Technical Report

UCAM-CL-TR-573
ISSN 1476-2986

Number 573

UNIVERSITY OF
CAMBRIDGE
Computer Laboratory

Mechanizing compositional reasoning for concurrent systems: some lessons

Lawrence C. Paulson

August 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Mechanizing Compositional Reasoning for Concurrent Systems: Some Lessons

Lawrence C. Paulson
Computer Laboratory, University of Cambridge

12 August 2003

Abstract

The paper reports on experiences of mechanizing various proposals for compositional reasoning in concurrent systems. The work uses the UNITY formalism and the Isabelle proof tool. The proposals investigated include existential/universal properties, **guarantees** properties and progress sets. The paper mentions some alternative proposals that are also worth of investigation. The conclusions are that many of these methods work and are suitable candidates for further development.

Contents

1 Background

Compositional reasoning means proving properties of a system from the properties of its components without reference to the components' implementations. Much research has concentrated on how to verify simple program units. Model checkers can cope with complex systems, which are formalized as monolithic units. Nonetheless, without compositional reasoning, we shall quickly exceed any verification tool's capacity. In the future, program components will increasingly be reused and combined to form complex systems—which we hope can be verified.

The present work is in the context of concurrent systems and uses the UNITY formalism [3]. Concurrent systems are becoming ubiquitous: the cash machine network is a giant concurrent system, and common desktop applications such as Web browsers and file explorers are multi-threaded. Concurrent systems are difficult to get right because of their inherent nondeterminism: even known faults can be difficult to reproduce. UNITY is an extremely simple formalism for concurrency. It supports reasoning about abstract implementations, but it also allows reasoning about programs on the basis of specifications alone. It has no claim to be sophisticated enough to support the verification of real-world programs; instead, its simplicity makes it easy to mechanize using mechanical proof tools. Its meta-theory is straightforward and easy to verify mechanically. UNITY is an ideal basis for experimenting with new techniques such as those for compositional reasoning. Lessons learned from UNITY can then be transferred to more sophisticated temporal formalisms.

This paper summarises several years of research into mechanizing compositional reasoning in UNITY. The common thread is the transfer of pencil and paper methods to computer based proof tools: specifically, Isabelle [24]. Pencil and paper methods rely on informal mathematics, but computer-based tools must inevitably use formal logic. Some of the assumptions implicit to the pencil and paper world are not easily accommodated in formal logic; moreover, because they are implicit, their importance can be underestimated and their very existence can be overlooked.

This paper reports experiments involving several techniques for compositional reasoning:

- existential and universal properties
- the **guarantees** relation, which is part of the previous technique [4, 7]
- progress sets [16]

Some of this work is in collaboration with my colleague Sidi Ehmety. We have had mixed results with these methods and attempt to summarise our findings below.

In the interests of brevity, this paper keeps details to a minimum. Please refer to cited papers for complete descriptions of the various experiments.

Before continuing, I should address Leslie Lamport's points in his paper "Composition: a way to make proofs harder" [14]. His paper actually devotes little of its 21 pages to composition. Much of the paper is an exposition of Lamport's temporal logic of actions (TLA) [13]. This formalism is a sophisticated rival to UNITY that has had some acceptance in industry. Lamport notes that to decompose a monolithic system into components is unnecessary when the original system can be verified using a model checker. In other words, he is talking about *decomposition*: the act of adding structure to an existing program. He does not consider composition (as is understood in the present paper) until his penultimate section, where he refers to reusable software and notes that engineers rarely verify the systems they build at present (1997 in his paper). They still do not verify their systems in 2003, and one objective of this research is to allow future engineers to do so.

Paper outline: we begin with a brief overview of the UNITY formalism (§2) followed by an outline of Isabelle/UNITY (§3), which is an UNITY implementation using the Isabelle proof tool [24]. Then we consider three techniques for compositional reasoning in turn: existential/universal properties (§4), guarantees reasoning (§5), and progress sets (§6). We briefly consider some alternative techniques (§7) that other researchers might investigate, and finally present brief conclusions (§8).

