

Predicate Transformers for Infinite-State Automata in NuPRL Type Theory

Mark Bickford
Odyssey Research Associates
Ithaca, NY, USA

Jason Hickey*
Cornell University
Ithaca, NY, USA

Abstract

This paper has two goals. The first is to present a formalization in Nuprl type theory of a very general methodology for system description, specification and verification. The method is especially suitable for describing distributed systems, and is based on a modification of the I/O automata of Lynch & Tuttle. By using infinite extendible records as the state spaces of automata we gain a key inheritance property that make modular verification tractible.

The second goal is to show how we can state and prove meta-theorems about the method in Nuprl by a reflection procedure whereby we define syntax and semantics for both system descriptions and specifications within Nuprl type theory. We can then define a syntactic predicate transformation algorithm that generates syntactic verification conditions, and then prove the meta-theorem that shows that the truth of (the meanings of) the verification conditions implies that (the meaning of) the description satisfies (the meaning of) the specification.

1 Introduction

At Cornell, we are currently using the Nuprl system to verify a large software system called *Ensemble*. Ensemble is a highly modular group communication system that eases the task of distributed programming by providing configurable implementations of critical distributed algorithms like failure detection and recovery, while giving precise guarantees for message ordering and delivery. This is ideal from the perspective of reliability: the critical distributed algorithms are cleanly isolated and identified. Formal verification of these protocols applies to all applications that use Ensemble.

The Ensemble architecture is designed around layered protocol stacks, composed from a collection of protocol layers that provide a roughly orthogonal set of primitives. Each layer is about 300 lines of code in Objective Caml [11]. Currently there are about 50 different layers that can be composed to implement thousands of useful protocols. Clearly, any feasible formal framework must capitalize on modularity and re-use.

We have developed a formal framework for specifying and verifying modular systems like Ensemble based on the I/O automata of Lynch and Tuttle [10, 9]. I/O automata provide an ideal specification language in some respects. They can specify modular components that range from the concrete OCaml layer code to abstract distributed services. However, the automata defined by Lynch and Tuttle do not, in general, allow formal theorems to be re-used as protocols are modified. For example, a FIFO protocol may have the informal specification, “all messages are delivered in the order in which they were sent.” When a total ordering layer is added to the protocol, it adds the additional property, “all processes receive messages in the same order.” In the standard formalization of I/O automata, the TOTAL protocol is different from the FIFO protocol, and the FIFO property must be re-proved after the total-ordering property is added.

This paper is about the mathematics of our formal model and meta-theorems about the model and not about its application to Ensemble. In this paper we want to highlight the elegant formalism that is made possible by Nuprl’s expressive type theory. We show how using the dependent function and dependent product types together with the intersection types, we can modify the semantics of I/O automata to gain a key inheritance property necessary for modular verification. We also show how a useful fragment of infinitary logic can be formalized within Nuprl so that

*Support for this research was provided by D ARPA grant F30602-95-1-0047. Contact: Jason Hickey, Upson Hall, Cornell University, Ithaca, NY 14853. Email: jyh@cs.cornell.edu, TEL: 607-255-0408.

we can prove a meta-theorem about a verification-condition generator that produces vc's whose validity establishes that an (possibly infinite) automaton satisfies a given (infinitary) invariant.

The verification-condition generation algorithm is a kind of automatic reasoning. It automates the standard steps of proving an invariant of a labeled transition system. These standard steps consist of induction, case analysis, and substitutions; they reduce a temporal property to a set of (non-temporal) formulas that must be shown to be valid. We could (and did) code these steps into a tactic in NuPrl, but by reflecting the whole enterprise into NuPrl we can prove a meta-theorem that proves the correctness of the algorithm, and then we can instantiate the meta-theorem more efficiently than running the tactic.

2 Transition Systems

A formal model especially useful for describing systems of concurrent processes is a *transition system*. A transition system consists of a *state space* S containing all possible states of the system, a subspace I of S containing the possible *initial states* of the system, and a *transition relation* T containing all pairs (s, s') where s' is a possible next state of state s . In NuPrl we can construct the type of all transition systems as the following dependent product

$$S : \mathbb{U} \times I : (S \rightarrow \mathbb{P}) \times (S \rightarrow S \rightarrow \mathbb{P})$$

A member of this type is a triple $\langle S, I, T \rangle$ where S is a type in *universe* \mathbb{U} , I is a unary relation on S , and T is a binary relation on S . We model relations as functions into \mathbb{P} , the type of *propositions* because since NuPrl is a constructive type theory, the functions into the boolean type \mathbb{B} are only the *decidable* relations. We do not assume that, in general, the initial states and transition relation of a system will be decidable.

