Reasoning about Composition: A Predicate Transformer Approach

[Position Paper]

Michel Charpentier
Department of Computer Science
University of New Hampshire
charpov@cs.unh.edu

ABSTRACT

As interest in components and composition-related methods is growing rapidly, it is not always clear what the goals (and the corresponding difficulties) actually are. If composition is to become central in the future of software engineering, we need to better identify the fundamental issues that are related to it, before we attempt to solve them as they occur in object-oriented systems or in concurrent and reactive systems. In this paper, we present our formulation of some of the composition problems in a context of formal methods and program specification and verification. This formalization is based on predicate calculus and predicate transformers and aims at remaining as general as possible. This way, we hope to better understand some of the fundamental issues of composition and component-based reasoning.

1. INTRODUCTION

Composition is receiving a lot of attention these days: Components are everywhere and everything is (or ought to be) "*compositional*". What is meant by that, though, is far from being clear, and there is a wide range of opinions on what is still to be done. Some might argue that the composition problem is now solved at a fundamental level and that actual techniques and tools just need to be put in place. At the other end of the spectrum, some might believe that composition, as we understand it today, cannot be achieved in software engineering and that other approaches must be sought. And, between these two extremes, are research projects, mostly independent from one another, that focus on specific instances of this composition problem, be it a type system in an object-oriented context or a temporal logic for reactive systems.

A possible reason for this apparent contradiction and confusion is that composition is a broad concept and that the composition problem might not be unique. There are many issues related to composition, some are easier to tackle than others, and many must be dealt with before the problem can be considered solved (or unsolvable). In this paper, we advocate the idea that an important step today is to identify those composition problems and to understand how they relate to each other.

We restrict our attention to composition in a formal methods context. Other contexts, such as for instance programming languages, lead to other composition problems and all have to be solved in order to make composition viable as a whole. Furthermore, we choose a static point of view: we reason about properties of systems and components whereas a dynamic point of view would focus on the *process* of building systems from components.

These choices, however, leave us in a broad background where fundamental questions related to many forms of composition can be explored: What are components? How are they composed? How are they described and specified? What do we expect from such specifications? What is the relationship between systems and components specifications? What does it mean to be "*compositional*"? How can we obtain compositional specifications? Can composition lead to simpler correctness proofs? How does composition relate to reuse? How does it relate to abstraction?

Our current effort focuses on addressing these questions without specializing the chosen context any further. This way, we hope to better understand what problems are common to different forms of composition and what problems are specific to families of components or laws of composition. As a guideline for this general exploration, we also consider the special case of concurrent composition of processes specified in temporal logic. This familiar but complex background, in which the composition problem is far from being solved, is both a source of inspiration and a test-bench for our abstract study of composition. Our approach to studying composition as well as some of our results are informally introduced in the remaining of this paper. Technical details can be found in cited references.

Specification and Verification of Component-Based Systems (SAVCBS'2001). Workshop at OOPSLA'2001. Pages 42–49.

2. SPECIFICATIONS AND PROOFS IN COMPOSITIONAL DESIGNS

2.1 Compositional Design versus Compositional Verification

Composition has often been advocated as a necessary step in the proof of large systems. While this is certainly true, we do not want to restrict composition to that role.

For instance, it is possible to build a system from components, generate correctness proof obligations from the *complete* system, and then apply composition at the proof level (split the global proof obligation into several independent proofs). This approach is suggested, for instance, in [24]. Compositional model-checking also follows this philosophy to some degree.

While the previous technique is relatively simple and allows verification techniques to handle large systems, we have in mind a more ambitious role for composition, namely the "open system" approach. In this approach, we want to verify the correctness of components in isolation, *before* they become part of any system. In the previous case, the complete knowledge of the system can be used to verify one component. For open systems, this is not true anymore. All that is known are specific assumptions on possible environments, which are part of a component specification. This tends to make proofs harder since these assumptions describe a set of possible environments instead of a completely specified context, and they have to be abstract and generic enough to allow a large number of environments to use the component.

