Program Composition in Isabelle/UNITY

Sidi O. Ehmety and Lawrence C. Paulson
Cambridge University Computer Laboratory
J J Thomson Avenue – Cambridge CB3 0FD – England
Tel. (44) 1223 763584 – Fax. (44) 1223 334678
{soe20,lcp}@cl.cam.ac.uk

Abstract

We describe the mechanization of recent examples of compositional reasoning, due to Charpentier and Chandy [4]. The examples illustrate a new theory for composition proposed by Chandy and Sanders [2, 3], based on the so-called existential *and* universal *properties. We show that, while avoiding hand proof mistakes, a such compositional reasoning can be mechanized quite straightforwardly.*

We also present the mechanization of some theoretical results [5] concerning existential properties and their relation with the guarantees *concept. The result is a new module added to the existing Isabelle/UNITY theory for composition.*

Keywords: *program composition, universal and existential properties, UNITY, Isabelle.*

1. Introduction

Compositional proof is one of the many methods that have been introduced for reasoning about concurrent systems. It explores the possibility of deriving a system's properties from those of its components.

Paulson has mechanized a version of the concurrent programming formalism UNITY [6, 7] and extended it with Chandy and Sanders's compositional theory [2, 3], using the Isabelle [8] proof tool. The resulting system, which includes all relevant definitions and basic theorems of UNITY as well as numerous illustrative examples, is what we call Isabelle/UNITY [9].

The present work is a continuation of Paulson's. It concerns the Isabelle/UNITY theory for composition. We mechanize two examples illustrating universal and existential composition. A *universal* property is one that holds in a system provided all components have the property. An *existential* property is one that holds in a system provided some components have the property.

We tried to be as close as possible to the original formalization. However, we have made improvements and corrected mistakes. In addition to simplifications, we propose improvement in the specification regarding treatment of variables (visibility conditions). This improvement leads to more abstract specifications. Furthermore, unlike the original formalization, where systems are described axiomatically, we have followed a definitional approach: systems have been first defined and then proved to be correct, i.e. they satisfy their specification. In the second example the definition has suggested a stronger specification. In the same example, the proof of one auxiliary theorem was wrong and we have had to invent one.

The theoretical issues we have mechanized concern the existence of the *weakest existential property*, which is shown [5] to be sufficient for expressing guarantees. A variant of rely/guarantee specification, the *guarantees* primitive is useful for making assumptions over environments. The mechanization has a high degree of automation: in a few lines we achieve 5 pages of proofs.

Paper outline. The paper begins with an overview of UNITY (Section 2). It describes the two examples of Charpentier and Chandy [4] (Sections 3 and 4) then the theoretical results of [5] (Section 5). Finally, the paper concludes (Section 6).

2. Isabelle/UNITY

The UNITY formalism is made of a programming language and an associated logic. Isabelle/UNITY is a set-theoretic interpretation of UNITY in higher-order logic. Thanks to the well-known equivalence between sets and predicates, the correspondence between the two formalisms is straightforward. The difference is basically notational: predicates (p, $p \vee q$, $p \rightarrow q$, $\forall x\, p$, $\ldots$) vs. sets (A, $A \cup B$, $\overline{A} \cup B$, $\bigcap x\, A$, $\ldots$). For further discussion the reader can refer to Paulson [9]. We continue our presentation using set theory notation.

UNITY programs. A program consists of an *initial condition* and a set of *atomic guarded-assignments* (actions). Both are expressed over a common set of *declared variables*. The set of actions always contains *skip*, the action that does no state changes. The execution model resembles an infinite loop during which actions are selected nondeterministically for execution.

UNITY logic. The logic divides program properties into two categories: *safety* and *progress*. Safety properties include co[1] (constrains), stable and invariant:

$$\begin{array}{lll} A\,\texttt{co}\,B & \equiv & \{F \mid \forall act \in \texttt{Acts}\,F.\, act``A \subseteq B\} \\ \texttt{stable}\,A & \equiv & A\,\texttt{co}\,A \\ \texttt{invariant}\,A & \equiv & \{F \mid \texttt{Init}\,F \subseteq A \wedge F \in \texttt{stable}\,A\} \end{array}$$

where $\texttt{Init}\,F$ and $\texttt{Acts}\,F$ denote the initial condition and the set of commands of the program F, respectively. The condition $\texttt{Init}\,F$ and the parameters A and B denote sets of states and they represent state predicates. The operation $act``A$ denotes the image of the set A under the relation act. A program property is represented as the set of programs satisfying that property. The statement 'program F satisfies property X' is written '$F \in X$'.

Progress properties include transient, ensures and $\mapsto$ ('leads-to'): $F \in \texttt{transient}\,A$ means that some action of F takes A to $\overline{A}$; it falsifies A. The atomic progress property $A\,\texttt{ensures}\,B$ is expressed as the conjunction of $\texttt{transient}\,(A-B)$ with $(A-B)\,\texttt{co}\,(A\cup B)$. The transient part means that we cannot have A without B forever. The co part means that the state remains in A until it enters B. Finally $\mapsto$ is defined to be the transitive and *disjunctive* closure of ensures.[2]

The PSP (progress-safety-progress) *law* we are using

$$\frac{F \in A \mapsto A' \qquad F \in B\,\texttt{co}\,B'}{F \in (A \cap B') \mapsto (A' \cap B) \cup (B' - B)}$$

is proved and is slightly stronger than the usual one.