When we reason about a highly non-deterministic transition system, it is often useful to partition the possible state transitions and label the partition classes. We can then reason by case analysis on these labels. The labels also provide a way to structure the description of the transition system. This leads to the definition of a *labeled transition system* as a member of the following dependent product type

$$A : \mathbb{U} \times S : \mathbb{U} \times I : (S \rightarrow \mathbb{P}) \times (S \rightarrow A \rightarrow S \rightarrow \mathbb{P})$$

The type A is the type of the labels, which we call *actions*, and the transition relation is now a ternary relation on S, A, S . Any labeled transition system with state space S and actions A can be converted into an “equivalent” (unlabeled) transition system with state space $S \times A$.

A transition system or a labeled transition system can be used as a model for a number of temporal logics. In this paper, we concern ourselves only with the simplest (and most useful) temporal specifications, namely *invariants* or *safety properties*. Suppose that $M = \langle S, I, T \rangle$ is a transition system, and that every state s in S is also a *structure* (in the sense of model theory) for a language L . This means that there is a satisfaction relation “ \models ” that assigns a truth value, $s \models \psi$ (in NuPrl, a proposition), to each pair of a state s in S and sentence ψ in L . We define inductively the *reachable* states of M , s is reachable in M if

$$I\ s \vee (\exists s' : S. \text{reachable}(M; s') \wedge T\ s'\ s)$$

Then we define $M \models \text{always } \psi$ iff

$$\forall s : S. \text{reachable}(M; s) \Rightarrow s \models \psi$$

3 Modular Verification

Rather than work directly with labeled transition systems, we describe systems in a “pseudocode” description language, that more closely resembles a “real” programming language. Since this language is based on the pseudocode

used by Lynch to describe I/O automata, we call these descriptions IO-automata, and they are members of a NuPrl type, Ioa , which we will define shortly. These IO-automata are the syntax of descriptions, and the semantics is defined by a meaning function that assigns a labeled transition system to each IO-automata. For now, let us write $M(A)$ for the labeled transition system that is the meaning of IO-automata A . If we choose the meaning $M(A)$ so that members of its state space $S(A)$ are structures for a specification language $L(A)$, then, for ψ in $L(A)$ we can say that “ $A \models \psi$ ” iff $M(A) \models \mathbf{always} \psi$.

We almost never have a complete description, A , of the entire system, and, even if we had a complete description, proving that “ $A \models \psi$ ” for a large description, A , would be intractable. Thus we must have a modular verification method that lets us decompose the system A into components $B_i, i \in I$, and prove properties ψ_i of the components and then compose these component properties to prove the global property ψ .

We may view an IO-automaton description A as a collection of constraints on the system, and the meaning $M(A)$ as the most general transition system satisfying the constraints. The composition operation on IO-automata is then just the union of the collections of constraints, which we write as $A + B$ for binary unions and as $(\cup i:I.B[i])$ for arbitrary unions. We would like to have the composition rule

$$(A \models \psi) \wedge (B \models \chi) \Rightarrow (A+B \models (\psi \wedge \chi))$$

Here ψ is a sentence in $L(A)$ and χ is a sentence in $L(B)$ and $(\psi \wedge \chi)$ is in $L(A+B)$. Since a state s in $S(A+B)$ must be a structure for $L(A+B)$ and since the natural definition of the satisfaction relation $s \models (\psi \wedge \chi)$ is $(s \models \psi) \wedge (s \models \chi)$, we want s to be a structure for $L(A)$ and for $L(B)$. Thus we want the state space $S(A+B)$ to be a *subtype* of the state spaces $S(A)$ and $S(B)$. In the standard semantics, the states $S(A+B)$ of the composition of A and B would be the pairs in $S(A) \times S(B)$ and this type is not a subtype of $S(A)$ or of $S(B)$.

To get the needed *inheritance* property, $S(A+B) \subseteq S(A)$, we modify the standard semantics of IO-automata. We use the dependent function types and intersection types of NuPrl to construct a family of infinite record types that we use for the state spaces $S(A)$ of our transition systems. These are discussed in the next section.

4 Structures and Records

We use the type Label to name symbols in our automata and their specifications. The signature of a language L can be given by an assignment of types to the symbols and we call this a Decl .

$$\text{Decl} == \text{Label} \rightarrow \mathbb{U}$$

If d is a Decl , then a structure s for a language L of signature d is an assignment of values to the symbols such that the value $s[x]$ assigned to symbol x has the type $d[x]$. Such a structure is exactly a member of the dependent function type

$$\{d\} == \lambda x:\text{Label} \rightarrow d[x]$$

This same type $\{d\}$ can also be viewed as the type of records r such that the component $r.x$ has the type declared by $d[x]$. So, we use the words *record* and *structure* synonymously and we write either $r[x]$ or $r.x$ for the application of the record or structure r to the component name or symbol x .

