strongest solution of some compositional equation, all enjoy a similar property, expressed by the following proposition:

[2] We use systems such as $(F{\circ}G){\star}H$ throughout the paper to illustrate our predicates and predicate transformers definitions. However, definitions are general and can be applied to any other form of composition, such as $(F \diamond (G \uparrow H)) \downarrow K$, whatever $\diamond$, $\uparrow$ and $\downarrow$ are.

Proposition 1 *For any system K:*

$$(X \circ Y) \star Z \;.\; K \;\equiv\; \langle \exists F, G, H : X.F \wedge Y.G \wedge Z.H : (F \circ G) \star H = K \rangle \;.$$

In other words, $(X \circ Y) \star Z$ characterizes those systems that can be obtained by composing an X-system, a Y-system and a Z-system through $\circ$ and $\star$ (in the right order). Any system composed this way satisfies $(X \circ Y) \star Z$ and any system that satisfies $(X \circ Y) \star Z$ can be decomposed this way.

Using proposition 1, we can rewrite the formula that was used to define compositional lists of properties. Formula (1) is equivalent to:

$$\langle \forall K :: (X \circ Y) \star Z.K \Rightarrow T.K \rangle \;,$$

which, by making use of the "everywhere operator" $[\cdots]$ [20], can be written:

$$[(X \circ Y) \star Z \;\Rightarrow\; T] \;.$$

From this point on, we do not have to refer to systems explicitly and we express all what we need in term of properties. Note that we can always reintroduce systems by using specific properties that are characteristic of one system exactly (singletons). For a system F, we define the property $F_=$ by:

$$\langle \forall G :: F_= \;.\; G \;\equiv\; (F = G) \rangle \;.$$

In other words, $F_=$ is the property "*to be the system F*". Thanks to the notation we have introduced, the mapping from F to $F_=$ is an isomorphism that preserves the structure of the composition, i.e.,

$$[((F \circ G) \star H)_= \;\equiv\; (F_= \circ G_=) \star H_=] \;.$$

Y and Z being fixed, $(X \circ Y) \star Z$ describes what becomes of property X when an X-system is composed (with two other systems that satisfy Y and Z). If, for instance, $(X \circ Y) \star Z$ is equivalent to X (for some Y and Z), it means that X has a "good" compositional behavior (it is entirely preserved when composed with Y and Z systems). $(X \circ Y) \star Z$ can also be stronger than X, in which case property X is "enriched" through composition with Y and Z. Sometimes, however, $(X \circ Y) \star Z$ will be weaker than X, meaning that only a part of property X survives composition. In the worst case, $(X \circ Y) \star Z$ reduces to *true*: all of X is lost. In other words, the predicate transformer $(\lambda X \cdot (X \circ Y) \star Z)$ tells us what happens to property X when composed in a specific context.

5 "*Weakest property*" Transformers

In this section, we define "*weakest property*" transformers that allow us to characterize adequate subsystems in a composition.

Given properties Y, Z and T, the equation (in predicate):

$$S \;:\; [(S \circ Y) \star Z \;\Rightarrow\; T] \tag{3}$$

has a weakest solution (the disjunction of all solutions is itself a solution). By the axiom of choice, there is a function that maps T to this weakest solution (Y and Z being fixed). We denote that function by $(?{\circ}Y){\star}Z$. Therefore, the weakest solution of (3) is $((?{\circ}Y){\star}Z).T$. Other functions (such as $(X{\circ}?){\star}Z$ and $(X{\circ}Y){\star}?$) are defined similarly.

Our notation may seem ambiguous at first sight. Is $(?{\circ}(Y{\star}Z)).T$ the weakest solution of:

$$S \ : \ \langle \forall F, G, H :: S.F \wedge Y.G \wedge Z.H \Rightarrow T.F{\circ}(G{\star}H) \rangle \ ,$$

or the weakest solution of:

$$S \ : \ \langle \forall F, G :: S.F \wedge (Y{\star}Z).G \Rightarrow T.F{\circ}G \rangle \ ?$$

Fortunately, thanks to proposition 1, both equations are equivalent.