However, the open system approach also has benefits that make its study worthwhile. Firstly, since components are already proved correct with respect to their specifications, the correctness proof of a complete system can rely on these specifications instead of the components' implementations. This allows designers not to take into account the many details of the internal structure of each component. Compositionality of designs breaks down when reasoning about a system requires managing too many details from each part of that system.

Secondly, and this is probably the main benefit, the open system approach allows designers to embed parts of a correctness proof into components, making these parts available each time a component is used to build a system. Indeed, when a component is proved correct with respect to its specification, relevant facts about this component are extracted from the details of its implementation and become part of the component specification. When this component is composed with a larger system, these facts can be used in the system correctness proof without the need for proving them again. Each time a component is reused, a (possibly difficult) proof is reused too, as well as any other correctness argument available such as tests or behavior in other systems.

2.2 Abstract Specifications

In order to be able to achieve such reuse, we need specifications to remain abstract enough to describe what is required from a component, all that is required and only what is required. When designing a system and looking for a suitable component, the specification used by the designer cannot include too many details about this component, because any component with the right functionalities should be usable, whatever its implementation details are. Such a specification must also be able to express that some aspects are irrelevant in order to avoid an overspecification of requirements. If requirements are overspecified, then designers might end up not finding any suitable component while actually some existing component would fit their needs perfectly.

A second reason why we want specifications to be abstract is to keep composition worthwhile and cost effective in spite of the natural overhead it generates. A key idea of component technology is that the same component can be used in many systems, and thus the effort that goes into specifying, proving and implementing components can be exploited many times. As explained before, each time a component is reused, a proof, the correctness proof of that component, is reused too. If a component specification contains abstract, relevant, hard-to-prove facts about the component, a possibly difficult and large proof is reused. However, if a component specification is too close to its implementation and not abstract enough, very little proof can be reused. Therefore, greater productivity is achieved by using components that embody substantial effort by containing proofs of *abstract* specifications.

This situation is illustrated in figure 1. Proofs labeled with 'T' are those component-correctness proofs that are left unchanged through composition and that can be reused in the design of several systems. Proofs labeled with 'C' are proofs of composition, i.e., proofs of system properties from component properties. The level of abstraction of component specifications clearly influences the amount of effort that has to be put in T-proofs and in C-proofs. A good framework for composition should allow us to put most of the effort in T-proofs and keep C-proofs as simple as possible. Even if the sum of C and T-proofs is larger and more complex than a direct (noncompositional) proof for the same system, composition is still worthwhile because existing T-proofs can be reused.

Part of the problem is that specifications that are too abstract do not contain enough information to be composed. Therefore, the right balance between abstraction and ability to be composed must be found.

3. SPECIFICITY OF OUR RESEARCH

3.1 Shortcomings of Current Approaches

When deterministic components are composed sequentially, the problem reduces to composition of functions and remains tractable. Developers use libraries of procedures every day and rely on their specifications without having to consider implementation details.

However, effective compositional design often involves nondeterministic components and concurrent composition. For instance, the different parts of a reactive system cannot be specified in terms of precondition and postcondition because of their possibly infinite behavior, which leads to tremendous difficulties in terms of composition.