Composition. In addition of its initial state ($\texttt{Init}\,F$) and actions ($\texttt{Acts}\,F$), a program F also contains an allowed actions part ($\texttt{AllowedActs}\,F$). Two programs F and G are compatible, noted $F\,\texttt{ok}^3 G$, if and only if (by definition):

$$(\texttt{Acts}\,G) \subseteq (\texttt{AllowedActs}\,F) \wedge (\texttt{Acts}\,F) \subseteq (\texttt{AllowedActs}\,G)$$

This notion of *allowed actions* limits the programs with which a component can be composed. It is needed for expressing local variables. Compatibility of a family of components is written $\texttt{OK}_{i\in I}\,F_i$ and means that they are mutually compatible with each other.

Programs are composed by forming the intersection of their initial states and allowed actions and the union of their actions. Composition of two programs is written $F \sqcup G$ and of a family of similar programs is written $\bigsqcup_{i\in I} F_i$.

Existential and universal properties are defined as follows:

$$\texttt{ex\_prop}\,X \equiv \forall F\,G.\, F\,\texttt{ok}\,G \to F \in X \vee G \in X \to F \sqcup G \in X$$

$$\texttt{uv\_prop}\,X \equiv \forall F\,G.\, F\,\texttt{ok}\,G \to F \in X \wedge G \in X \to F \sqcup G \in X$$

For example, initially and transient properties are existential, while co, stable and invariant are universal. In general, $\mapsto$ properties are neither universal nor existential. However, Chandy and Sanders [3] have pointed out that they can be reduced to combinations of existential and universal properties.

Proof method. UNITY is embedded in Isabelle/HOL, from which it inherits the logic as well as a powerful arsenal of tactics. Examples of these tactics are simp_tac which performs simplification by rewriting; blast_tac which proves goals by classical reasoning; and auto_tac which combines the two previous tactics. In addition, there are also two UNITY specialized tactics: constrains_tac and ensures_tac for proving, respectively, safety and progress properties when a program is specified. In this work, these tactics are only used to prove properties of components from their definitions. A property of a composed program is deduced from the universal and existential characteristics of that property and other natural deduction rules.

3. The Toy Example

Consider an I-indexed family of components sharing a global variable: a counter C. Each component also has a local counter c and performs a certain action a. Components increase their local counters and the global counter by one each time they perform the action a. Clearly, each c_i records the number of actions performed by component i. Using compositional reasoning we want to prove system correctness, that C always equals the sum of c_i:

$$\texttt{invariant}\,C = \Sigma_{i=0}^{i=I-1}\,c_i.$$

We present two mechanizations of this example. One is a set-theoretical translation of the original specification. The other, differing in treatment of variables, may be seen as an abstraction of the former. The following sections discuss these two variants.

[1]Charpentier and Chandy [4] write $A\,\texttt{next}\,B$ instead.

[2]Thus contrary to Charpentier and Chandy [4] our inductive definition of $\mapsto$, from Misra [6], is based on ensures. The two definitions are equivalent.

[3]ok corresponds to the $*$ and $\surd$ notations in the papers [4] and [5] respectively.

3.1. Component Specification

The formal specification of component i [4] consists of three safety properties:

$$\texttt{init}(c_i = 0 \land C = 0)$$

$$\forall k.\, \texttt{stable}\,(C = c_i + k)$$

$$\text{For all variables } v\text{, other than } c_i \text{ and } C,\ \forall k.\, \texttt{stable}\,(v = k)$$

The first property fixes the initial values of both C and c_i at zero. The second means that component i always increases C and c_i by the same value. The last states that component i changes no variables other than c_i and C.

For comparison, the Isabelle versions of these properties are given below, for any component $\texttt{comp}\,i$:

$$\texttt{comp}\,i \in \texttt{initially}\,\{s \mid s(\texttt{c}\,i) = 0 \land s(\texttt{C}) = 0\} \qquad (1)$$

$$\texttt{comp}\,i \in \texttt{stable}\,\{s \mid s(\texttt{C}) = s(\texttt{c}\,i) + k\} \qquad (2)$$

$$\texttt{comp}\,i \in \texttt{stable}\,\bigcap_v \{s \mid v \neq (\texttt{c}\,i) \land v \neq \texttt{C} \rightarrow s(v) = k\} \qquad (3)$$

Note that in (3) we use intersection instead of a universal quantification over v. The Isabelle versions are implicitly universally quantified over their free variables.

We don't start from the specification above. We first define a component program (see Figure 1) and then, for correctness, we prove the safety properties.