It should be emphasized that $(?{\circ}Y){\star}Z$ is a function from properties to properties (i.e., a predicate transformer) while $(X{\circ}Y){\star}Z$ is a property. Furthermore, $(?{\circ}Y){\star}Z$ is *not* the transformer $(\lambda X \cdot (X{\circ}Y){\star}Z)$ described above.

An interesting property of $((?{\circ}Y){\star}Z).T$ is that it characterizes the systems that lead to a T-system when they are extended by a Y-system and a Z-system using $\circ$ and $\star$ (in the right order). This is expressed by the following proposition:

Proposition 2 *For any system K:*

$$((?{\circ}Y){\star}Z).T \ . \ K \ \equiv \ \langle \forall F, G : Y.F \wedge Z.G : T.(K{\circ}F){\star}G \rangle \ .$$

$((?{\circ}Y){\star}Z).T$ is the answer to the question: "What do I have to prove on my component to ensure that, when it is composed with Y and Z components, the resulting system always satisfies T?" If T is a property which already enjoys a nice compositional behavior (for instance, it is such that $[(T{\circ}Y){\star}Z \Rightarrow T]$), it may be enough to verify that the component satisfies T. If, however, T is not one of those properties that compose well, we may have to prove a property stronger than T itself to ensure that at least T will be preserved when the component is composed with Y and Z systems. In the worst case, there may be no way to guarantee that the component will provide the property T when composed with Y and Z systems, in which case $((?{\circ}Y){\star}Z).T$ reduces to *false*.

6 Relationship with *wlp* and *sp*

The approach used in previous sections is very similar to Dijkstra's *wlp* and *sp* [20]. We can define a correspondence between a "*program semantics*" world and a "*composition*" world:

$$\begin{array}{lcl} \text{state (predicate)} & \leftrightarrow & \text{system (property)} \ , \\ \text{run a program} & \leftrightarrow & \text{build a system} \ , \\ \text{assign a variable} & \leftrightarrow & \text{add a subsystem} \ . \end{array}$$

Then, the "?" transformers we have just defined correspond to *wlp*, the weakest liberal precondition:

$wlp.s.q$: weakest state predicate such that, if statement s is executed from such a state (and s terminates), the resulting state satisfies q.

$(?{\circ}F_=).T$: weakest system property such that, if system F is added (on the right, through $\circ$) to such a system, the resulting system satisfies T.

Because in our approach everything is property and systems are just a special case, the formulation can be generalized:

$(?{\circ}Y).T$: weakest system property such that, if a Y-system is added (on the right, through $\circ$) to such a system, the resulting system satisfies T.

It can also be shown that, like *wlp*, $(?{\circ}Y){\star}Z$ (and other similar transformers) is universally conjunctive, hence monotonic.

On the other hand, the function $(\lambda X \cdot X{\circ}Y)$ corresponds to *sp*, the strongest postcondition:

$sp.s.p$: strongest state predicate that holds after statement s is executed from a state that satisfies p.

$X{\circ}Y$: strongest property that holds after a Y-system is added (on the right, through $\circ$) to a system that satisfies X.

As the (partial) correctness of a Hoare triple can be expressed equivalently in terms of *wlp* or in terms of *sp*:

$$[p \Rightarrow wlp.s.q] \;\equiv\; [sp.s.p \Rightarrow q] \;\; ,$$

"composition correctness" can be expressed equivalently in terms of $?{\circ}Y$ or in terms of $(\lambda X \cdot X{\circ}Y)$:

$$[X \Rightarrow (?{\circ}Y).Z] \;\equiv\; [X{\circ}Y \Rightarrow Z] \;\; .$$

In the same way, the well known rule of *wlp* for sequential composition of programs:

$$[wlp.(s;s').q \;\equiv\; wlp.s.(wlp.s'.q)]$$

also holds for property transformers:

$$[((?{\circ}Y){\star}Z).T \;\equiv\; (?{\circ}Y).((?{\star}Z).T)] \;\; .$$

7 Using Conjugates of Predicate Transformers

So far, we have considered the question of deducing system properties from subsystems properties. As we mentioned earlier, we believe that the converse problem, to deduce subsystems properties from system properties, is equally important.