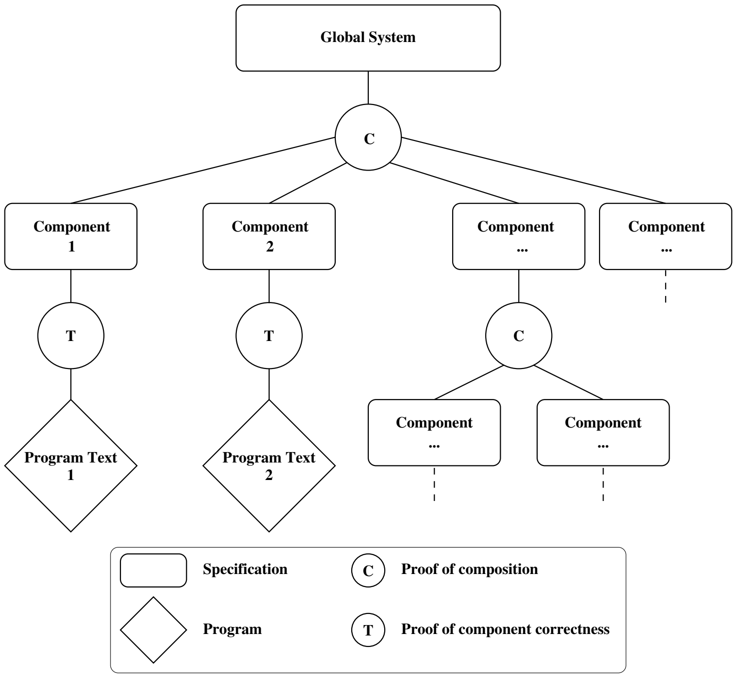

Figure 1: A compositional design

Composition of such systems, which interact at the level of their behavior, not at the level of their initial/final state, has been extensively studied. Very schematically, two distinct families emerge.

On the one hand, process algebras, such as CSP, CCS or π-calculus to name a few, integrate composition as a central part of their design. Systems are compositions of processes and processes compose quite naturally. The resulting formally well-defined notation, however, often looks like a programming language more than a specification language. In this context, it is quite difficult to express abstract properties on the expected behavior of these components and systems. As a consequence, it is difficult to obtain reusable generic specifications, as well as specifications easily related to informal requirements.

Temporal logics, on the other hand, such as LTL, CTL, TLA, or UNITY, are well-suited to express nonoperational, abstract specifications. They provide us with specification languages that are closer to informal descriptions, which makes specifications more easily readable and checkable with respect to informal requirements. However, the starting point of these notations is the specification of a system, globally. Composition is viewed as an additional issue, which requires a specific treatment. Work has been done to manage composition issues with specific logics [25, 19, 18, 2, 20, 23, 34, 21], but little work has been done to study composition in itself, independently of the underlying logic [1, 4, 3].

3.2 Composition in the Abstract

The specificity of our approach is to study composition independently from what components and the laws of composition actually are. We are not focusing on a specific domain, nor do we want to design specification languages tailored to certain forms of composition. Our view is one of a component-based software industry, where composition is involved in almost every design. We want to deal with this composition, whether it works well or not, whether it is easy or not. This departs from the works on temporal logic cited above, where usually the whole language is made "composable" by restricting up front the type of interaction under consideration. For instance, composition works fine in TLA (it reduces to conjunction) [2], but component assumptions are made at the transition level and cannot be at the computation level, as in the case of liveness assumptions (see example in section 4.4).

The context of our work is therefore independent of the nature of systems. It is not a context of "variables", "states", "computations", "interleaving", "safety" or "liveness", but rather one of "systems", "components", "specifications" and "composition laws". No specific logic or process model is used and few hypotheses are made on composition laws. This way, it is hoped that we can understand aspects that are common to many forms of composition and many types of systems. Later, that knowledge can be applied to the concurrent composition of reactive systems, for instance.

This approach inherits from both the process algebra and the temporal logic families mentioned above. On the one hand, we consider that systems are specified logically (without choosing a specific logic), which provides us with a rich specification language and allows us potentially to apply results to temporal logics for reactive systems. On the other hand, we define an algebra of composition with the goal of obtaining a calculus that would allow us to calculate (instead of guess and then prove) properties of systems and components. In this respect, our approach relates to process calculus based approaches.

4. CURRENT WORK

4.1 Existential and Universal Specifications

The starting point of our exploration is the definition of a simple model of components, systems and specifications. Because of our concern with generality, we use a monoid-like structure of components and specifications are boolean functions (predicates) on components and systems. In other words, we assume that components are composed with a single law of composition for which we assume associativity but no other property such as symmetry or idempotency. As a consequence, the model can be instantiated with transformational programs (specified in terms of preconditions and postconditions and composed sequentially) or with reactive processes (specified with temporal logics and composed concurrently), among other things.