If we have languages $L(A)$ and $L(B)$ of signatures $d(A)$ and $d(B)$, then s of type $\{d(A)\}$ is a structure for $L(A)$. When we combine automata A and B , we will combine their declarations $d(A)$ and $d(B)$ to get the signature “ $d(A);d(B)$ ” for $L(A+B)$, and we want $\{d(A);d(B)\}$ to be a subtype of $\{d(A)\}$ and $\{d(B)\}$, so that a structure for $L(A+B)$ will still be a structure for $L(A)$. To get this “inheritance property”, we make use of the *intersection types* in the NuPrl type theory to define the combination

$$d1;d2 == \lambda x.d1 \ x \cap d2 \ x$$

and to combine an arbitrary collection of signatures

$$D[i] \text{ for } i \in I == \lambda x.(\cap i:I.D[i] \ x)$$

The subtype property can be stated as a theorem in NuPrl

$$\forall I:U. \forall D:I \rightarrow \text{Decl}. \forall j:I. \forall z:\{D[i] \text{ for } i \in I\}. z \in \{D[j]\}$$

Suppose we want a record or structure with only one declared component x of type t . We need declaration d that maps x to t , but d must be a total function from Label to types so d must map labels other than x to some type. We want d to model our knowledge that x has type t and that we have no knowledge about the types of any other symbols. In NuPRL every term is a member of the type Top , so assigning Top to the unknown symbols is the right way to say that we have no information. So, we define the base case declaration as follows

```
x:t == λa.if a = x then t else Top fi
```

Since the intersection $(\text{Top} \cap t1)$ has the same members as $t1$, records for the combined declaration $\{x:t1; y:t2\}$ will be those for which x has type $t1$ and y has type $t2$.

We can also use a declaration d to form a type that is, in some sense, dual to the record type. We call it a sigma-type, and it is the dependent product

```
(Σd) == l:Label × d[l]
```

A member of Σd is a pair $\langle k, v \rangle$ of a kind and a value where the value v has type $d[k]$. We use the record types for the state space of our transition systems and we use the sigma types for the space of actions in our labeled transition systems.

5 Collections

We make use of the fact that for any type T we have a collection type $\text{Collection}(T)$ that is closed under arbitrary unions

```
(U i:I. C[i])
```

contains the singletons $\langle x \rangle$ and empty collection $\langle \rangle$ and is closed under the operations of “map”, “filter”, and “accumulate”

```
<f[x] | x ∈ c>
<x ∈ c | P[x]>
(U x ∈ c. f[x])
```

6 IOA Descriptions

Before we define the type Ioa of IO-automata descriptions, we show an example

```
fifo ==
  state sent : Mlist
  state received : Mlist
  action SND : M
  action RCV : M
  initially sent = nil
  initially received = nil
  precondition RCV(m) ⇒ rcv_ok[received; sent; m]
  effect SND(m) ⇒ sent := append sent m ∈ Mlist
  effect RCV(m) ⇒ received := append received m ∈ Mlist
  only [SND] can affect sent
  only [RCV] can affect received
```

This IO-automaton description consists of two state variable declarations, two action declarations, two initial state conditions, one action precondition, two effect clauses, and two frame conditions. These are the six kinds of basic syntactic elements in our description language. Each element represents some constraint on the labeled transition system that is described.

We define an IO-automaton description to be a tuple of collections of these six basic elements

```

Ioa ==
  ds:Collection(Dec)
  × da:Collection(Dec)
  × init:Collection(Rel)
  × pre:Collection(Pre)
  × eff:Collection(Effect)
  × Collection(Frame)

```

The composition of automata descriptions is just the union of their syntactic elements. Since collections are closed under arbitrary unions, IO-automata are closed under arbitrary compositions. We can also use the map function on collections to define useful operations, such as renaming, on IO-automata.

A Dec is just a pair of a Label and a SimpleType, where a SimpleType is a binary tree of Label. The labels at the leaves of the tree represent basic types and the nodes of the tree represent function arrows.

```

Dec == lbl:Label × SimpleType
SimpleType == Tree(Label)

```

The initial condition is just a relation in the language $L(A)$ of the automaton. A relation is syntactically a relation name and a list of terms. A relation name is either the name for equality over some type, or some other symbol. A term is a binary tree of term symbols, where the nodes of the tree represent application. There are three kinds of term symbols: function symbols, state symbols, and variables.

```

Rel == name:RelName × Term List
RelName == SimpleType + Label
Term == Tree(TermSymbol)
TermSymbol == Label + Label + Label

```

A precondition associates a relation with an action. The action is enabled only when the given relation is true in the current state.

```

Pre == kind:Label × val:Label × Rel

```

An effect clause asserts that an action of the given kind and value, where the value has the given type, has the effect of a given statement. A statement asserts that the new value of the given state variable is the current value of the given term of the given type.

```

Effect == kind:Label × val:Label × typ:SimpleType × Smt
Smt == lbl:Label × term:Term × SimpleType

```

A frame condition asserts that a given state variable of a given type is left unchanged by all actions whose kinds are not in the given list.

```

Frame == var:Label × typ:SimpleType × Label List

```

This completely defines the abstract syntax of our IO-automaton description language and sketches out its intended meaning. In the next section we will give explicit definitions of some of the meaning functions.