2 The UNITY Formalism

A UNITY program is a set of atomic *actions* that operate upon a shared program *state* [20, 21]. An execution step applies an action to the current state, resulting in a new state. Among the actions must be **skip**, which leaves the state unchanged: these are called *stuttering* steps. This requirement simplifies the theory. Actions are chosen nondeterministically. Although traditionally actions have been assumed to be deterministic and total, both of these assumptions are unnecessary: an action can be an arbitrary relation over pairs of states. Intuitively, a program state is simply a map from variables to their values. Formalizing this intuition is difficult: for example, it may require committing ourselves to a particular definition of "value."

Execution of a program F begins in a state satisfying the *initial condition*, written **Init** F. The primitive safety operator is *constrains*, which is abbreviated as **co**. A constrains assertion is like a Hoare triple; a program satisfies A **co** B provided that each of its actions takes any state in A to some state in B. The *stable* assertion **stable** A abbreviates A **co** A: once execution enters the set A it can never leave. An *invariant* assertion is one that holds initially and that is preserved by all actions: **invariant** A means that A is stable and all initial states belong to A.

A state predicate—such as A or B above—is simply a set of states. Program properties such as A **co** B can also be formalized as sets, identifying a property with the set of programs satisfying it. Thus, $F \in A \;\mathbf{co}\; B$ means that the program F satisfies the property A **co** B. We can define **stable** and **invariant** in terms of **co** as follows:

$$\begin{aligned}\mathbf{stable}\, A &\triangleq A \;\mathbf{co}\; A\\ \mathbf{invariant}\, A &\triangleq \{F \mid \mathbf{Init}\, F \subseteq A \land F \in \mathbf{stable}\, A\}\end{aligned}$$

Liveness and progress properties are proved under some fairness constraint. *Weak fairness* is the standard choice. It requires that if some action is continuously enabled (in other words, it could be executed) then the action will be executed infinitely often [20]. In a private communication, Ernie Cohen has recommended *unconditional fairness*. It simply requires that each action must be executed infinitely often. If all actions are total, then weak fairness and unconditional fairness coincide. Non-total actions exhibit behaviour that cannot be implemented. At the extreme is the empty action. Unconditional fairness insists that it be executed infinitely often, which is impossible; therefore, there are no fair traces, and all liveness properties hold vacuously. Non-total actions are analogous to imaginary numbers: we may not know what to do with them, but they lead to interesting mathematics. We have a precedent for them in the "miraculous statements" of the refinement calculus [23]. The choice of unconditional fairness simplifies the theory while increasing its expressiveness.

The primitive progress properties are **transient**, **ensures** and $\mapsto$ ("leads-to"). A program satisfies **transient** A if some action takes A to $\overline{A}$, the complement of A: intuitively, the action falsifies A. The program satisfies A **ensures** B if it takes A to B by an atomic action. It is expressed as follows, where $A \setminus B$ abbreviates $A \cap \overline{B}$:

$$A \;\mathbf{ensures}\; B \triangleq \mathbf{transient}(A \setminus B) \cap ((A \setminus B) \;\mathbf{co}\; (A \cup B))$$

The set satisfying A **ensures** B is the intersection of two other sets of programs:

- **transient**$(A \setminus B)$ is the set of programs that cannot stay in $A \setminus B$ forever.
- $(A \setminus B)$ **co** $(A \cup B)$ is the set of programs that stay in A until they enter B.

The *leads-to* relation, written $A \mapsto B$, is the transitive and disjunctive closure of the **ensures** relation. *Disjunctive closure* means that if $F \in A_i \mapsto B$ for all i in I then $F \in (\bigcup_{i\in I} A_i) \mapsto B$.

UNITY's *substitution axiom* allows any program invariant to be conjoined with state formulas in any UNITY assertion. The intuition is that if, for example, `x` is always an even number, we can use this fact in proofs. Sanders [28] has shown the original form of the axiom to be unsound, but a similar effect can be obtained using rather obvious definitions. Let **reachable**(F) denote the set of states reachable in the program F. *Weak* forms of the various program properties are defined by restricting the original versions to reachable states:

$$\begin{aligned} A \ \mathbf{co}_w\ B &\triangleq \{F \mid F \in (\mathbf{reachable}(F) \cap A)\ \mathbf{co}\ B\} \\ \mathbf{stable}_w\, A &\triangleq A\ \mathbf{co}_w\ A \\ \mathbf{always}\, A &\triangleq \{F \mid \mathbf{Init}\, F \subseteq A \land F \in \mathbf{stable}_w\, A\} \\ A \mapsto_w B &\triangleq \{F \mid F \in (\mathbf{reachable}(F) \cap A) \mapsto B\} \end{aligned}$$

These weak properties satisfy many of the same laws as the strong ones, but are less amenable to compositional reasoning.