In this context, we first focused on two particular families of specifications called *existential* and *universal* [7, 14]. We say that a specification is existential exactly when, for all systems, the specification holds in a system if it holds in at least one component of that system. Similarly, a specification is universal if it holds in a system when it holds in all components of that system. Existential and universal are characteristics of specifications, independent of a particular set of components. Some specifications are existential, some are universal and, of course, some are neither. However, when existential and universal specifications are used, they naturally lead to simple proofs of composition (C-proofs), properties being inherited by a system from its components.

4.2 A "*guarantees*" Operator for Assumption-Commitment Specifications

If we only allow existential and universal specifications to appear in component descriptions, this is a restriction on how components can be described. This is the price to pay for simple proofs of composition. However, we have found these two classes to be surprisingly rich. For instance, the work on temporal logics in [19, 18, 2, 20, 34, 21] relies almost exclusively on existential-like composition.

One reason why existential specifications appear to be so convenient is the existence of the *guarantees* operator defined in [7]. The *guarantees* operator can be used to express existential assumption-commitment specifications. Its main originality is that it is not defined in terms of component *environments*, as assumption-commitment specifications usually are (components are making assumptions on their possible environments). In the case of *guarantees*, the commitment part of the specification *as well as the assumption part* apply to a complete system (environment + component): X *guarantees* Y holds in a component F if and only if Y holds in $G{\circ}F{\circ}H$ (where $\circ$ denotes the law of composition under consideration) when X holds in $G{\circ}F{\circ}H$, for all systems G and H that can be composed with F. The fundamental property of *guarantees* is that X *guarantees* Y is existential regardless of what the specifications X and Y are. Therefore, proofs of composition are simplified when components are specified in terms of *guarantees*.

4.3 Predicate Transformers for Composition

By studying the *guarantees* operator carefully, we made the observation that it is merely the application to logical implication of a more general operator which we called $\mathbb{WE}$ [6]. This allows a separation of concerns: $\mathbb{WE}$ actually represents composition while logical implication represents the assumption-commitment mechanism. $\mathbb{WE}$ is a predicate transformer, in other words, a function from specifications to specifications. Formally, for a specification X, $\mathbb{WE}.X$ is defined as the weakest existential specification stronger than X (which exists regardless of X).

It can be proved that $\mathbb{WE}.X$ characterizes those components F such that specification X holds in any system that contains F as a component [14]. As a consequence, the specification X *guarantees* Y is actually equivalent to $\mathbb{WE}.(X \Rightarrow Y)$. In other words, *guarantees* is the weakest (the most abstract) strengthening of logical implication that makes it composable (for the existential form of composition). This, in some sense, is a theoretical argument to claim that *guarantees* can provide us with abstract, reusable specifications.

$\mathbb{WE}$ is the first of a series of predicate transformers that we have started to study. Indeed, we can define $\mathbb{SE}.X$ as the strongest existential property weaker than X. The corresponding theorem states that $\mathbb{SE}.X$ characterizes those systems that contain at least one component that satisfies X. In other words, when a component that satisfies X is used in a system, this system satisfies $\mathbb{SE}.X$. In the best case (when X is an existential specification), the system satisfies X ($\mathbb{SE}.X$ is equivalent to X); in the worst case (where all of X is lost through composition), $\mathbb{SE}.X$ reduces to *true*. In some sense, $\mathbb{SE}.X$ represents the part of specification X that composes (existentially). Equivalently, $\mathbb{SE}.X$ characterizes those systems that are (or can be) built using a component that satisfies specification X [15].

Things are different in the case of universal composition. A transformer $\mathbb{SU}$ can be defined (as the strongest universal specification weaker than a given specification), but we are still looking for a suitable $\mathbb{WU}$. Such a transformer would be useful to characterize what has to be proved on a component instead of a (nonuniversal) specification X in order to inherit the simplicity of universal composition. However, it cannot be defined as the weakest universal specification stronger than a given specification because such a weakest element does not always exist, depending on the nonuniversal specification that is considered. We have started to study several possible candidates for a $\mathbb{WU}$ operator but we do not have a strong argument in favor of one of them yet. As a guideline for that search of $\mathbb{WU}$, we have also studied the question of strengthening nonuniversal properties in a more restricted context, namely a linear temporal logic (see section 4.4).