Every predicate transformer $\mathcal{T}$ has a unique conjugate $\mathcal{T}^*$ defined by $[\mathcal{T}^*.X \equiv \neg\mathcal{T}.(\neg X)]$. Therefore, the transformer $(?{\circ}Y){\star}Z$ has a conjugate $((?{\circ}Y){\star}Z)^*$ such that $((?{\circ}Y){\star}Z)^*.T$ characterizes the remainder of a T-system partially built from a Y-system and a Z-system. In other words, it is the answer to the question: "I am building a T-system; I already know that I am going to use a Y-system and a Z-system and I am looking for a component to complete my design; what do I know about this missing component?" So, $((?\circ Y)\star Z)^*.T$ guides us into finding a suitable component to complete the design of a T-system when there exist a constraint that a Y-system and a Z-system must be used. Note that this is only a necessary condition: such a component does not guarantee that the resulting system will satisfy property T. For that, we need that the chosen component also satisfies $((?\circ Y)\star Z).T$.

In the same way, we can consider the conjugate of $(\lambda X \cdot (X\circ Y)\star Z)$. The property $(\lambda X \cdot (X\circ Y)\star Z)^*.T$ represents what has to be proved on a system partially built with Y and Z components in order to ensure that the remaining part satisfies T.

8 Special Case: One Law of Composition

Properties such as $(X\circ Y)\star Z$ or $((?\circ Y)\star Z).T$ make the structure of composition explicit: what operators are used, how many of them and where, all this appear in the writing of the formula.

The next step in our approach is then to "quantify over the structure", i.e., make the structure implicit. This involves introducing generic forms of composition from which, hopefully, the general case can still be reached.

We have started to explore this path in a limited context where a single composition operator $\circ$ is involved. Furthermore, we assume that this operator is associative and has an identity element (monoid). We do not assume other properties such as symmetry or idempotency. Note that these hypotheses make it possible for the composition operator to be either parallel composition ($\|$) or sequential composition (;) of programs, but does not allow us to describe systems where both operators are interleaved. Several predicate transformers have been defined and studied in [15, 14]. Here, we show how they can be seen as special cases of our weakest and strongest transformers.

In this context, we focus on two generic forms of composition called *existential* and *universal*. Existential and universal are compositional characteristics of properties. A property X is existential (denoted by $\mathit{exist}.X$) if and only if X holds for any system in which one component at least satisfies X. A property X is universal (denoted by $\mathit{univ}.X$) if and only if X holds in any system in which all components satisfy X [9]. Formally:

$$\begin{aligned}\mathit{exist}.X &\triangleq \langle \forall F, G :: X.F \vee X.G \Rightarrow X.F\circ G\rangle \ ,\\ \mathit{univ}.X &\triangleq \langle \forall F, G :: X.F \wedge X.G \Rightarrow X.F\circ G\rangle \ .\end{aligned}$$

As before, we can choose not to use explicit quantification over components and express *exist* and *univ* in terms of predicates only:

$$\begin{aligned}\mathit{exist}.X &= [X\circ \mathit{true} \Rightarrow X] \wedge [\mathit{true}\circ X \Rightarrow X]\\ &= [(X\circ \mathit{true}) \vee (\mathit{true}\circ X) \Rightarrow X]\\ &= [\mathit{true}\circ X\circ \mathit{true} \Rightarrow X] \ ,\\ \mathit{univ}.X &= [X\circ X \Rightarrow X] \ .\end{aligned}$$

Some properties are "naturally" existential or universal (which is one reason why we decided to focus on these types of composition). For instance, some fundamental properties in the UNITY logic [25, 24] are existential or universal (see section 9). However, in the same context, some useful properties (such as *leads-to*) are neither existential nor universal.