The UNITY theory comprises a large number of laws that follow from the definitions and that can be used to reason about programs. Among these is the progress-safety-progress (PSP) law:

$$\frac{F \in A \mapsto A' \qquad F \in B\ \mathbf{co}\ B'}{F \in (A \cap B) \mapsto ((A' \cap B) \cup (B' \setminus B))}$$

Unfortunately, temporal reasoning is often unintuitive. Consider the program whose sole action is `x := x+1`. Obviously, if the current value of `x` is k then eventually that variable's value will be $k + 1$. However, the formal proof of $\mathtt{x} = k \mapsto \mathtt{x} = k + 1$ requires an application of PSP, combining a proof that $\mathtt{x} = k$ must eventually be falsified with a proof that if $\mathtt{x} = k$ now then the next state must satisfy $\mathtt{x} = k$ or $\mathtt{x} = k + 1$. The blame for this convoluted proof lies not with UNITY but with the intrinsic complexity of concurrent systems. Proving $\mathtt{x} = 0 \mapsto \mathtt{x} = k$ is much harder still.

3 UNITY in Isabelle

Isabelle [24] is an interactive proof tool providing a high degree of automation. The *simplifier* performs conditional rewriting and arithmetic reasoning; the *classical reasoner* proves subgoals using tableau methods. Isabelle's Isar language allows proofs to be expressed either in an imperative tactic style, or as readable structured arguments. Its document preparation system automatically typesets formal developments using LaTeX.

Isabelle supports reasoning in a number of different logics. Isabelle/HOL is its instantiation to higher-order logic, while Isabelle/ZF is its instantiation to axiomatic set theory [27]. Higher-order logic is an outstanding formalism for machine verification because of its polymorphic type system. Set theory provides an untyped formalism. Types have been a source of difficulties in our UNITY experiments, so we have built UNITY environments on top of both HOL and ZF.

UNITY is traditionally presented as an axiomatic theory. When it comes to mechanizing the theory for a proof tool, it is best to proceed by formalizing the operational semantics, proving the "axioms" as theorems. Most other researchers also follow this approach [1, 12]. The same mechanisms that let Isabelle support multiple logics also allow the derived UNITY theorems to be used as if they were primitive rules of inference.

Both of the Isabelle UNITY formalizations represent state predicates as sets of states, program actions as relations on states and program properties (such as $A \mapsto B$) as sets of programs. I have previously described a mechanization of UNITY using Isabelle/HOL [25]; the Isabelle/ZF mechanization was derived from the Isabelle/HOL one, so naturally the two versions have much in common. As of this writing, the Isabelle/ZF one is undocumented.

The concept of program state is deliberately left underspecified so that it can be tailored to specific examples. In the Isabelle/HOL version, the entire UNITY theory is polymorphic, which lets each program component define its own type of states. In the Isabelle/ZF version, the UNITY theory refers to an abstract set of states, which users constrain later by asserting axioms to specify the types of variables as they are introduced. Any use of axioms runs the risk of introducing a contradiction, in this case giving a variable two different types. While the risk might be unacceptable in a commercial verification project, it is tolerable for the purposes of research.

4 Existential and Universal Properties

Let X be a program property, perhaps expressing safety or progress. Property X is *existential* when it holds in each system some of whose components satisfy X. The property is *universal* when it holds in each system all of whose components satisfy X. These concepts are not specific to UNITY and can be defined for any notion of system composed of parts. Charpentier and Chandy [6] illustrate them on bags of coloured balls. In the realm of cooking, the property **gluten-free** is universal: for a meal to count as gluten-free, all its ingredients must be organic. The property **contaminated** is obviously existential.