$\texttt{datatype var} \;=\; \texttt{c nat} \mid \texttt{C}$
$\texttt{types state} \;=\; \texttt{var} \Rightarrow \texttt{int}$

`(*The original definition of action` a`*)`
$\texttt{act}\,i \equiv \{(s, s') \mid s' = s(\texttt{c}\,i := s(\texttt{c}\,i) + 1,\ \texttt{C} := s(\texttt{C}) + 1)\}$

`(*Component's initial state*)`
$\texttt{init} \equiv \{s \mid s(\texttt{C}) = 0 \land s(\texttt{c}\,i) = 0\}$

`(*Component's allowed actions*)`
$\texttt{allowed} \equiv \bigcup F \in \texttt{preserves}(\lambda s.s(\texttt{c}\,i)).\,\texttt{Acts}\,F$

$\texttt{comp}\,i \equiv \texttt{mk\_program}(\texttt{init}, \texttt{act}\,i, \texttt{allowed})$

`(*Our alternative definition of action` a`*)`
$\texttt{act}'\,i \equiv \{(s, s') \mid s'(\texttt{c}\,i) = s(\texttt{c}\,i) + 1 \land s'(\texttt{C}) = s(\texttt{C}) + 1)\}$

Figure 1. Component program in Isabelle.

The `datatype` declaration defines the space of variables using disjoint sum so that $\texttt{C} \neq \texttt{c}\,i$ and $\texttt{c}\,i \neq \texttt{c}\,j$ for any $i \neq j$. The `types` declaration introduces component states as total functions from variables to integers. The constant $\texttt{act}\,i$ denotes the action of component i. The relational notation means that the action starts at a state s and ends at state s', which coincides with s everywhere except in $\texttt{c}\,i$ and $\texttt{C}$, where $s'(\texttt{c}\,i) \equiv s(\texttt{c}\,i) + 1$ and $s'(\texttt{C}) \equiv s(\texttt{C}) + 1$. This definition, which utilises the := notation for function updating[4], is suggested by property (3): the action of component i can change no variables other than $\texttt{C}$ and $\texttt{c}\,i$. Finally, the constant `comp` denotes the program. It is defined using the Isabelle/UNITY built-in three-argument function `mk_program`. The first argument is the component's initial state: $s(\texttt{C}) = 0 \land s(\texttt{c}\,i) = 0$. The second is the component's set of actions: `act` and the `skip` command, which is implicitly added by `mk_program`. The third argument is the component's set of allowed actions. It can be interpreted as follows: component i can be composed with any program F whose actions leave the local variable $\texttt{c}\,i$ unchanged; in other words they preserve $\texttt{c}\,i$. The lambda abstraction in `preserves` is needed because Isabelle usually represents variables as functions over states rather than having a type of variables.

[4]$s(x := u, y := v)$ denotes multiple simultaneous updates.

From the component definition, we prove the desired safety properties: (1) is derived directly from the definition while (2) and (3) are proved automatically by one call to `constrains_tac`.

As we can see, the definition of $\texttt{act}'$ says nothing about s' outside $\texttt{C}$ and $\texttt{c}\,i$. Taking $\texttt{act}'$ as the action of a component, properties (1) and (2) still hold. And instead of property (3), which no longer holds, we have the following (general) one, for any program F:

$$F\,\texttt{ok}\,(\texttt{comp}\,i) \rightarrow F \in \texttt{stable}\{s \mid s(\texttt{c}\,i) = k\} \qquad (4)$$

which derives a weak form (but sufficient) of property (3), see the next Section.

The main difference between the two specifications, i.e. (1), (2), and (3) vs. (1), (2), and (4), is that the former restricts the variables that component i can modify to $\texttt{C}$ and $\texttt{c}\,i$ and only them, while the latter allows components to have and modify additional (local or global) variables. Thus the latter can be seen as abstraction of the former.

3.2. System Specification

Another difference between the two program descriptions appears in the system specification. With the former, we first prove that components are compatible thanks to property (3), i.e. for any integer I, $\texttt{OK}_{i<I}\,(\texttt{comp}\,i)$. Then we can express the system's invariant as follows:

$$\left(\bigsqcup_{i<I} \texttt{comp}\,i\right) \in \texttt{invariant}\{s \mid\ s(\texttt{C}) = \texttt{sum}\,\texttt{c}\,I\,s\},$$

where $\texttt{sum}\,\texttt{c}\,I\,s$ defines $\Sigma_{i=0}^{i=N-1}\, s(\texttt{c}\,i)$.

Using the latter specification, the invariant is instead expressed as follows:

$$\texttt{OK}_{i<I}\,(\texttt{comp}\,i) \rightarrow$$
$$\left(\bigsqcup_{i<I} \texttt{comp}\,i\right) \in \texttt{invariant}\{s \mid s(\texttt{C}) = \texttt{sum}\,\texttt{c}\,I\,s\}$$

since components are no longer compatible by definition.

Note that from property (4) we derive the following one

$$\mathtt{OK}_{i<I}\,(\mathtt{comp}\,i) \to \forall i.\, i < I \to$$
$$\mathtt{comp}\,i \in \mathtt{stable}\left(\bigcap_{j<I}\{s \mid j \neq i \to s(\mathtt{c}\,j) = k\}\right)$$

whose conclusion is weaker than property (3).

