Under consideration for publication in Formal Aspects of Computing

Mechanizing Compositional Reasoning for Concurrent Systems: Some Lessons

Sidi O. Ehmety[1] and Lawrence C. Paulson[2]

[1]Faculté des Sciences et Techniques, Université de Nouakchott, BP. 5026, Nouakchott, Mauritania
[2]Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, England

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 results also apply to related proposals such as traditional assumption-commitment guarantees and Misra's closure properties. Findings that have been published in detail elsewhere are summarised and consolidated here. One conclusion is that UNITY and related formalisms leave some important issues implicit, such as their concept of the program state, which means that great care must be exercised when implementing tool support. Another conclusion is that many compositional reasoning methods can be mechanized, provided that the issues mentioned above are correctly addressed.

Keywords: UNITY, Isabelle, compositional reasoning, existential properties, universal properties, guarantees assertions

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.

For reasoning to count as compositional, it is not enough that properties of program components can be inherited or combined. For example, **transient** properties (defined below) have a simple inheritance rule. Unfortunately, each **transient** property refers to a specific atomic command. Thus, it depends upon

Correspondence and offprint requests to: L. C. Paulson the component's implementation. Compositional reasoning should depend upon the components' abstract properties only.

The present work is in the context of concurrent systems and uses the UNITY formalism [CM88]. 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 [NPW02]. 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 [CS00, CC03]
- progress sets [MS00]

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, we should address Leslie Lamport's points in his paper "Composition: a way to make proofs harder" [Lam98]. 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) [Lam94]. The TLA 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: the paper begins 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 [NPW02]. 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

UNITY provides a simple guarded-command language and an equally simple temporal logic, with a collection of easily-grasped inference rules. A UNITY program is a set of atomic *actions* that operate upon a shared program *state*. 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. Stuttering models the possibility that the program does nothing in the current time interval, and the inclusion of **skip** simplifies the theory. Actions are chosen nondeterministically. Although the original UNITY book [CM88] assumes actions to be deterministic and total, both of these assumptions appear to be unnecessary. My Isabelle formalization treats an action as an arbitrary relation over pairs of states and defines the corresponding transition semantics. All the usual UNITY laws can be derived from this definition.

When formalizing UNITY, a crucial issue is that of the program state. Informal presentations of UNITY seldom define this concept. 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.” The issue of states becomes critical when we consider composition, because typically different program components will have different sets of variables, and therefore different views of the state. We outline the problems that can occur in an earlier paper [Pau01] and again below, §5.

2.1. Safety Properties

Execution of a program F begins in a state satisfying the *initial condition*, written $\textbf{Init}\,F$. The primitive safety operator is *constrains*, which is abbreviated to **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 $\textbf{stable}\,A$ 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: $\textbf{invariant}\,A$ 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, $F \in A\ \textbf{co}\ B$ means that the program F satisfies the property A **co** B. We can define **stable** and **invariant** in terms of **co** as follows:

$$\textbf{stable}\,A \triangleq A\ \textbf{co}\ A$$

$$\textbf{invariant}\,A \triangleq \{F \mid \textbf{Init}\,F \subseteq A \land F \in \textbf{stable}\,A\}$$

2.2. Liveness Properties

Liveness and progress properties are proved under some fairness constraint, which imposes restrictions on execution traces. *Weak fairness* is the standard choice. First, we need some definitions. An action is *enabled* provided the current state belongs to the domain of that action (which is simply a relation). An action is *continuously enabled* in a trace if it is enabled now and at every future point. Now, weak fairness allows only those traces such that each continuously enabled action is executed infinitely often [Mis01, §6.3.2]. In a private communication, Ernie Cohen has recommended *unconditional fairness*. It simply requires that each action must be executed infinitely often, whether enabled or not. If all actions are total, then actions are always enabled; thus, 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 [Mor94]. 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 $\textbf{transient}\,A$ 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 $A \cap \overline{B}$:

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

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

- $\textbf{transient}(A \setminus B)$ is the set of programs that cannot stay in $A \setminus B$ forever.
- $(A \setminus B)\ \textbf{co}\ (A \cup B)$ 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 $F \in (\bigcup_{i \in I} A_i) \mapsto B$.