Let us transfer these concepts to UNITY systems. The *composition* $F \sqcup G$ of programs F and G is the program whose set of actions is the union of those of F and G and whose set of initial condition is the conjunction of those of F and G [18]. Composition is only defined if F and G are compatible, a concept that we need not elaborate here. With this definition, composition is obviously commutative and associative. It also has an identity element: the trivial program whose sole action is **skip** and whose initial condition is **true**. We can analogously define the composition $\bigsqcup_{i\in I} F_i$ of the family $\{F\}_{i\in I}$ of programs indexed by the finite, non-empty set I.

Strong safety properties are compositional. If $F \in A$ **co** B and $G \in A$ **co** B then $F \sqcup G \in A$ **co** B, since an action of $F \sqcup G$ is either an action of F or an action of G and therefore takes the precondition A to the postcondition B. The converse direction holds too: we have the equivalence

$$F \sqcup G \in A \ \mathbf{co}\ B \iff (F \in A \ \mathbf{co}\ B) \wedge (G \in A \ \mathbf{co}\ B).$$

Among the other safety properties is the inheritance of strong invariants:

$$\frac{F \in \mathbf{invariant}\ A \qquad G \in \mathbf{invariant}\ A}{F \sqcup G \in \mathbf{invariant}\ A}$$

Thus, A **co** B and **invariant** A are universal properties.

Unfortunately, weak safety properties are not compositional [25, §9.4]. The difficulty is the lack of a simple expression for **reachable**$(F \sqcup G)$ in terms of **reachable**(F) and **reachable**(G).

Progress properties are not compositional in general. We cannot infer $F \sqcup G \in A \mapsto B$ from $F \in A \mapsto B$ and $G \in A \mapsto B$ for the obvious reason that the programs F and G might interfere with one another. However, transient properties are compositional:

$$F \sqcup G \in \mathbf{transient}\ A \iff (F \in \mathbf{transient}\ A) \vee (G \in \mathbf{transient}\ A)$$

Thus, **transient** A is an existential property.

Chandy and Sanders [4, §7.2] describe a simple way to obtain compositional reasoning for progress. They abandon UNITY's traditional primitive progress property, the **ensures** relation. Since **ensures** is a combination of **transient** and **constrains**, it is neither existential or universal. As an alternative base case, they suggest the following:

$$\frac{F \in \mathbf{transient}\ A}{F \in A \mapsto \overline{A}}$$

Because **transient** is an existential property, this inference supports compositional reasoning. The leads-to properties can be combined using transitive and disjunctive closure and using the PSP law mentioned above. However, removing **ensures** from our vocabulary is but a small advance.

The primary problem with relying on existential and universal properties is identifying such properties in the first place. Charpentier and Chandy [6, §4] have shown that for every property X there exists a weakest existential property $\mathbb{WE}(X)$ that implies X. This is the best we could hope for: an existential property that is strong enough to imply X and no stronger. They define $\mathbb{WE}(X)$ to be the union of all existential properties that are stronger than X; the union is obviously stronger than X, and the point is that it is also an existential property. Charpentier also shows that there is no weakest universal operator. Coming up with universal properties therefore requires creativity. In a separate paper, Charpentier and Chandy [5] present two proofs involving universal properties in order to demonstrate this creative process.

- Their first proof concerns a toy example. There is a set of components, each with a separate counter but also sharing a global counter. We must prove a safety property: that the global counter equals the sum of the local counters.

- The other verification, which is more complicated, concerns a system of processes with conflicting priorities. A directed graph represents the constraints. We must prove a liveness property: that every process will eventually get the top priority.

My colleague Sidi Ehmety has mechanized all this material using Isabelle/UNITY [11]. The theory of the weakest existential property was almost trivial to mechanize: several pages of hand proofs collapsed down to a few lines of Isabelle proof script. The examples involving universal properties were also easy to mechanize. The proof script for the toy example comprises 15 theorems, each proved using one or two commands. Many of the proofs are simple inductions or immediate from the definitions. The final compositional proof is trivial, as the authors intended it should be. The priority system example rests on a theory that defines basic concepts of graphs. Not counting this preliminary theory, the development of comprises 30 theorems. Again most of them are proved using one or two commands, although few of the proofs are a bit longer. The compositional part of the reasoning comprises three lemmas whose statements and proofs fit on single screen. Thus, the complicated example turns out to be only slightly more difficult to handle than the toy one.