One way to deal with properties that do not enjoy existential or universal characteristics is to *transform* them into existential and universal properties, by strengthening or weakening them. We can for instance define a predicate transformer $\mathbb{WE}$ such that $\mathbb{WE}.X$ is the weakest existential property stronger than X (such a property always exists because disjunctions of existential properties are existential) [15]. Then, systems can be specified in terms of $\mathbb{WE}.X$ instead of X so that, when they are composed with other systems, the property X is preserved. In other words, a stronger component (a $\mathbb{WE}.X$-component instead of an X-component) is developed to make later compositions easier (property X holds in any system that uses such a component). The idea is to keep the property as weak as possible and to compensate for the extra amount of effort required when designing a component by reusing it in several systems, where we benefit from the fact that composition with an existential property leads to simpler proofs.

Formally, $\mathbb{WE}$ can be related to "?" transformers:

$$[\mathbb{WE} \;\equiv\; \mathit{true} \circ {?} \circ \mathit{true}] \;.$$

Proposition 3 *For any property X, there exists a weakest existential property stronger than X and it is $(\mathit{true} \circ {?} \circ \mathit{true}).X$.*

In other words, using a formulation with an explicit quantification over systems, the weakest existential property stronger than X characterizes those systems that bring the property X to any system that contains them:

$$\begin{array}{c}\text{“}F \text{ satisfies the weakest existential property stronger than } X\text{”} \\ \equiv \\ \langle \forall G, H :: X \;.\; G{\circ}F{\circ}H \rangle \;.\end{array}$$

In a similar way, a property X can be weakened into an existential property $\mathbb{SE}.X$ such that $\mathbb{SE}.X$ is the strongest existential property weaker than X [14]. $\mathbb{SE}.X$ holds in any system that contains a subsystem that satisfies X. It is, in some sense, the "existential part" of X, the part of the property that is preserved when an X-system is composed. $\mathbb{SE}$ can also be expressed in terms of our general transformers:

$$[\mathbb{SE}.X \;\equiv\; \mathit{true} \circ X \circ \mathit{true}] \;.$$

Proposition 4 *For any property X, there exists a strongest existential property weaker than X and it is $\mathit{true} \circ X \circ \mathit{true}$.*

Things are different with the universal form of composition because it can be shown that, for some properties, there does not exist a weakest universal property that strengthen them [15]. Therefore, we cannot define a transformer $\mathbb{WU}$ in the same way as we defined $\mathbb{WE}$. Still, any property X has a strongest universal property weaker than itself, denoted by $\mathbb{SU}.X$. $\mathbb{SU}.X$ characterizes those systems built from X-components only.

Finally, each one of the transformers $\mathbb{WE}$, $\mathbb{SE}$ and $\mathbb{SU}$ has a unique conjugate. As $\mathbb{WE}$, $\mathbb{SE}$ and $\mathbb{SU}$ allow us to deduce system properties from subsystems properties, their conjugates are used to deduce subsystems properties from system properties. For instance, if a system satisfies a property X, all its subsystems must satisfy $\mathbb{WE}^*.X$. Components that do not satisfy $\mathbb{WE}^*.X$ need not be considered at all when the goal is to build an X-system, since in no case can they lead to such a system. $\mathbb{WE}^*$ is an indication of what components can possibly be used to obtain a given system. Similarly, to ensure that at least one subsystem of a system satisfies X, the necessary and sufficient condition is to prove that the system satisfies $\mathbb{SU}^*.X$.

9 Special Special Case: Concurrent Composition of Reactive Systems

We can further instantiate the previous framework by choosing what the single law of composition is, what components are, and how they are specified. Because of our interest in formal specification and verification of reactive systems, we have started to investigate the application of our work on composition to reactive systems composed concurrently and specified with temporal logics. Below is a short presentation of what can be done using a UNITY-like logic. The use of other logics, such as CTL, has been investigated as well [6].

In UNITY[3], processes are represented as a form of fair transition systems. Basically, a system consists of a state predicate (the set of possible initial states), a set of atomic transitions, and fairness assumptions. Properties of such systems are described using a fragment of linear temporal logic.