In both cases, the proof of the invariant is easy. It relies on the following Isabelle/UNITY theorems:

$$\mathtt{Init}\left(\bigsqcup_{i\in I} F_i\right) = \left(\bigcap_{i\in I} \mathtt{Init}\,F_i\right)$$

$$\left(\bigsqcup_{i\in I} F_i \in \mathtt{stable}\,A\right) = (\forall i \in I.\, F_i \in \mathtt{stable}\,A)$$

which mean that `initially` properties are existential and `stable` properties are universal.

We provide these facts as well as technical lemmas about `sum` to `Auto_tac`, which does the rest of work automatically.

4. The Priority System

The priority system consists of a set of conflicting components. Each of them is constantly looking to perform an action that requires it to have priority over all its neighbors. The resulting conflicts are managed in such a way that (a) no conflicting components are given priority at the same time and that (b) each component is given priority in turn. This priority mechanism is modelled by a directed finite graph (the *priority graph*) whose nodes represent the components and whose arrows represent priorities between them. The graph is kept acyclic when altering priorities.

For brevity, we abbreviate the expressions 'highest/lowest priority over all its neighbors' by 'highest/lowest priority'.

We use relations to represent graphs. This is because relations provide a concise way for representing all the concepts we need and Isabelle proves most of the theorems about relations automatically. Another advantage of using relations is that in a finite universe, acyclicity coincides with well-foundedness. As a consequence, one of the general theorems about graphs we need is simply a result of well-founded relations and it already exists in Isabelle.

Charpentier and Chandy [4] assume first an undirected graph P that is given as a function N which associates to each vertex the set of its neighbors. Then they define a priority graph as any directed graph without symmetric arrows covering all edges of P. We instead simply consider a priority graph as any relation r. When we want to ignore orientation we form its symmetric closure, say $(\mathtt{symcl}\,r)$. This is because if r is a priority relation over P, then N, which represents P, is the function that associates to each vertex i the set $((\mathtt{symcl}\,r)``\{i\}) - \{i\}$, where $(\mathtt{symcl}\,r)``\{i\}$ denotes the image of the set $\{\mathtt{i}\}$ under the relation $(\mathtt{symcl}\,r)$ and $-$ denotes set difference. Thus we can use this definition instead of assuming P. However we have to be sure that the changes made on r will never modify the undirected graph N (see property (10), page 6).

As with the toy example, the component's specification includes a rule constraining the visibility of variables. Note that here variables are functions over states. Precisely, they are boolean functions: variable $(\mathtt{arrow}\,i\,j) = \lambda r.\,(i,j) \in r$ represents priority between components i and j: $(\mathtt{arrow}\,i\,j)$ corresponds to the $i \to j$ notation in the Charpentier and Chandy's paper and means that i has priority over j. Normally a UNITY variable is either local to one program or is global. In this example, we have the rare case where variables are shared between some programs: $\mathtt{arrow}\,i\,j$ can be modified by both i and j and by no other component. We interpret this situation as follows: $(\mathtt{arrow}\,i\,j)$ is global in both i and j, but local in their composition. We define a new meta constructor that takes a program and then imposes a more restrictive set of allowed actions upon it. This constructor, called `localize`, which takes as first argument a variable v, satisfies the following three equalities:

$$\mathtt{Init}\,(\mathtt{localize}\,v\,F) = \mathtt{Init}\,F$$

$$\mathtt{Acts}\,(\mathtt{localize}\,v\,F) = \mathtt{Acts}\,F$$

$$\mathtt{AllowedActs}\,(\mathtt{localize}\,v\,F) =$$
$$(\mathtt{AllowedActs}\,F) \cup \left(\bigcup_{G\in\mathtt{preserves}(v)} (\mathtt{Acts}\,G)\right)$$

This operation gives the ability to form $(\mathtt{comp}\,i)\bigsqcup(\mathtt{comp}\,j)$ and to make $(\mathtt{arrow}\,i\,j)$ local to that composition.

Finally we divide the specification into two theories (modules), separating between what are general facts about graphs (auxiliary) and what are component specifications.

4.1. General Properties of Graphs

Figure 2 presents the theory. The declaration `types` introduces the type `vertex` of vertices which we then assert (in `rules`) to be finite. `UNIV` is a polymorphic constant denoting the universal set in Isabelle and here it is constrained to represent that of vertices. The `constdefs` part contains the needed definitions. The definition of the symmetric closure of a relation is standard. The figure omits the types of constants. Note that a relation r implicitly represents the priority graph with vertex set `UNIV` and edge set r. The neighbors function is defined as previously commented. Its correctness properties are, for any vertices i and j:

$$i \notin (\mathtt{N}\,i\,r) \text{ and } (i \in (\mathtt{N}\,j\,r)) = (j \in (\mathtt{N}\,i\,r)).$$