Our experience with existential and universal properties is positive. The dramatic collapse in proof length in the theory of weakest existentials is not unusual in mechanical developments of meta-theory. It would be nice if we could see such a reduction in verifications of actual systems, but at least no special difficulties emerged.

5 Guarantees and the Allocator Example

If X and Y are program properties, then so is X **guarantees** Y. The program F satisfies X **guarantees** Y provided for every program G, if $F \sqcup G$ satisfies X then $F \sqcup G$ also satisfies Y. Charpentier and Chandy have shown that **guarantees** can also be expressed in using the weakest existential operator: X **guarantees** Y is the weakest existential property that is stronger than X implies Y. Formally, if we regard program properties as sets of programs, then X **guarantees** Y equals $\mathrm{WE}(\overline{X} \cup Y)$.

Since the guarantees operator is just an example of an existential property, one might think that the previous section has covered all relevant issues. This view would be incorrect for two reasons: first, the guarantees operator has a specialised theory of its own, and second, the main example demonstrating its use highlights several other issues that are relevant to compositional reasoning as a whole.

The example is Chandy and Charpentier's *token allocation system* [2]. A family of clients request tokens from a central allocator, which attempts to meet those requests from the remaining supply of tokens. Requests and responses are delivered over a network. The family of clients can be presented to the central allocator as a single, virtual client: merging and distribution networks channel messages to and from the real clients. The components are assumed to be well-behaved: they never make unreasonable requests and they respond to all reasonable requests. The objective is to prove that the system as a whole is well-behaved. All client requests should eventually be met, and the number of tokens allocated should never exceed the maximum allowed to exist.

Compared with any real-world system, this resource allocation example is still a toy. Nevertheless, it is much more complicated than most other examples in the literature. The issues it raises include the following:

- *The representation of component states.* Each component has some private variables and therefore has its particular view of the state. Imposing a uniform state on all programs is undesirable, especially if we hope that components can be re-used. If there is no uniform state representation, then we need a means of establishing a common one for the system at hand.

- *The replication of components.* There is a family of clients, each client with a unique identity, but otherwise alike. It is natural to specify a single client and to generate the family by replication. This is easy if each system component has its own state representation, because the state representation of a family of components is a function space. However, with a global state representation, replication seems to require some form of subscripting.

Ehmety and I have spent months grappling with these problems. There are a number of different approaches, but they fall into one of two categories:

- Each component can have its own state representation, and they are combined as the system is assembled. Program properties must be transferred between a component's state representation and the system one. If we are to use a property of the form X **guarantees** Y, then we must be able to transfer both X and Y from system states to component states and back again. The difficulty lies in performing these transfers.

- A uniform, global state representation is imposed. The difficulty is that the formal definitions may be overly rigid, allowing only a predetermined collection of data types.

The simplest approach is strongly typed and represents each component's state by a record enumerating that component's variables [26]. The system state will be a similar record, but with more variables. Because the representations differ, not all program properties can be transferred. Those that can be transferred include safety properties, both weak and strong. Liveness properties are difficult to transfer. It helps to adopt unconditional fairness, which defines the same behaviour as weak fairness on real programs while having better theoretical properties. It allows liveness properties to be transferred from component states to system states,[1] although not in the opposite direction. It follows that we can handle X **guarantees** Y where Y involves liveness, although X can only involve safety. This restriction is serious and precludes a successful treatment of the allocator example.

A variation on this strongly typed approach is to include in each record a dummy variable polymorphic type [26]. This variable models the unknown part of the state and is instantiated appropriately as the components are assembled. In effect, all components share the common state representation. Program properties can be transferred without restriction, including guarantees properties.

To adopt a uniform state representation, we can define a state to be a map from variable names to values. Many researchers working with hand proofs take this representation for granted, but I find it neither natural nor convenient. However, Ehmety and I have built a UNITY environment using Isabelle/ZF to support this state representation. We set out to improve upon Vos [29], who gave an earlier formalization with a uniform state representation. Because ZF is untyped, we did not have to define a value space using disjoint sums, with their attendant injection and projection functions. Although Vos worked in higher-order logic, her use of

---

[1] We can interpret this result as saying that liveness properties can similarly be transferred even under weak fairness, for programs containing only total actions—that is, for all real programs.

a disjoint sum made her formalization untyped in spirit: every variable had type `Val`.