7 IOA semantics

To define the semantics of our IO-automaton descriptions, we define (in the following bulleted paragraphs) a family of meaning functions to give meanings to `SimpleType`, `Term`, `RelName`, `Rel`, etc. and then use these to define the meaning function that assigns a labeled transition system to an IO-automaton description. An automaton description is really a template. It doesn't provide the meanings for the basic type symbols, function symbols, and relation symbols. Those meanings must be provided by an environment which is an extra parameter to the meaning functions.

- The meaning of a simple type. The environment here is a `Decl`, `rho`, that assigns meanings from \mathbb{U} to basic type symbols.

```
[[s]] rho == t_iterate(rho; λx,y.x → y;s)
```

The function `t_iterate(f, g, t)` is a generic iterator on binary trees `t` that applies `f` at a leaf and, at a node, applies `g` to the recursively computed values of the children. Since \mathbb{U} is closed under \rightarrow , the meaning of a simple type will always be a type in \mathbb{U} .

- The meaning of a `Dec` is the base case `Decl`

```
[[<x,s>]] rho == x:([s]] rho)
```

The meaning of a `Collection(Dec)` is the `Decl` formed by intersecting all the meanings of the members of the collection, as defined in section 4.

```
[[ds]] rho == [[d]] rho, for d ∈ ds
```

- To evaluate the meaning of a `Term`, we need three environments, `e`, `s`, and `a`. Environment `e` assigns meanings to the function symbols, environment `s` (which will be a state of the transition system) assigns values to the state symbols, and environment `a` assigns values to the variables.

```
[[t]] e s a == term_iterate(s;e;a;λx,y.x y;t)
```

The function `term_iterate(s, e, a, g, t)` is a generic iterator on terms. The meaning of a term will not be well-formed unless the term type-checks. We can write a type-checking algorithm that computes the possible types

```
term_types(ds;da;de;t)
```

of a term from the term and the declarations of the environments. We can then prove that the meaning is well-formed as long as there are any possible types.

```
∀ds,da:Collection(Dec). ∀de:Sig. ∀rho:Decl. ∀t:Term.
∀s:{{[[ds]] rho}. ∀e:{{[[de]] rho}.1}. ∀a:{{[[da]] rho}.
∀b:SimpleType.
b ∈ term_types(ds;da;de;t) ⇒ [[t]] e s a ∈ [[b]] rho
```

- The meaning of a `RelName` is either the equality relation on the type which is the meaning of `Q`, or else it is provided by the environment `e`.

```
[[rn]] rho e ==
Case(rn)
Case relname_eq(Q) =>
λx,y.x = y ∈ [[Q]] rho
Case relname_other(R) =>
e.R
```

- The meaning function for a relation `r` in `Rel` requires environments `rho`, `e`, `s`, and `a`, where `s` provides the meanings of the state symbols, `a` provides meanings for the variables, and `e` is a pair of `Decl`'s that provide meanings for the function symbols and for the relation names.

```
[[r]] rho e s a ==
  list_accum(x,t.x [[t]] e.1 s a;[[r.name]] rho e.2 ;r.args)
```

The iterator `list_accum(f,b,l)` accumulates the value $f \dots f(f b l_1)l_2 \dots l_n$ on the list l_1, l_2, \dots, l_n . So we form the meaning of the relation by starting with the meaning of the relation name and then successively applying it to the meanings of its arguments.

We define a predicate to be a collection of relations. The meaning of a predicate is the conjunction of the meanings of the constituent relations.

```
[[p]] rho e s a ==
   $\forall r:Rel. r \in p \Rightarrow [[r]] rho e s a$ 
```

Notice that since collections can be infinite, our specification language is an infinitary logic (but we have only infinite conjunction, not disjunction).