Like many others simple properties of this section, they are trivially derivable from the definition. Constant `R` defines

```
types vertex

rules (* We assume that ...*)
(*...the universe of vertices is finite*)
finite_vertex_univ finite(UNIV :: (vertex)set)

constdefs (* Constant definitions *)
(* symmetric closure and neighbors *)
 symcl r ≡ r ∪ (r⁻¹)
 N i r    ≡ ((symcl r) “ {i}) − {i}

(* predecessors and successors of i *)
 R i r ≡ r “ {i}
 A i r ≡ r⁻¹ “ {i}

(* reachable and above vertices;
   orginal notations are R* and A* *)
 reach i r ≡ r⁺ “ {i}
 above i r ≡ (r⁺)⁻¹ “ {i}

(* changing priorities of a vertex i *)
 incident i r ≡ r ∩ {(x, y) | x = i ∨ y = i}
 reverse i r ≡ (r − (incident i r)) ∪ (incident i r)⁻¹

(* Derive *)
 derive i r q ≡
    (symcl r) = (symcl q) ∧ (A i r) = ∅ ∧ (R i q) = ∅ ∧
    ∀ k k′. k ≠ i ∧ k′ ≠ i → ((k, k′) ∈ r) = ((k, k′) ∈ q))

 derive′ i r q ≡ (A i r) = ∅ ∧ q = (reverse i r)
```

Figure 2. General properties of graphs.

the set of vertices directly reachable from vertex i (i's successors). Constant A defines the set of vertices from which i is directly reachable (i's predecessors). The sets $(\texttt{reach}\,i\,r)$ and $(\texttt{above}\,i\,r)$, respectively, denote $(\texttt{R}\,i\,r^+)$ and $(\texttt{A}\,i\,r^+)$. They correspond to $\texttt{R}^*$ and $\texttt{A}^*$ in Charpentier and Chandy's paper [4]. We have changed the notation because it can be confused with reflexive transitive closure of a relation.

The following facts are trivially derivable:

$$(i \in (\texttt{reach}\,j\,r)) = (j \in (\texttt{above}\,i\,r))$$

$$\texttt{acyclic}\,r = \forall i.\ i \notin (\texttt{above}\,i\,r) = \forall i.\ i \notin (\texttt{reach}\,i\,r)$$

And the others,

$$((\texttt{above}\,i\,r) = \emptyset) = ((\texttt{A}\,i\,r) = \emptyset) \text{ and}$$

$$((\texttt{reach}\,i\,r) = \emptyset) = ((\texttt{R}\,i\,r) = \emptyset),$$

rely on the following general result of relations $((r^+)\,``\{i\} = \emptyset) = (r\,``\{i\} = \emptyset)$. Note that $(\texttt{A}\,i\,r = \emptyset)$ means that vertex i has the highest priority in r while $(\texttt{R}\,i\,r = \emptyset)$ means that it has the lowest priority.

One of the main lemmas of this section is

$$\texttt{acyclic}\,r \rightarrow ((\texttt{above}\,i\,r) \neq \emptyset \rightarrow \exists j \in (\texttt{above}\,i\,r).\ (\texttt{A}\,j\,r) = \emptyset). \quad (5)$$

It states that every above set has a maximal vertex and it corresponds to the well-known theorem of well-foundedness. Because the universe of vertices is finite, so is relation r and hence $\texttt{acyclic}\,r = \texttt{wf}\,r$.

Constant reverse defines the operation which inverts the orientation of all arrows incident on i: outgoing arrows become incoming and incoming arrows become outgoing.

The definition of derive is taken from Charpentier and Chandy [4]. It relates two relations r and q, with respect to a vertex i. The relations r and q are equal up to the arrows' orientation, $(\texttt{symcl}\,r) = (\texttt{symcl}\,q)$. All arrows incident on i (if any) are outgoing in r $(\texttt{A}\,i\,r = \emptyset)$ and incoming in q $(\texttt{R}\,i\,q = \emptyset)$. Everywhere else, relations r and q are the same. Our alternative definition is derive′. The equivalence of the two is proved automatically by Auto_tac.

The other main lemma of this section is

$$\texttt{derive}\,k\,r\,q \rightarrow \forall i.\ (\texttt{reach}\,i\,q) \subseteq (\texttt{reach}\,i\,r) \cup \{k\}. \quad (6)$$

It states that if q is derived from r, with respect to vertex k, then the reachability of vertices in q is smaller than the union of what is reachable in r and the singleton $\{k\}$. The proof is done by induction on the transitive closure of r; recall that $(\texttt{reach}\,i\,r) = r^+\,``\{i\}$ and so the induction is essentially on the length of the path through the graph.

4.2. Component Specification

Figure 3 presents the component's specification in Isabelle. The component's states are declared to be relations over vertices. The unspecified constant init represents a component's initial state. Constants highest and lowest define priorities. Constant act defines the unique action of a component. Possession of the highest priority is its precondition. The reverse operation guarantees lowest priority as post-condition. The constant comp i denotes the program with its initial condition init, actions act and unconstrained allowed actions UNIV. Recall that mk_program implicitly adds a skip command.

```
types state = (vertex × vertex) set

consts init :: state

constdefs
  highest i r ≡ (A i r) = ∅

  lowest i r  ≡ (R i r) = ∅

  act i  ≡ {(s, s′) | s′ = (reverse i s) ∧ (highest i s)}

  comp i  ≡ mk_program({init}, {act i}, UNIV)
```

Figure 3. Component Program in Isabelle.