Furthermore, by describing composition in terms of predicate transformers, for which a large amount of literature exists [22], we are able to reuse classic techniques such as *conjugates*. Every predicate transformer $\mathcal{T}$ has a unique conjugate $\mathcal{T}^*$ such that $\mathcal{T}^*.X = \neg\mathcal{T}.(\neg X)$. The transformers we have defined for existential and universal composition also have conjugates, namely $\mathbb{WE}^*$, $\mathbb{SE}^*$ and $\mathbb{SU}^*$. It should be noted that, while $\mathbb{WE}$, $\mathbb{SE}$ and $\mathbb{SU}$ describe composition from components to systems (what has to be proved on components, what can be deduced on systems), $\mathbb{WE}^*$, $\mathbb{SE}^*$ and $\mathbb{SU}^*$ describe composition from systems to components (what should be proved on systems, what can be deduced on components). For instance, $\mathbb{WE}^*.X$ is true of any component that is used to build a system that satisfies specification X. This form of reasoning, from systems to components, is sometimes neglected. We believe it to be extremely important because it is the kind of reasoning that is involved when system designers are looking for components. A designer who is building a system to satisfy specification X knows that only components that satisfy $\mathbb{WE}^*.X$ can be used and that other components need not be considered. We find conjugates to be a powerful and elegant way to switch from bottom-up to top-down views on composition [13]. In particular, many properties of predicate transformers, such as junctivity and monotonicity, are inherited from transformers to conjugates. This allows us to avoid duplicating proofs.

4.4 Application to UNITY logic

In parallel with our work on predicate transformers and composition, we have started to apply our ideas to specifications and proofs of concurrent and distributed systems. Theoretical investigation is one way to claim the usefulness of operators (for instance, by proving that they are the weakest solution to some set of equations). Practical attempts at writing specifications and proofs based on these operators are another.

Two of these examples were fully developed and published. One focuses on shared memory systems [11], while the other deals with distributed systems [12, 5].

In the first example, universal specifications are used instead of *guarantees*, which does not seem to fit this example well enough. In this case, the correctness argument relies on the fact that some dependency graph among processes remains acyclic. Since each process only modifies the dependency graph locally (by interacting with its neighbors), no single process can guarantee that the graph remains acyclic, using an existential property. However, there can be a property that states that no process will ever create a cycle in the graph. Such a property can be formulated in a universal way so that, when it is satisfied by all processes, the global system also satisfies it and cycles cannot be introduced in the graph.

This raises a number of interesting questions. In this example, it appears that universal specifications are required to describe the behavior of shared variables (variables that are written by several processes). However, there are other examples with shared variables that can successfully be specified in terms of *guarantees*. There are also systems without shared variables (distributed systems) but where a shared virtual data structure (such as a graph among processes) is used in the correctness proof. Should such a system be specified in terms of *guarantees* (it usually can, from the absence of shared variables) or in terms of universal specifications of the shared virtual data structure? And if *guarantees* is used, should the correctness proof rely directly on it or can we obtain a simpler proof by using an intermediate (universal) specification that is deduced from the original (existential) specification? These are the kind of fundamental questions we plan to explore through the development of other examples.

Using universal specifications gives rise to other interesting issues. For instance, the UNITY logic (which was used in our examples) exists in two forms: a weak form and a strong form [31, 28, 27]. The UNITY operator *invariant* leads to universal specifications in its strong form but not in its weak form. For the sake of simplicity, we used the strong form of UNITY logic in our example. However, this is not realistic from a practical point of view (the strong form of the logic is much too strong for a specification) and we have to find ways of strengthening the weak form to make it universal. We have defined such a strengthening based on $\mathbb{WE}$ [9] (the resulting universal form of the weak invariant resembles a similar operator from [34]), but we cannot tell if this is an optimal solution. In other words, we do not know if the resulting operator is the weakest universal specification stronger than the weak invariant (we do not even know if such a weakest solution exists). Besides its practical interest, this question also relates to the problem of finding a suitable transformer $\mathbb{WU}$, as explained earlier in section 4.3.