2.3. Weak Versions of the Properties

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 [San91] 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 forms of properties satisfy many of the same laws as the strong ones, but are less amenable to compositional reasoning.

The UNITY theory includes numerous 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, requiring a quirky form of induction.

3. UNITY in Isabelle

Isabelle [NPW02] 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 [PG96]. 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 Ehmety has built a UNITY environment on top of ZF to augment the existing HOL one. Because Isabelle is generic, Ehmety was able to take the HOL formalization as a starting point, modifying the proof scripts according to the differences in the ZF version. A generic proof tool encourages the re-use of developments in different logics.

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 [APP94, HC96]. 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. Paulson has previously described a mechanization of UNITY using Isabelle/HOL [Pau00]; because the Isabelle/ZF mechanization was derived from the Isabelle/HOL one, they 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 [CC00] illustrate them on bags of coloured balls. In the realm of cooking, the property **nutritionally balanced** is universal: a meal prepared entirely of nutritionally balanced ingredients will itself be nutritionally balanced. The property **organic** is *strongly universal*: a meal is organic if and only if all of its ingredients are organic. The property **contaminated** is strongly 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 initial condition is the conjunction of those of F and G [Mis01, §8.2]. 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.

The composition $F \sqcup G$ is only defined if F and G are *compatible*. This typically means that they assign the same types to common variables and respect one another's private variables. Some authors also require that the initial conditions are not disjoint, since otherwise $F \sqcup G$ would have an empty (false) initial condition; the Isabelle formalizations ignore this requirement. Compatibility turns out to be easy to formalize, so we need not consider this issue any further.

4.1. Compositionality and Safety Properties

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

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

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

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

To summarize, **invariant** A is a universal property, while A **co** B is a strongly universal property.

Unfortunately, weak safety properties are not compositional [Pau00, §9.4]. The difficulty is the lack of a simple expression for $\mathbf{reachable}(F \sqcup G)$ in terms of $\mathbf{reachable}(F)$ and $\mathbf{reachable}(G)$.

4.2. Compositionality and Liveness Properties

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:

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

Thus, **transient** A is strongly existential.

Chandy and Sanders [CS00, §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 inferring a specific instance of leads-to from a **transient** property:

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

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.

One can argue that reasoning about **transient** A is not compositional, even though it is an existential property. Let us recall two points:

- Compositional reasoning means proving properties of a system from those of its components without reference to implementations.
- A program satisfies **transient** A if it has an action that takes A to $\overline{A}$.

Thus, we see that **transient** A refers to the existence of a specific atomic action in the system. Replacing the atomic action by a sequence of two simpler actions could falsify **transient** properties while preserving more abstract progress properties. There is room for different points of view. Pragmatists will note that reasoning about a system's **transient** properties is easy, even as purists decry the slight dependence on implementations.

4.3. The Weakest Existential Operator

The primary problem with relying on existential and universal properties is identifying such properties in the first place. Charpentier and Chandy [CC00, §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:

$$\bigcup\big(\{Y \mid Y \subseteq X \land Y \text{ is existential}\}\big).$$

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.

4.4. Case Studies Involving Universal Properties

Because there is no weakest universal operator, definging universal properties requires creativity. In another paper, Charpentier and Chandy [CC99] 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.

Ehmety has mechanized all this material using Isabelle/UNITY [EP02]. 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 comprises 30 theorems. Again most of them are proved using one or two commands, although a few of the proofs are a bit longer. The compositional part of the reasoning comprises three lemmas whose statements and proofs fit on a 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 $\mathbb{WE}(\overline{X} \cup Y)$.

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* [CC02]. 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 components (including the network) 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 more complicated than many other examples in the literature. Abstract implementation issues can be examined. For instance, 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 allocation system raises some important issues:

- *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 appealing 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: either a way to write specifications involving subscripted variables, or an operation to attach subscripts to variable names in existing specifications. Replication arises in many examples, notably the Dining Philosophers problem [CM88, Chap. 12]: the "philosopher" processes are essentially identical.

We have spent a long time 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.