We can now define the meaning of an IO-automaton A in $\mathbb{I}oa$. We need only the environments ρ and e to assign meanings to the type symbols and the function and relation symbols. As before, we can define a type-check condition and show that when it holds, the meaning of the IO-automaton is a well-formed labeled transition system. In this transition system, the action space is the sigma-type for the action declarations, and the state space is the record type for the state declarations. The initial state space of the transition system is the meaning of the predicate formed from all the initial state relations. It is evaluated with $\lambda x. Void$ as the variable environment because the initial state relations should not contain free variables.

```
[[A]] rho e ==
  < $\Sigma$ ([[A.da]] rho), {[[A.ds]] rho}>,

   $\lambda s. [[A.init]] rho e s (\lambda x. Void)$ ,

   $\lambda s1, a, s2.$ 
    ( $\forall p:Pre$ 
       $p \in A.pre$ 
       $\Rightarrow p.kind = kind(a)$ 
       $\Rightarrow [[p.rel]] rho e s1 (\lambda x. value(a))$ )
   $\wedge (\forall ef:Effect$ 
       $ef \in A.eff$ 
       $\Rightarrow ef.kind = kind(a)$ 
       $\Rightarrow s2.ef.smt.lbl =$ 
         $[[ef.smt.term]] e.1 s1 (\lambda x. value(a))$ )
   $\wedge (\forall fr:Frame$ 
       $fr \in A.frame$ 
       $\Rightarrow \neg(kind(a) \in fr.acts)$ 
       $\Rightarrow s2.fr.var = s1.fr.var) >$ 
```

The transition relation of the labeled transition system is a relation on $s1, a, s2$. Its definition has three conjuncts that incorporate the constraints of the preconditions, the effect clauses, and the frame conditions. All the preconditions for the $kind(a)$ must hold in state $s1$ when the variable environment is $(\lambda x. value(a))$. This is the right variable environment because the preconditions may have only one free variable, the value variable for the action. The effect clauses state that the new value (the value in state $s2$) of the given state variable ($ef.smt.lbl$) is the current value of the given term ($ef.smt.term$), again evaluated with variable environment $(\lambda x. value(a))$. Finally the frame conditions say that if the action kind is not in the given list then the given state variable remains unchanged.

8 Predicate Transformation Algorithm

We want to define a verification condition generator that works with the syntactic descriptions of an automaton A and a predicate I and generates conditions that imply that A satisfies invariant I . The outline of the algorithm is clear. There

should be one verification condition that implies that the initial states of A satisfy I , and for each action kind k , there should be a verification condition to show that if I holds in state s_1 and the preconditions for an action of kind k also hold in state s_1 , then I holds in any state s_2 which is related to s_1 by all the effect clauses and frame conditions. To get such a condition we transform I by substituting the effect and frame conditions into I . This gives us an I' such that $s_1 \models I'$ implies $s_2 \models I$.

Since predicates are just collections of relations, they are not closed under the \Rightarrow operation. So we need a new type for the verification conditions. We also need to handle the fact that effect and precondition clauses mention a free variable that stands for the value of the action, and this variable has to be universally quantified in our verification condition. So we make a type with two cases, one for a simple “hypothesis implies conclusion” where both hypothesis and conclusion are predicates and are intended to have no free variables. The second case has an extra label k , that informs us that all free variables that occur should be bound to the same value of the type corresponding to kind k , and then we universally quantify over such values. Thus we define the Vc type and the meaning function for the two cases.

```
Vc == hyp:Pred × Pred + k:Label × hyp:Pred × Pred
[[h ⇒ c]] rho e s a ==
  [[h]] rho e s (λx.Void) ⇒ [[c]] rho e s (λx.Void)
[[k: h ⇒ c]] rho e s a ==
  ∀v:a[k]. [[h]] rho e s (λx.v) ⇒ [[c]] rho e s (λx.v)
```

In the Vc meaning function the environment a is a $Decl$ that maps the action kind to the type of its value. This a will be provided by the meaning of the action declarations in the automaton.

The vc algorithm $ioa_inv_vc(A;I)$ computes a collection of Vc 's, an initial vc and one vc for each declared action. The vc for the action of kind k says that I together with the preconditions for k must imply the I' obtained by transforming I by all the statements for actions of kind k .

```
ioa_inv_vc(A;I) ==
  <(A.init ⇒ I)> + <ioa_trans(A;a.lbl;I) | a ∈ A.da>

ioa_trans(A;k;I) ==
  (k: (I ∧ action_pre(k;A.pre)) ⇒
   smts_eff_pred(action_effect(k;A.eff;A.frame);I))
```

The action precondition for k is just the collection of relations from the preconditions for k . The effects for k is the collection of statements from the effect clauses for kind k , together with “noop” statements for the frame clauses that apply to k .

```
action_pre(k;ps) ==
  <p.rel | p ∈ <p ∈ ps | p.kind = k>>

action_effect(a;es;fs) ==
  <e.smt | e ∈ <e ∈ es | e.kind = k>> +
  <f.var := f.var ∈ f.typ | f ∈ <f ∈ fs | ¬bk ∈ f.acts>>
```