In our ZF/UNITY environment, users do not have to specify types of variables initially. A variable's "type" can be any ZF set, such as the set of natural numbers. Ehmety has found an ingenious method of making well-typing implicit, eliminating the need to conjoin all state predicates with a well-typing predicate. Despite this device, proofs are more difficult in the ZF version of UNITY than they are in the HOL version. Compared with Isabelle/ZF, Isabelle/HOL has better arithmetic support and bigger lemma libraries, and it benefits from strong typing.

Ehmety [10] has investigated an unusual approach that combines a global state representation with strong typing. The idea is due to Stefan Merz [17], who used it in his implementation of TLA in Isabelle. It involves regarding the state space as an abstract type with a coalgebraic structure. Instead of defining a specific state type, we describe its desired properties axiomatically. Each variable is a state inspector and the axioms ensure that variables can be updated independently. We performed some experiments using this approach, but found the axioms too unintuitive.

Our conclusion is that reasoning about **guarantees** is straightforward, but resolving the problem of state representation is not. On balance, the best approaches are (1) individual state representations with polymorphic dummy variables, or (2) Merz's abstract state types. Both of these suggestions require complicated, unintuitive definitions. This problem can be addressed by building a front end that accepts some verification formalism. We have conducted all our experiments using pure logic. If we wish to make this work available to a wider class of users, then it needs to be integrated with a development environment, which could take care of the ugly details. Such an environment would be based not on UNITY—which is too simple for use in real-world development—but on some richer temporal logic.

6 Proving Non-Interference with Progress Sets

Traditionally, compositional reasoning has often meant showing that a program continues to satisfy a certain property even when it is composed with another program. A recent contribution to this tradition is the method of *progress sets*, due to Meier and Sanders [15, 16]. A progress set is a set of state predicates: in other words, a set of set of states. It must be a complete lattice (closed under intersections and unions) and satisfy a number of closure conditions. Among the parameters in the definition of progress set is the state predicate T, which identifies the set of program states currently of interest. The point of the method is the *progress set union theorem*, whose premises are as follows:

- F is a program that satisfies the progress property $A \mapsto B'$

- C is a progress set for the parameters F, T and B
- $B \subseteq B'$ and $B' \in C$
- F satisfies **stable** T, meaning that no F-action leaves T
- G satisfies $(X \setminus B)$ **co** X for all $X \in C$

The conclusion is that $F \sqcup G$ satisfies $T \cap A \mapsto B'$.

This is a traditional non-interference result: if F makes progress and G satisfies a safety property, then their composition $F \sqcup G$ satisfies something close to the original progress property. Obviously G must contribute liveness properties to $F \sqcup G$, since otherwise it serves no purpose. Sometimes F makes progress and G has to wait, while other times G makes progress and F has to wait. The parameter T facilitates this reasoning by identifying the condition under which it is F's turn to make progress. In order to reason about the progress made by G, we must exchange the roles of F and G in the theorem. We eventually obtain several leads-to properties about $F \sqcup G$ that we can combine in order to derive the desired properties of the system. Compared with **guarantees**, which can relate arbitrarily complex program properties, reasoning by non-interference is rather low-level.

I have mechanized the theory of progress sets using Isabelle/HOL, along with an underlying theory of predicate transformers for UNITY. The mechanization was straightforward, although not trivial. I found some errors in the proofs and even in the definitions. Unfortunately, the papers on progress sets [15, 16] include no interesting examples. The only nontrivial example is a generalisation of the dining philosophers problem, and its treatment is only sketched.

The main advantage of the progress set method is that it subsumes a number of other methods for proving non-interference. Although I have mechanized the formal development, I cannot claim to have grasped the intuitions behind progress sets and I have no idea how to apply them in a program proof. An expository paper by the method's advocates would remedy this situation.

7 Methods Not Investigated

This section briefly discusses a number of methods for compositional reasoning that—for a variety of reasons—were not deeply investigated.