5.1. Local States As Strongly-Typed Records

The simplest representation of states uses strong typing. Each component's state is a record enumerating that component's variables [Pau01]. The system state will be a similar record, but with more variables. Informal notations can be misleading. Suppose that a specification involves the formula $x \leq y$, where x and y are the component's only variables. Suppose that the system state introduces a third variable, z. When we transfer the formula to the extended state type, the informal notation is still $x \leq y$. The crucial but invisible difference is that it is now a formula in the variables x, y and z; a better notation might be $x \leq y \wedge z = z$. Introducing the variable z changes the language in which we write specifications. Because the difference is invisible, we can overlook its importance.

Using strongly typed records seems natural, but it greatly complicates the use of **guarantees** assertions. We must must transfer each component's guarantee to the system state type before we can use it to reason about the system. Suppose a program component satisfies X **guarantees** Y and we wish to derive another guarantee, say X' **guarantees** Y', at the extended state type. The derivation requires proving two assertions [Pau01, §5.1]:

1. If the system satisfies X', then the projected system satisfies X. (The *projected system* refers to the system as seen from the component's state type.)
2. If the projected system satisfies Y, then the (original) system satisfies Y'.

In other words, we need to transfer the property X' down to another property X of the projected system, apply the component's guarantee, and finally transfer its conclusion Y back to the full system state type, obtaining Y'.

Here, X, X', Y, Y' are typically safety or liveness properties; it turns out that not all such properties can be transferred [Pau01, §6]. 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. Under unconditional fairness, liveness properties can be transferred from component states to system states. (We can interpret this result as saying that liveness properties can similarly be transferred even under weak fairness, at least for programs containing only total actions—that is, for all real programs.) Unfortunately, even under unconditional fairness, liveness properties cannot be transferred from system states to component states.

In essence, our problem is how to apply a component's guarantee in the more expressive language of the full system. The results outlined above allow us to apply a component's property X **guarantees** Y where X can only involve safety, although where Y may involve liveness. The restriction on X is serious. It precludes a natural treatment of any example in which a system makes progress through the co-ordinated actions of several components. In such cases, the precondition of the guarantees property will involve progress. If only safety preconditions are permitted, then guarantees is being used only to express non-interference with the component's progress.

A variation on this strongly typed approach is to include in each record a dummy variable polymorphic type [Pau01, §7]. 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. The drawback is that instantiating the dummy variables is tricky, although it could be automated through a dedicated front-end.

5.2. Global States: Two Approaches

To adopt a uniform state representation, we can define a state to be a map from variable names to values. Many researchers working with semi-formal notations take this representation for granted, but we find it neither natural nor convenient. However, we have built a UNITY environment using Isabelle/ZF to support this state representation. We set out to improve upon Vos [Vos99], 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 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. The function `type_of` maps variables to their types; this function is initially left unspecified, and types of variables are introduced individually by axioms such as `type_of(x)=int`. A valid state is one that assigns to each variable v an element of `type_of(`v`)`. We prove once and for all that our program is well-typed, namely that it maps valid states to valid states; we can then rely on variables having the correct types in all other reasoning. Despite this device, proofs are harder in the ZF version of UNITY than they are in the HOL version. There are several reasons: Isabelle/HOL has better arithmetic support and bigger lemma libraries, and it benefits from strong typing.

An objection to `type_of` is that it requires the use of axioms rather than definitions. Proof tool users generally prefer to avoid asserting axioms because of the risk of inconsistency. However, the axioms governing `type_of` are simply an incremental definition by cases. Isabelle, like most proof tools, does not allow a function to be defined incrementally. The solution is either to allow such definitions or (again) to write a dedicated front-end that generates the axioms safely.

Ehmety [EP01] has investigated an unusual approach that combines a global state representation with strong typing. The idea is due to Stefan Merz [Mer99], 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. To summarise, we have three options:

1. Individual state representations with polymorphic dummy variables. This option requires a proof tool that supports polymorphism.
2. Merz's abstract state types.
3. A formalization of UNITY within set theory. This option provides an untyped verification environment and it obviously requires a proof tool that supports set theory.