Now we have a collection of statements and we have to use them to transform predicate I . To transform a predicate, we transform each of its member relations. Because there may be more than one effect clause for a given variable, we have to deal with some non-determinism. So each relation transforms into a collection of relations. The transformed predicate is the union of all of these. The effect of a collection of statements on a single state symbol is the collection of right hand sides (terms) of the statements that have that symbol on the left hand side. The effect on a relation r is the set of all possible substitutions of r where each state symbol is replaced by one of the terms in its effect collection.

```
smts_eff_pred(ss;p) == (Ur∈p.smts_eff_rel(ss;r)
smts_eff(ss;x) == <s.term | s ∈ <s ∈ ss | s.lbl = x>>
smts_eff_rel(ss;r) == col_subst(λx.smts_eff(ss;x);r)
```

To form the collection of all substitutions, given the mapping c from state symbols to collections of terms, we map c over the list of state symbols that occur in r ($\text{rel_vars}(r)$) and then form the product of this list of collections. That product is a collection of lists of terms. We zip each of these with the ($\text{rel_vars}(r)$) to get a collection of substitutions, where a substitution is an association list, in this case, a list of (symbol,term) pairs. The substitution functions $\text{term_subst}(as, t)$ and $\text{rel_subst}(as, r)$ are defined in the obvious way by recursion using our term and list iterators.

```
col_subst(c;r) ==
  <rel_subst(as;r) | as ∈
    <zip(rel_vars(r);s) | s ∈
      col_list_prod(map(c;rel_vars(r)))>>
```

9 Correctness of the Algorithm

We prove the correctness of the vc-generation algorithm by proving the following theorem. The first five hypotheses of the theorem are syntactic side conditions which we will discuss further below. The last hypothesis of the theorem asserts that vc's generated by the algorithm of the preceding section are *valid* in the given environment. Validity of the vc's means that their meanings are true in every structure with the declared signature (recall that we also need a variable type environment to universally quantify the free variables). The conclusion of the theorem is that the meaning of the predicate I is an invariant (as defined in section 2) of the meaning of the automaton A .

```
vc_correctness:
  ∀A:Ioa. ∀I:Pred. ∀rho:Decl. ∀de:Sig. ∀e:{{{de}} rho}.
    tc_ioa(A;de)
    ⇒ tc_pred(I;A.ds;A.da;de)
    ⇒ covers_pred(A;I)
    ⇒ closed_pred(I)
    ⇒ single_valued_decls(A.ds)
    ⇒ valid(ioa_inv_vc(A;I),rho,e,A.ds,A.da)
    ⇒ [[A]] rho e |= always s.([[I]] rho e s)

valid(vs,rho,e,ds,da) ==
  ∀s:{{{ds}} rho}. [[vs]] rho e s ([[da]] rho)
```

The “input data” for the theorem are the automaton description A , the predicate I , the type environment ρ , the function and relation symbol environment e , and one extra thing, a signature de which declares the environment e .

A signature is a pair of syntactic associations. The first assigns simple type symbols to the function symbols. The second assigns a list of simple type symbols to the relation names, giving their arity and argument types. The meaning of a signature is a pair of Decl's that map labels to types in \mathbb{U} . The meaning of the list of simple types t_1, \dots, t_n in a relation signature is obtained by evaluating the meanings of the simple types in the list, to get T_1, \dots, T_n and then forming the type $T_1 \rightarrow \dots \rightarrow T_n \rightarrow \mathbb{P}$. Once we have a pair of Decl's we can form the corresponding record-pair type. A member of this record-pair type is a structure for the signature.

```
Sig ==
  fun:(Label → SimpleType) × (Label → SimpleType List)
  {{{de}} rho == <λop.{{{de.fun op}} rho, λR.{{{de.rel R}} rho>
  {{{l}} rho == reduce(λs,m.{{{s}} rho → m;ℙ;1)
  {p} == {p.1} × {p.2}
```

Now we can discuss the side-conditions of the correctness theorem. The first side-condition $\text{tc_ioa}(A;de)$ is the type-check condition on the automaton. It asserts that all the types in the effect clauses and frame conditions match the declarations and that the terms on the right hand sides of the statements have possible types that include

the given types. The second side-condition $\text{tc_pred}(\mathbb{I}; A.\text{ds}; A.\text{da}; \text{de})$ asserts that the predicate \mathbb{I} type-checks in the context of the given signatures. The third side-condition $\text{covers_pred}(A; \mathbb{I})$ asserts that for every state symbol x mentioned in predicate \mathbb{I} , there must be a frame condition of A that lists which actions affect x , and for every action listed in that frame condition, there must be an effect clause of A that covers x . The fourth side-condition $\text{closed_pred}(\mathbb{I})$ asserts that the predicate \mathbb{I} does not contain free variables.