The *conditional properties* of Misra [18] represent the traditional rely/guarantee or assumption/commitment reasoning. If X and Y are arbitrary program properties, then a typical conditional property for a program F states that for every G that satisfies X, the combined program $F \sqcup G$ satisfies Y. Recall that a Chandy-Sanders guarantee is slightly different: if $F \sqcup G$ satisfies X then

$F \sqcup G$ satisfies Y. Such guarantees are easier to use because all reasoning refers to the complete system $F \sqcup G$. The advantage becomes clearer when we consider a system of the form $F_1 \sqcup F_2 \sqcup \ldots \sqcup F_k$: assumption/commitment forces us to consider the k "environments" $F_2 \sqcup \ldots \sqcup F_k$, etc.

Assumption/commitment guarantees yield a natural style of reasoning. Countless researchers have investigated it, which is one reason for us not to do the same. At a formal level, the rules governing assumption/commitment guarantees are similar to those for Chandy-Sanders guarantees. Our experience is that the latter works quite well with mechanical proof tools and is safe to assume that the former type of guarantee would work equally well. It is also clear that the state representation issues discussed above are equally pertinent with both types of guarantees property.

Misra has proposed *closure properties* [19] as providing a more convenient style of reasoning than assumption/commitment guarantees. Let X be a program property. Then to say that F satisfies the closure of X is to say that $F \sqcup G$ satisfies X for all G. The closure of X is simply **true guarantees** X, with either interpretation of guarantees. In fact, the closure of X is equivalent to $\mathbb{WE}(X)$, the weakest existential property stronger than X. Thus, closure properties appear to be subsumed by the forms of compositional reasoning that we have investigated. However, Misra's concept includes various type-checking and linking procedures that may introduce additional expressive power.

Continuing with the work of Misra, we come to his programming language Seuss [22], whose purpose is to ensure that concurrent programs behave no differently from sequential ones:

> Seuss fosters a discipline of programming that makes it possible to understand a program execution as a single thread of control, yet it permits program implementation through multiple threads. ... A central theorem establishes that multiple execution threads implement single execution threads, i.e., any property proven for the latter is a property of the former as well.

Recall that our project is concerned with compositional reasoning about concurrent systems. Seuss has no explicit concurrency primitives: its objective is to demonte concurrency from a programming concept to an implementation detail.[2] For this reason, Seuss lies outside the scope of our project.

Possibly worth investigating is the *achievement* property proposed by Ernie Cohen [8]. Achievement is related to leads-to: the concepts coincide when the postcondition is stable. The main advantage of achievement is that under certain conditions, a system inherits the achievement properties of its components. The

---

[2] More information is availble from `http://www.cs.utexas.edu/users/psp/welcome.html`.

theory is relatively straightforward and it should not be difficult to mechanize. However, this work is for the future.

8 Conclusions

We have investigated several proposals for compositional reasoning about concurrent programs, focusing on their suitability for use with mechanical proof tools. Many of our difficulties stem from assumptions prevalent in the world of hand proofs that are not appropriate for machine proof. The representation of program states remains a major issue: the requirement for a global program state is not easily achieved with mechanical proof tools. However, it certainly can be achieved if enough effort is invested.

One unnecessary obstacle to progress is that many UNITY researchers use the notation and proof style of Dijkstra and Scholten [9]. This notation does address a real issue, namely that while predicate transformers act upon sets, sometimes one would rather work with formulas. This issue could easily be addressed by introducing a simple abbreviation for the set determined by a given formula, say $[p]$ to abbreviate $\{s \mid s \text{ satisfies } p\}$. Instead, we are expected to use a notation in which logical symbols sometimes behave as set operators and sometimes not, depending upon their context. For example, $A \Rightarrow B$ might mean $A \to B$ (Boolean implication), $A \subseteq B$ (the subset relation) or even $\overline{A} \cup B$. More than once I have discovered that, after decipherment, the underlying formula is a well-known law of set-theory and the accompanying proof is redundant. The dogma also demands that all proofs be written in a linear fashion, whether it suits them or not; the result can be baffling. If the UNITY community is to grow, the first step is to adopt standard mathematical notation and terminology.