The fundamental safety operator of the logic is *next*: p *next* q is a property that asserts that if the state predicate p is true and an atomic transition is executed, then q is true in the resulting state. The fundamental liveness operator is *transient*: *transient* p means that there is at least one fair transition in the system that falsifies predicate p. These operators usually come in two flavors: a weak form where only reachable states are considered, and a strong form where reachability is not taken into account. In their strong form, UNITY properties can be composed: *transient* p is existential and p *next* q is universal. In their weak form[4] however, they correspond to closed systems properties and that compositionality is lost.

[3] What we refer to as "UNITY" in this paper does not coincide exactly with the framework as it is defined in [8] or in [25, 24]. Some liberties are taken with UNITY syntax and semantics, but the general philosophy is retained.

[4] The weak form of *transient* is not used directly. A property *leads-to* is used instead, that can be defined in terms of *transient*. *Leads-to* properties are notoriously difficult to compose.

The strong form of the logic can be used as a tool in proofs, but is not suitable for component specifications (it is much too strong). However, transformers such as $\mathbb{WE}$ can be applied to the weak form of the logic to obtain useful, compositional specifications. In particular, specifications of the form $\mathbb{WE}.(X \Rightarrow Y)$ have proved to be powerful assumption-commitment specifications [13]. The fact that those properties are existential even when X is a progress property (such as *leads-to*) allows us to write component specifications that embed substantial parts of a correctness proof inside a component (see discussion in [11]). This is an important feature because it makes it possible to reuse proof efforts when components are reused. Such a potential reuse of proofs is what makes composition worthwhile in spite of the natural overhead it generates [22].

The absence of a transformer $\mathbb{WU}$ and the fact that some properties do not have a weakest universal strengthening give raise to an interesting question: How can the non universal property p *weak-next* q be strengthened into a universal property? We would like to find a universal property X that is stronger than p *weak-next* q but weaker than p *strong-next* q and $\mathbb{WE}.(p$ *weak-next* $q)$ (which are universal). Ideally, we would like X to be the weakest universal property stronger than p *weak-next* q, provided it exists. The problem looks deceptively simple but, so far, we have been unable to answer it. We were able to define a universal property that is stronger than p *weak-next* q and weaker than p *strong-next* q and $\mathbb{WE}.(p$ *weak-next* $q)$ [10], but we do not know if that property is the weakest solution to our problem or not. We do not even know if such a weakest element exists at all. Such a question deserves to be investigated because a universal *next*-like operator can be useful to specify invariance properties in a compositional way. It is also a first step into finding a suitable definition for a transformer $\mathbb{WU}$. (A more precise statement of these questions can be found at `http://www.cs.unh.edu/~charpov/Composition/challenge.html`.)

10 Summary

We started this study because of our conviction of the importance of composition is system design. We believe that systems, and especially software systems, should increasingly be constructed from generic "off the shelf" components. This means that *reuse* of systems and components is going to be a central issue.

In order to obtain reusable components, we must be able to describe requirements in a rather abstract way, focusing on the functionality we expect from the component and without refering to implementation details. However, there is a tradeoff between hiding details and being composable: more abstract specifications lead to more difficult composition, and specifications that are too abstract may even totally prevent composition. Our "?" transformers are a way of describing what is the minimum one has to say about a component to ensure the applicability of specific forms of composition (such as existential or universal or with an explicit system pattern).

This need for abstract specifications made us depart from the process calculus approach and focus our interest on logical specifications. Especially, we are interested in applying our ideas to concurrent systems specified with temporal logics. A great amount of work has been done regarding composition of reactive systems specified with temporal logics. Among the logics that were considered, we can cite the linear temporal logic [21, 23], TLA [2], ATL [5] or Unity [16–19, 26]. All these works, and others [1, 3, 4], rely on the same hypothesis that systems are described in terms of *open computations*, i.e., infinite traces that are shared with the environment.