The fifth side-condition $\text{single_valued_decls}(A.\text{ds})$ asserts that every state symbol x declared in A has only one simple type associated with it. In general, our syntax and semantics allow multiple types for a state symbol x . The automaton description A could contain both declarations $x : \tau_1$ and $x : \tau_2$, and the meaning would constrain x to be in the intersection of the meanings of the simple types τ_1 and τ_2 . The only reason the fifth side condition is needed in our current theorem is that our defined type-checking algorithm has no way to know whether this intersection is empty or not. If we added subtype information to our signatures, so that, for instance, we could declare that $\tau_1 \subseteq \tau_2$, then we could relax the single-valued assumption and allow multiple declarations as long as they had a common lower bound.

We conclude this section with a few words about the proof of the correctness theorem. The proof was completed in the `Nuprl` system. It was difficult and took about three weeks to construct. Part of the difficulty lay in the fact that when reasoning about the meanings of relations, we must reason about the meaning of the argument list. Syntactically, this is a list of `Term`, but once we apply the meaning function we have a heterogeneous list. This means that the list as a whole does not have a type, so we can't use it in many of `Nuprl`'s rules. Because of this, our induction arguments had to be constructed with extreme care. We conjecture that this meta-theorem can not even be proved (without a major reformulation) in a logical system that has a decidable typing and requires all terms to be typed.

Another complication in the proof arose from the fact that `Nuprl` normally allows only constructive reasoning. In the inductive proof of the invariant, we have to prove that the effect of every action a preserves the invariant. The `vc`-algorithm generates a `vc` only for the actions declared in the automaton. Because of the ‘‘covers’’ condition, and the type-check condition, any action that is not declared would have no effect on any state variables mentioned in the invariant. So, the classical proof would have a case-split at this point on whether a is or is not declared in the automaton. But the declarations are an arbitrary collection, and membership in an arbitrary collection is not decidable, so this is not a constructive proof step. We were able to complete the proof without having to assume that the declarations form a decidable collection, by delaying the case analysis. Since the invariant is a predicate, and a predicate is a collection of relations r , it is enough to show that each r is preserved by each action. We were able to postpone the case analysis to the point where the case in question was whether r is left unchanged by a certain substitution. Since the relations are completely finite, syntactic terms, this question is decidable.

10 Example VC's

As an example of the result of the `vc`-algorithm, if we take the automaton `fifo` from section 6 and the predicate containing the single relation `iseg[received, sent]` then we get a collection of three `vc`'s:

```
<sent = nil  $\wedge$  received = nil  $\Rightarrow$  iseg[received, sent]> +
<SND: iseg[received, sent]  $\Rightarrow$  iseg[received, append sent m]> +
<RCV: iseg[received, sent]  $\wedge$  rcv_ok[received; sent; m]
 $\Rightarrow$  iseg[append received m, sent]>
```

We define the environment `rho` to map `M` to any type M and to map `Mlist` to M *List* and define the environment `e` to map

```
e nil = nil
e append =  $\lambda x, y. x @ y$ 
rcv_ok =  $\lambda r, s, m. ||r|| < ||s|| \wedge s[||r||] = m$ 
e iseg =  $\lambda x, y. x \leq y$ 
```

where

```
11 ≤ 12 == ∃l:T List. 12 = 11 @ l
```

The three vc's are then easily shown to be valid in the given environment.

11 Infinite State Automata

The `fifo` example has an infinite state space only because its two state variables are unbounded lists. But we did not need an infinitary logic to state the invariant, because there were only two state variables. The `fifo` automaton models a single fifo channel. We make use of the infinitary nature of the automata and their specifications when we replicate the `fifo` automaton to allow an unbounded number of senders and receivers. We can do this using renaming and composition of automata.

The renaming function applies a renaming map `f` of type `Label → Label` to all the state symbols and action kinds in the automaton description.

```
ioa_rename(f;A) ==
  mk_ioa(<dec_rename(f;d) | d ∈ A.ds>,
        <dec_rename(f;d) | d ∈ A.da>,
        <rel_rename(f;r) | r ∈ A.init>,
        <pre_rename(f;p) | p ∈ A.pre>,
        <eff_rename(f;e) | e ∈ A.eff>,
        <frame_rename(f;fr) | fr ∈ A.frame>)
```

The composition operator for an indexed family `A[i]` of automata just forms the union of all the syntactic constraints. We write the composition as an intersection because we are thinking of the meanings of the automata which are being constrained as we compose.

```
(∩i∈I.A[i]) ==
  mk_ioa((∪i:I.A[i].ds),
        (∪i:I.A[i].da),
        (∪i:I.A[i].init),
        (∪i:I.A[i].pre),
        (∪i:I.A[i].eff),
        (∪i:I.A[i].frame))
```