Our second example involves distributed systems. It makes use of *guarantees*, mixed with techniques for abstract communication description that were previously developed [16, 26, 8, 33, 17]. This abstract description of communication is made possible by the ability of *guarantees* to involve liveness specifications in its assumption part. Basically, a network component guarantees that the sequence of received messages is always a prefix of the sequence of sent messages (safety) and that any message that is sent is eventually received (liveness).

There are other places in this example where our use of liveness specifications combined with *guarantees* leads to simpler proofs of composition by embedding larger proofs in components verification (see the discussion in 2.2). For instance, this example involves a resource allocator component that satisfies a property of the form: *clients return resources in finite time (and other conditions)* guarantees *any request for resources is eventually satisfied.* The proof of composition remains simple because the corresponding client component property that states that clients actually return resources in finite time can be plugged (through network specifications) into the left-hand side of this *guarantees* property to deduce that all requests are eventually granted.

If liveness properties cannot be used in the assumption part of a composition operator $\mapsto$ (as in [1, 2, 18, 19, 20, 21]), the resource allocator specification has to be of the form: *enough resources are available to satisfy the first pending request* $\mapsto$ *the first pending request is eventually granted.* In this case, the fact that clients return resources in finite time cannot be used directly as before. Instead, a first proof of composition is required to show that enough resources will eventually be available to satisfy the first pending request and then a second proof to show that other requests are eventually satisfied. When *guarantees* is used, these two proofs (by induction) are inside the correctness proof of the allocator component and can be reused when the allocator component is reused. In the other case, they are in the proof of composition and have to be redone every time a new system is built from these components.

5. OUTLINE OF FUTURE RESEARCH

The work described above represents a first step towards our exploration of composition issues in system design. Starting with *guarantees* as a middle point, the research is now developing both upstream (towards predicate transformers and other fundamental composition-related operators) and downstream (towards practical application to concurrent systems).

One of our goals is the definition of a formal calculus in which specifications can be transformed to fit specific composition constraints. In other words, starting from requirements that are not compositional, we want to calculate a suitable compositional specification. In the case of existential composition, for example, it is not enough to know that $\mathbb{WE}.X$ is what needs to be proved on a component to ensure that systems which use that component will satisfy specification X. We need to know *how* to prove $\mathbb{WE}.X$ given a component description.

This can be achieved at different levels. At the most abstract level, we can exhibit theorems about $\mathbb{WE}$ that allow us to reduce the calculation of $\mathbb{WE}.X$ using known $\mathbb{WE}.Y$, where Y is a part of X (for instance, using existential Y specifications). When this is possible, we can *calculate* $\mathbb{WE}.X$ inside the logic in which X is expressed, which gives us the corresponding component specification. We were able to achieve such calculations on toy examples [14], but we need more theorems and rules related to $\mathbb{WE}$ and our other transformers to be able to conduct such calculations on examples from more interesting domains. One difficulty when seeking such properties of the transformers is to free ourselves from implicit assumptions regarding the law of composition. For instance, we sometimes use concurrent composition of processes as a guideline to find general rules about the transformers, but we must be careful not to use an hypothesis such as symmetry or idempotency which we decided not to include systematically in our model.

Another way to deal with the transformers is to first instantiate our framework with a specification language and then to derive rules about $\mathbb{WE}.X$, when X is expressed in the chosen logical language (instead of using general theorems about $\mathbb{WE}$). We have started this process with Unity logic in order to build the necessary correctness proofs in our examples with concurrent and distributed systems [9]. Furthermore, we also need to apply our approach to other frameworks for the specification and verification of concurrent systems. This effort has already started, for instance with CTL [32], but we want to consider other frameworks, such as TLA or I/O-automata.