From Charpentier and Chandy's paper [4], a component should (a) wait until it has the highest priority over all its neighbors; (b) not introduce cycles in the graph, by having the lowest priority after eventually performing the action; (c) yield priority to its neighbors in finite time.

The first and second items are safety properties. Our specification is a set-theoretical translation of the original ones:

$$\texttt{comp}\, i \in \{s \mid (\texttt{arrow}\, i\, j)(s) = b\} - \{s \mid \texttt{highest}\, i\, s\}\ \texttt{co}\ \{s \mid (\texttt{arrow}\, i\, j)(s) = b\} \quad (7)$$

$$\texttt{comp}\, i \in \{s \mid \texttt{highest}\, i\, s\}\ \texttt{co}\ \{s \mid \texttt{highest}\, i\, s\} \cup \{s \mid \texttt{lowest}\, i\, s\} \quad (8)$$

Both are proved by `constrains_tac`.

The last item is expressed as a progress property. According to Charpentier and Chandy, it would be

$$\texttt{transient}\{s \mid \texttt{highest}\, i\, s\},$$

which means that the condition $\texttt{highest}\, i\, s$ will eventually be falsified by the action `act`. The axiom implicitly assumes a connected graph s, since otherwise it would fail: consider the case where i is an isolated vertex (has no neighbors). We propose instead the following weaker specification[5]:

$$\texttt{comp}\, i \in \texttt{transient}\ (\{s \mid \texttt{highest}\, i\, s\} - \{s \mid \texttt{lowest}\, i\, s\})\,. \quad (9)$$

Note that when vertex i is isolated then $\texttt{lowest}\, i\, s = \texttt{highest}\, i\, s$, otherwise $\texttt{lowest}\, i\, s \rightarrow \neg\texttt{highest}\, i\, s$ and $\texttt{highest}\, i\, s \rightarrow \neg\texttt{lowest}\, i\, s$. This property is proved by the `ensures_tac` tactic, which prove progress properties from the component definition.

From the paper [4], the locality constraint corresponds to the following safety property, for any i, j, and k:

$$\texttt{comp}\, i \in \{s \mid j \neq i \wedge k \neq i \wedge (\texttt{arrow}\, j\, k)(s) = b\}\ \texttt{co}\ \{s \mid (\texttt{arrow}\, j\, k)(s) = b\}$$

where b is a boolean, and is proved by `constrains_tac`. As in the toy example, our alternative rule for locality is

$$F\ \texttt{ok}\ (\texttt{localize}\, (\texttt{arrow}\, i\, j)\, (\texttt{comp}\, i \sqcup \texttt{comp}\, j)) \rightarrow F \in \texttt{stable}\, \{s \mid (\texttt{arrow}\, i\, j)(s) = b\}$$

for any program F.

In addition we prove that the the undirected graph (function `N`) remains unchanged:

$$\texttt{comp}\, i \in \texttt{stable} \bigcap_j \{s \mid \texttt{N}\, j\, s\} \quad (10)$$

[5]Note the equality $(\{s \mid \texttt{highest}\, i\, s\} - \{s \mid \texttt{lowest}\, i\, s\}) = \{s \mid \texttt{highest}\, i\, s \wedge \neg\texttt{lowest}\, i\, s\}$.

4.3. System Specification

The informal specification of the priority system requires that no conflicting components are given priority at the same time (a safety property) and that each component is given priority in turn (a progress property):

$$\texttt{system} \in \texttt{stable} \bigcap_i \{s \mid (\texttt{highest}\, i\, s) \rightarrow \forall j \in (\texttt{N}\, i\, s)\,.\, \neg\texttt{highest}\, j\, s\} \quad (11)$$

$$\texttt{system} \in \{s \mid \texttt{acyclic}\, s\} \mapsto \{s \mid \texttt{highest}\, i\, s\} \quad (12)$$

Compared with Charpentier and Chandy [4], properties (11) and (12) are expressed slightly differently. The former is originally expressed as invariant. But they implicitly assume an initial state satisfying that property. This can be simplified as `stable`. Charpentier and Chandy do not prove this (universal) safety property and simply say that it is easy. Indeed it is: we prove it by applying `constrains_tac` and then `Auto_tac`.

According to Charpentier and Chandy, the left-hand side of $\mapsto$, in property (12), would be `UNIV` rather than $\{s \mid \texttt{acyclic}\, s\}$, which would be proved to be invariant and introduced later in the proof as hypotheses using substitution axiom. However, the proof of invariance of acyclicity again requires the assumption that initial state is acyclic. Our specification avoids such complications.