Until now, we have used the type `Label` as a primitive type, but it actually has some structure. It is closed under a pairing function that we write with square brackets. Thus if `a` and `b` are members of `Label` then so is `a[b]`. So a label is a binary tree, and at the leaves we can put strings or integers. We define a useful renaming function `subscript(ls,q)` that maps every label `a` in the list `ls` to the label `a[q]` and leaves other labels unchanged. Using this and the renaming and composition we can define a replication operator

```
ioa_repl(I;f;ls;A) ==
  (∩i∈I.ioa_rename(subscript(ls;f i);A))
```

The replication operator takes an index type `I`, a map `f` from indices to labels, a list `ls` of labels and automaton `A`. For each index `i`, it adds the subscript `f i` to each of the listed state and action names in `A` to get a new `A[i]`, and then composes all of them.

Suppose the `Pid` is a possibly infinite type of process identifiers, and `lp` is a map from `Pid` into `Label`. Then we can form the replicated

```
fifo1 == ioa_repl(Pid, lp, [sent,SND,received,RCV], fifo)
```

This automaton has state variable `sent[p]`, `received[p]` for each `p` in the range of `lp`. We can replicate again

```
fifo2 == ioa_repl(Pid, lp, [received,RCV], fifo1)
```

Now `fifo2` has state variables `sent[p]`, `recieved[p][q]` and actions `SND[p](m)`, `RVC[p][q](m)` for each `p` and `q`. The reader can check that it makes sense to view the action `SND[p](m)` as a broadcast of message `m` by process `p` and `RVC[p][q](m)` as the receipt of message `m` from `p` to `q`.

We can show, as an instance of a general theorem, that if `fifo` \models `iseg[received, sent]` then

$$\text{fifo2} \models (\forall p:\text{Pid}.\ (\forall q:\text{Pid}.\ \text{iseg}[\text{received}[p][q], \text{sent}[p]])$$

Here the “universal quantification” is really an infinitary conjunction because our language has the infinitely many symbols `sent[p]`, `recieved[p][q]`. In fact, our definition of the “forall” on predicates is just the union

$$(\forall i:\mathbb{I}.\ P[i]) == (\cup i:\mathbb{I}.\ P[i])$$

We could now compose (intersect) the replicated `fifo2` with another automaton that imposes further constraints. If we want to establish an invariant `Q` of the composed system and `Q` is not an invariant of any of the components then we can use our vc-generation algorithm. Since there are infinitely many actions in the replicated system, there will be infinitely many vc’s. They will be parameterized in the same way that the replication was parameterized so we can prove that they are valid with one parameterized proof.

12 Related Work

Abadi and Lamport [1] have explored composition in TLA (Temporal Logic of Actions) using *assume-guarantee* specifications. We describe our systems with automata, rather than a temporal *logic* like TLA, because we can use a single language to represent both programs and specifications. However, our formalism shares the same problem space with TLA. In fact, our safety theorem ($A \models \text{always } P \Rightarrow (A \cap B \models \text{always } P)$) is currently too weak: while the intersection $A \cap B$ has the safety properties of both A and B , additional safety properties may hold because of interference between the two automata. It is likely that we can use the assume-guarantee style of reasoning to extend our framework with a more complete result.

Although our approaches differ at a high level—TLA is a temporal *logic* and we deal directly with automata, both approaches describe the same problem space.

Automata are used for specifying distributed systems in [6, 3]. In [7], protocol layers for point-to-point messaging are formally specified and composed using TLA [8].

I/O automata are defined by Lynch and Tuttle [9], and are discussed further in Lynch [10]. The NuPrl proof development system includes both a logic and a mechanism for reasoning. An early version of the system is described in Constable et. al. [2]; more recent descriptions can be found in Jackson’s thesis [5]. Records are a central part of our formalization; a more complete description of the type theory of records can be found in Hickey [4].

References

- [1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Toplas*, 17(3), May 1995.
- [2] R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice–Hall, 1986.
- [3] Alan Fekete. Formal models of communications services: A case study. *IEEE Computer*, 26(8):37–47, August 1993.
- [4] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College.
- [5] Paul Bernard Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, January 1995.
- [6] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. on Prog. Lang. and Systems*, 16(2):259–303, March 1994.

- [7] David A. Karr. *Protocol Composition on Horus*. PhD thesis, Dept. of Computer Science, Cornell University, December 1996.
- [8] Leslie Lamport. Introduction to TLA. Technical Report 1994-001, DIGITAL SRC, Palo Alto, CA, 1994.
- [9] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands*, 2(3):219–246, September 1989. Also Tech. Memo MIT/LCS/TM-373.
- [10] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [11] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.