Recently, we have started to generalize our approach to systems in which several laws of composition are used at the same time. An example of such a system is a software system in which components are composed sequentially *and* in parallel. According to preliminary results, it seems that our approach can still be applied. In other words, we are still able to define weakest and strongest transformers that represent specific views on composition (independently, this time, from existential and universal specifications). Furthermore, the resulting predicate transformers bear strong similarities with Dijkstra's *wlp* and *sp* transformers for program semantics, from which we can draw new inspirations [10]. This new set of transformers has now to be explored carefully. Especially, relationships between transformer properties and assumptions on the different laws of composition have to be found.

6. SUMMARY

The lack of composition-based methods is a major factor in the limited use of formal methods in actual designs. We believe our project adopts a novel view on an old and important problem. Most work on composition has focused on a specific form of composition (sequential, parallel with shared variables, parallel with message passing, etc.) and a specific type of component (namely, programs, either with states or with so-called "open system computations"). By choosing a much more general view, we hope to understand fundamental aspects of composition that are independent from the types of components and the way they interact.

Our ultimate goal is to build a calculus for composition. It would be a formal framework that can be instantiated with many form of compositions and many types of systems and components. We hope this framework will include generic rules and theorems about composition and logical specifications. The search for such fundamental rules, common to any kind of composition, is an exciting problem. Then, each instantiation enriches the framework with additional rules that are specific to this instantiation, making it more complete and more practically usable.

Besides this theoretical part of the project, we are experimenting with several notations for the specification and verification of concurrent systems to see how they can be extended through our approach into compositional notations. We hope, by modifying and extending existing notations, to develop an interesting framework to reason about concurrent composition of reactive systems. Another aspect of the problem is related to mechanization. We are investigating the question of the mechanization of *guarantees* through a collaboration with Larry Paulson from the University of Cambridge. Larry is currently working on a mechanization of Unity [29] extended with *guarantees* [30] in the higher-order generic theorem prover *Isabelle*. His work is guided by his attempts at mechanizing hand proofs from our example involving distributed systems.

We are convinced that the future of software engineering is tied to composition. Component-based designs and reuse of generic components will be at the core of future software systems. Composition involves a number of practical issues, but also raises fundamental questions regarding component specifications and compositional reasoning. We need to improve our understanding of composition if we want to be able to devise the tools and principles that will allow us to use components reliably and efficiently in software engineering. Our project has started an exploration of some of the fundamental questions inherent in compositional design.

7. 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] K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, April 1999. Accepted for publication.

[6] K. Mani Chandy and Michel Charpentier. Predicate transformers for composition. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science: proccedings of the 1999 Oxford-Microsoft symposium in honour of Sir Tony Hoare*, Cornerstones of Computing, pages 81–90. Palgrave, 2000.

[7] K. Mani Chandy and Beverly Sanders. Reasoning about program composition. http://www.cise.ufl.edu/~sanders/pubs/composition.ps.

[8] Michel Charpentier. *Assistance à la Répartition de Systèmes Réactifs*. PhD thesis, Institut National Polytechnique de Toulouse, France, November 1997.

[9] Michel Charpentier. Making UNITY properties compositional. Unpublished report, California Institute of Technology, 1999.

[10] Michel Charpentier. A theory of composition motivated by wp. Submitted for publication, August 2001.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Michel Charpentier and K. Mani Chandy. Specification transformers: A predicate transformer approach to composition. Submitted for publication, July 2001.

[16] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe 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.

[17] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. The observation: an abstract communication mechanism. *Parallel Processing Letters*, 9(3):437–450, September 1999.

[18] 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.

[19] Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50:31–35, 1994.

[20] 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.

[21] 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.

[22] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Texts and monographs in computer science. Springer-Verlag, 1990.

[23] 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.

[24] 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.

[25] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

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

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

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

[29] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1), July 2000.

[30] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Computational Logic*, 2001. To appear.

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

[32] Beverly A. Sanders and Hector Andrade. Model checking for open systems. Submitted for publication, 2000.

[33] Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997.

[34] Rob T. Udink. *Program Refinement in UNITY-like Environments*. PhD thesis, Utrecht University, September 1995.