This progress property is neither universal nor existential. However we show that it can be derived from combinations of existential and universal properties, using the `ensures` primitive, PSP law, etc. We simplify the proof in the following steps:

Safety. Two main properties are proved here:

First is that the `above` set of any component i that does not have priority does not increase, for any j:

$$\texttt{system} \in \{s \mid \neg\texttt{highest}\, i\, s\} \cap \{s \mid j \notin \texttt{above}\, i\, s\}\ \texttt{co}\ \{s \mid j \notin \texttt{above}\, i\, s\}$$

This property is equivalent to, for any (set of vertices) x:

$$\texttt{system} \in \{s \mid \neg\texttt{highest}\, i\, s\} \cap \{s \mid (\texttt{above}\, i\, s) = x\}\ \texttt{co}\ \{s \mid (\texttt{above}\, i\, s) \subseteq x\} \quad (13)$$

Note that 'above set' represents all those components on which a component depends, directly (neighbors) or indirectly.

Charpentier and Chandy [4] give a detailed proof that relies on a universal (`co`) property and lemma (6). Here the `constrains_tac` tactic proves it directly for all components. We introduce lemma (6) and use the equivalence $(j \in \texttt{reach}\, i\, s) = (i \in \texttt{above}\, j\, s)$.

The second property is stability of acyclicity:

$$\texttt{system} \in \texttt{stable}\ \{s \,|\, \texttt{acyclic}\, s\}\,. \tag{14}$$

It is proved as the disjunction of (13) and the following universal property:

$$\texttt{system} \in \{s \,|\, \texttt{highest}\, i\, s\}\ \texttt{co}\ \{s \,|\, \texttt{highest}\, i\, s\} \cup \{s \,|\, \texttt{lowest}\, i\, s\} \tag{15}$$

The proof uses the following (four) rules for rewriting:

$$(\texttt{acyclic}\, s) = \forall\, i.\, i \notin (\texttt{above}\, i\, s)$$

$$(\texttt{highest}\, i\, s) = ((\texttt{above}\, i\, s) = \emptyset)$$

$$(\texttt{lowest}\, i\, s) = ((\texttt{reach}\, i\, s) = \emptyset)$$

$$(j \in \texttt{reach}\, i\, s) = (i \in \texttt{above}\, j\, s)$$

Finally, the property

$$\texttt{system} \in \texttt{stable}\ \{s \,|\, \texttt{maximal}\, s\}\,, \tag{16}$$

where `maximal` is an abbreviation for

$$\forall\, i.\, \texttt{above}\, i\, s \neq \emptyset \rightarrow (\exists\, j \in (\texttt{above}\, i\, s).\, (\texttt{highest}\, j\, s))\,,$$

is derived from (14). As discussed in lemma (5), in finite universes acyclicity coincides with well-foundedness (maximality). This is an important property: any non-priority component always has a component above it that has highest priority.

Progress. Any component with priority will eventually escape every 'above set':

$$\texttt{system} \in \{s \,|\, \texttt{highest}\, i\, s\} \mapsto \bigcap_j \{s \,|\, i \notin \texttt{above}\, j\, s\} \tag{17}$$

The original proof is based on the weak axiom $\texttt{transient}\,\{s \,|\, \texttt{highest}\, i\, s\}$ and can not be followed. The new proof relies on the property

$$\texttt{system} \in \{s \,|\, \texttt{highest}\, i\, s\} \mapsto \{s \,|\, \texttt{lowest}\, i\, s\} \tag{18}$$

and on $\{s \,|\, \texttt{lowest}\, i\, s\} \subseteq \bigcap_j \{s \,|\, i \notin \texttt{above}\, j\, s\}$, which means that a lowest i can never be in any 'above set'. To prove property (18) we reduce $A \mapsto B$ to $A\, \texttt{ensures}\, B$. Unfolding the `ensures` definition yields two properties:

$$\texttt{system} \in \texttt{transient}\ \{s \,|\, \texttt{highest}\, i\, s\} - \{s \,|\, \texttt{lowest}\, i\, s\}$$

$$\texttt{system} \in \{s \,|\, \texttt{highest}\, i\, s\} - \{s \,|\, \texttt{lowest}\, i\, s\}\ \texttt{co}\ \{s \,|\, \texttt{highest}\, i\, s\} \cup \{s \,|\, \texttt{lowest}\, i\, s\}$$

The first property is existential and its proof relies on (9). The second, a universal property, is obtained from (15) by strengthening the left-hand side.

PSP. The fundamental property here is that any 'above set' will eventually decrease and is proved from (13), (16) and (17) using the PSP law, for any i:

$$x \neq \emptyset \rightarrow \texttt{system} \in \{s \,|\, (\texttt{above}\, i\, s) = x\} \cap \{s \,|\, \texttt{acyclic}\, s\} \mapsto \{s \,|\, (\texttt{above}\, i\, s) \subset x\}\,.$$

Here $\subset$ denotes *proper subset* relation. The proof is not difficult: from safety properties (13) and (16) we have that the set $\texttt{above}\, i\, s$ doesn't increase and that there is always a component j in it that has highest priority. Progress property (17) tells us that j will eventually be removed from $(\texttt{above}\, i\, s)$. However the proof includes annoying rewriting steps.