All of these suggestions require axioms or complicated, unintuitive definitions, and the solution is to implement a dedicated front-end. We have conducted all our research using pure logic, but if program proving is to become practical, it will require not a general-purpose theorem prover but a specialized verification tool that takes care of the ugly details. Such a tool 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 F continues to satisfy a certain property even after it is composed with another program, G. In other words, G does not interfere with F. A recent contribution to this tradition is the method of *progress sets*, due to Meier and Sanders [Mei97, MS00]. 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 other state predicates are A, B' and B.) 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 $\textbf{stable}\, T$, meaning that no F-action leaves T
- G satisfies $(X \setminus B)\ \textbf{co}\ X$ 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, non-interference arguments are rather low-level. They fall somewhat short of the ideal of compositional reasoning.

Paulson has mechanized the theory of progress sets using Isabelle/HOL, along with an underlying theory of predicate transformers for UNITY. The mechanization was straightforward, but surprisingly difficult considering that this material was meta-theory and was published together with detailed proofs. He found errors in the proofs and even in the definitions. Unfortunately, the papers on progress sets [Mei97, MS00] 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 we have mechanized the formal development, we cannot claim to have grasped the intuitions behind progress sets and we have no idea how to apply them in a program proof. An expository paper by the method's advocates would remedy this situation.

7. Other Methods

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 [Mis01, 9.2] 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* [Mis01, §9.3] 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, whose purpose is to ensure that concurrent programs behave no differently from sequential ones [Mis01, p. 3]:

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 demote concurrency from a programming concept to an implementation detail.[1] For this reason, Seuss lies outside the scope of our project.

Possibly worth investigating is the *achievement* property proposed by Ernie Cohen [Coh02]. 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 theory is relatively straightforward and it should not be difficult to mechanize. However, this work is for the future.

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

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 [DS90]. 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 $A \rightarrow B$ (Boolean implication), $A \subseteq B$ (the subset relation) or even $\overline{A} \cup B$. More than once we 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 existing methods for compositional reasoning appear to be easily mechanizable. They may not be convenient when expressed in the logic of a theorem prover, but the underlying complications could be hidden with a little implementation effort. Much of the work in this project has gone to addressing other problems, one being simply that temporal reasoning is counter-intuitive. We have come full circle: having done some experiments on mechanical reasoning about monolithic programs, we have looked into mechanical reasoning about systems of programs. We have developed mechanical procedures that work acceptably well. Proofs are difficult, chiefly because of their intrinsic complexity. Specialized support for temporal reasoning can only help matters, but this support can be aimed at providing more productive ways of reasoning about program components, especially proving progress properties of components. Specialized support for composition does not seem necessary apart from the simple proposals given at the end of §5. We also need more examples. The Allocation System is the only big example we have investigated, and other examples might highlight other problems.

Acknowledgements. The work reported above 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

[APP94] 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.

[CC99] 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.

[CC00] Michel Charpentier and K. Mani Chandy. Theorems about composition. In R. Backhouse and J. Nuno Oliveira, editors, *Mathematics of Program Construction: 5th International Conference, MPC 2000*, LNCS 1837, pages 167–186. Springer, 2000.

[CC02] K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, 2002.

[CC03] Michel Charpentier and K. Mani Chandy. Specification transformers: A predicate transformer approach to composition. *Acta Informatica*, 2003. in press.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Coh02] Ernie Cohen. Asynchronous progress. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*. Springer, 2002.

[CS00] 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.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.

[EP01] 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.

[EP02] 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.

[HC96] 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.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam98] 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.

[Mei97] David Meier. *Progress Properties in Program Refinement and Parallel Composition.* PhD thesis, Swiss Federal Institute of Technology Zurich, 1997.

[Mer99] 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.

[Mis01] Jayadev Misra. *A Discipline of Multiprogramming.* Springer, 2001.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, second edition, 1994.

[MS00] David Meier and Beverly Sanders. Composing leads-to properties. *Theoretical Computer Science*, 243(1-2):339–361, 2000.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.

[Pau00] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, 2000.

[Pau01] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 25(5):626–656, 2001.

[PG96] 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.

[San91] Beverly Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Vos99] Tanja E. J. Vos. *UNITY in Diversity, a Stratified Approach to the Verification of Distributed Algorithms.* PhD thesis, Utrecht University, 1999.