One originality of our work is to drop this view on composition and to attempt to reason about composition in the abstract: we do not assume that systems have computations or even states. This way, we hope to better understand what the fundamental issues of composition-based designs are. Nevertheless, we do not forget our interest in temporal logics and concurrent systems. We have used existential and universal properties to derive examples of specifications and correctness proofs of distributed and concurrent systems [12, 13, 7, 6]. Several of these examples make use of an operator called *guarantees* [9] which has been proved to be a special case of the transformer $\mathbb{WE}$ [15].

Most of our previous work is based on a single law of composition that is assumed to be associative and to have an identity element. In some cases, we have additional hypotheses regarding the existence of inverses of components. In our work on temporal logic and concurrent composition, we use a law of composition that is symmetric and idempotent in addition to being associative. In this paper, we attempt an exploration of the common basis to all previously defined transformers. We show that our use of predicate transformers can be systematic and that, for each composition related question, there is a predicate or a predicate transformer that relates to it precisely.

Our long term goal is the construction of a calculus for composition. Such a calculus requires us to have access to a large number of generic theorems and proof rules. We have started to state some of these rules. In the context of a single law of composition over a monoid, we were able in some cases to *calculate* $\mathbb{WE}.X$ for some properties X, in other words, to answer a question about composition with a systematic calculation [15]. This was only achieved on toy examples and our current work is to find other generic rules as well as rules specific to temporal logic and concurrent composition.

Acknowledgments

This work is based on many discussions with K. Mani Chandy. The author is grateful to him for these helpful exchanges of ideas and the valuable comments he made on an earlier draft of this paper.

References

1. Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.
2. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
3. Martín Abadi and Stephan Merz. An abstract account of composition. In Jivrí Wiedermann and Petr Hajek, editors, *Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science*, pages 499–508. Springer-Verlag, September 1995.
4. Martín Abadi and Gordon Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.
5. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.
6. Hector Andrade and Beverly A. Sanders. An approach to compositional model checking. In *International Parallel and Distributed Processing Symposium. Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'02)*. IEEE, 2002.
7. K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, January 2002.
8. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
9. K. Mani Chandy and Beverly Sanders. Reasoning about program composition. http://www.cise.ufl.edu/~sanders/pubs/composition.ps.
10. Michel Charpentier. Making Unity properties compositional: the transformer $\mathcal{E}$, the predicate SIC and the property type $next_U$. Unpublished research report, September 1999.
11. Michel Charpentier. Reasoning about composition: A predicate transformer approach. In *Specification and Verification of Component-Based Systems (SAVCBS'2001)*, pages 42–49. Workshop at OOPSLA'2001, October 2001.
12. Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In J. Rolim, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1215–1227. Springer-Verlag, April 1999.
13. Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM'99), (Vol. I)*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer-Verlag, September 1999.
14. Michel Charpentier and K. Mani Chandy. Reasoning about composition using property transformers and their conjugates. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics (IFIP-TCS'2000)*, volume 1872 of *Lecture Notes in Computer Science*, pages 580–595. Springer-Verlag, August 2000.
15. Michel Charpentier and K. Mani Chandy. Theorems about composition. In R. Backhouse and J. Nuno Oliveira, editors, *International Conference on Mathematics of Program Construction (MPC'2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 167–186. Springer-Verlag, July 2000.

16. Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to* UNITY. Doctoral thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, June 1994.
17. Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50:31–35, 1994.
18. Pierre Collette and Edgar Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. In *International Conference on Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 353–367. Springer-Verlag, 1995.
19. Pierre Collette and Edgar Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183:253–279, 1997.
20. Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics.* Texts and monographs in computer science. Springer-Verlag, 1990.
21. J.L. Fiadeiro and T. Maibaum. Verifying for reuse: foundations of object-oriented system verification. In I. Makie C. Hankin and R. Nagarajan, editors, *Theory and Formal Methods*, pages 235–257. World Scientific Publishing Company, 1995.
22. Leslie Lamport. Composition: A way to make proofs harder. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS'97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423. Springer-Verlag, September 1997.
23. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.
24. Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.
25. Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.
26. Rob T. Udink. *Program Refinement in* UNITY*-like Environments.* PhD thesis, Utrecht University, September 1995.