For example, the previous property becomes

$$x \neq \emptyset \rightarrow \texttt{system} \in \{s \,|\, (\texttt{above}\, i\, s) = x\} \cap \left(\bigcup_j \{s \,|\, j \in (\texttt{above}\, i\, s)\} \cap \{s \,|\, \texttt{highest}\, j\, s\} \right) \mapsto \{s \,|\, (\texttt{above}\, i\, s) \subset x\}$$

after simplification, since acyclicity coincides with maximality, and after expressing maximality using set operations.

Charpentier and Chandy mistakenly suggest rewriting property (17) to

$$x \neq \emptyset \rightarrow \texttt{system} \in \{s \,|\, (\texttt{above}\, i\, s) = x\} \cap \{s \,|\, j \in (\texttt{above}\, i\, s)\} \cap \{s \,|\, \texttt{highest}\, j\, s\} \mapsto \{s \,|\, j \notin x\}$$

by strengthening the left-hand side, weakening the right-hand side and introducing x. The problem is that x is constant. And on the left-hand side we have $j \in x$. That means we have $j \in x \mapsto j \notin x$ where both j and x are constant. Our proof corrects this mistake.

Finally, the system progress property (12) is proved as suggested in the paper [4], by induction on the size of the `above` set.

5. Weakest Existential Property

This section describes some theoretical issues relating existential properties and guarantees. By definition $F \in (X\, \texttt{guarantees}\, Y)$ means that for all program G such that $F\, \texttt{ok}\, G$, if $F \sqcup G$ satisfies X then $F \sqcup G$ also satisfies Y. The parameters X and Y are program properties: safety, progress or even *guarantees* properties. Unlike many other rely/guarantee specifications here both X and Y refer to the same system. As a result, *guarantees* properties satisfy many of the rules of implication.

Charpentier and Chandy [5] show that for any program property X one can construct the *weakest existential property* stronger than X[6]. To prove this result they suggest two definitions:

$$\texttt{wx}\, X \;\equiv\; \bigcup\{Y \mid Y \subseteq X \wedge (\texttt{ex\_prop}\, Y)\}$$

$$\texttt{wx}'\, X \;\equiv\; \{F \mid \forall\, G.F\, \texttt{ok}\, G \rightarrow F \sqcup G \in X\}.$$

Here wx corresponds to $\mathcal{E}$ in Charpentier and Chandy's paper. The definition of $\texttt{wx}'$ is slightly different, since here composition ($\sqcup$) is commutative.

Then, in 10 propositions and 5 pages of proofs, they proceed by showing that $\texttt{wx}\, X$ is the weakest existential property stronger than X, and that $\texttt{wx}'\, X$ is also the weakest existential property stronger than X. And from the uniqueness of such a weakest property, they conclude that $\texttt{wx} = \texttt{wx}'$.

We mechanize this proof in few lines, since we directly prove that $\texttt{wx} = \texttt{wx}'$. The wx correctness properties, for any X,

$$(\texttt{wx}\, X) \subseteq X \;\wedge\; \texttt{ex\_prop}\,(\texttt{wx}\, X) \wedge \\ \forall\, Z.\,(\texttt{ex\_prop}\, Z) \wedge Z \subseteq X \rightarrow Z \subseteq (\texttt{wx}\, X),$$

are proved by one call to Auto_tac each, while the equality $\texttt{wx} = \texttt{wx}'$ is shown in an 11-step proof.

Charpentier and Chandy's main result [5] states *guarantees* in term of wx: $(X\, \texttt{guarantees}\, Y) = \texttt{wx}\,(X \rightarrow Y)$. The 1-step proof uses the equality $\texttt{wx} = \texttt{wx}'$ and the definition of guarantees. Finally, as corollary we have $\texttt{ex\_prop}(X\, \texttt{guarantees}\, Y)$.

6. Conclusion

Our mechanization of the two examples improves the original specification and corrects its mistakes. The treatment of variables illustrated in the toy example leads to more abstract specifications. For example, it allows the condition '$\texttt{locala}, \texttt{b}, \texttt{c}$' to be a refinement of '$\texttt{locala}$'. With weaker specification, several proofs in the priority example have had to be invented.

The higher degree of automation in the proofs of the theoretical results shows that the effort needed for the rigour of mechanized proof is becoming more and more modest. Mechanical tools are proving many technical details automatically that previously required human intervention. We have started to mechanize the notation of *weakest guarantees* by Chandy and Sander [3]. The first undertaken proofs include a good degree of automation.

[6]Charpentier and Chandy [5] give a counterexample showing that there is no unique universal property stronger than X.

Acknowledgements

Thanks to the U.K.'s Engineering and Physical Sciences Research Council, who funded this research, grant GR/M 75440 'Compositional Proofs of Concurrent Programs'.

References

[1] David Aspinall. Proof general: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems 6th International Conference, TACAS 2000*, LNCS 1785. Springer, 2000. On the Internet at http://www.proofgeneral.org/.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

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