Many of the proposed methods for compositional reasoning appear to work. They may not be convenient when expressed in the logic of a theorem prover, but with a little implementation effort, the underlying complications can be hidden. Much of the effort in this project has gone to addressing other problems. One of the main problems is simply that temporal reasoning for program components is counter-intuitive. We have come full circle: having come to a reasonable understanding of reasoning about monolithic programs, we have investigated ways of reasoning about systems of programs. These reasoning methods work acceptably well, in that most of our time has been spent trying to prove properties of the components. In other words, we should be focusing on more intuitive and more productive ways of reasoning about program components, especially proving progress properties of components. We also need more examples. The Allocation System is the only big example we have investigated, and other examples might highlight other problems.

Acknowledgements. Much of the work reported above was done by Sidi Ehmety. It has benefited from many discussions with Michel Charpentier and Ernie Cohen. The research was funded by the U.K.'s Engineering and Physical Sciences Research Council, grant GR/M75440 "Compositional Proofs of Concurrent Programs."

References

[1] Flemming Andersen, Kim Dam Petersen, and Jimmi S. Pettersson. Program verification using HOL-UNITY. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: HUG '93*, LNCS 780, pages 1–15. Springer, 1994.

[2] K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, 2002.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[4] K. Mani Chandy and Beverly A. Sanders. Reasoning about program composition. Technical Report 2000-003, CISE, University of Florida, 2000. available via `http://www.cise.ufl.edu/~sanders/pubs/composition.ps`.

[5] Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In José Rolim, editor, *Parallel and Distributed Processing*, LNCS 1586, pages 1215–1227, 1999. Workshop on Formal Methods for Parallel Programming: Theory and Applications.

[6] Michel Charpentier and K. Mani Chandy. Theorems about composition. In R. Backhouse and J. Nuno Oliveira, editors, *Mathematics of Program Construction: Fifth International Conference*, LNCS 1837, pages 167–186. Springer, 2000.

[7] Michel Charpentier and K. Mani Chandy. Specification transformers: A predicate transformer approach to composition. *Acta Informatica*, 2003. in press.

[8] Ernie Cohen. Asynchronous progress. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*. Springer, 2002.

[9] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.

[10] Sidi O. Ehmety and Lawrence C. Paulson. Representing component states in higher-order logic. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, number EDI-INF-RR-0046 in Informatics Report Series, pages 151–158. Division of Informatics, University of Edinburgh,

September 2001. Online at http://www.informatics.ed.ac.uk/publications/report/0046.html.

[11] Sidi O. Ehmety and Lawrence C. Paulson. Program composition in Isabelle/UNITY. In *Parallel and Distributed Processing*. IEEE, 2002. Workshop on Formal Methods for Parallel Programming: Theory and Applications; text on CD-ROM.

[12] Barbara Heyd and Pierre Crégut. A modular coding of UNITY in COQ. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs '96*, LNCS 1125, pages 251–266. Springer, 1996.

[13] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] Leslie Lamport. Composition: A way to make proofs harder. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, pages 402–423. Springer, 1998. LNCS 1536.

[15] David Meier. *Progress Properties in Program Refinement and Parallel Composition*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1997.

[16] David Meier and Beverly Sanders. Composing leads-to properties. *Theoretical Computer Science*, 243(1-2):339–361, 2000.

[17] Stephan Merz. Yet another encoding of TLA in Isabelle. On the Internet at http://www.loria.fr/~merz/projects/isabelle-tla/doc/design.ps.gz, 1999.

[18] Jayadev Misra. Asynchronous compositions of programs. At URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/composition.ps.Z, apr 1994.

[19] Jayadev Misra. Closure properties. At URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/closure.ps.Z, sep 1994.

[20] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995. Also at URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/progress.ps.Z.

[21] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995. Also at URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/safety.ps.Z.

[22] Jayadev Misra. *A Discipline of Multiprogramming*. Springer, 2001.

[23] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[25] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, 2000.

[26] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 25(5):626–656, 2001.

[27] Lawrence C. Paulson and Krzysztof Grąbczewski. Mechanizing set theory: Cardinal arithmetic and the axiom of choice. *Journal of Automated Reasoning*, 17(3):291–323, December 1996.

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

[29] Tanja E. J. Vos. *UNITY in Diversity, a Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Utrecht University, 1999.