Formal Verification of Pure Production System Programs

Rose F. Gamble and **Gruia-Catalin Roman** and **William E. Ball**[*]
Washington University
Department of Computer Science
St. Louis, MO 63130
rfg@cs.wustl.edu, roman@cs.wustl.edu, web@cs.wustl.edu

Abstract

Reliability, defined as the guarantee that a program satisfies its specifications, is an important aspect of many applications for which rule-based programs are suited. Executing rule-based programs on a series of test cases does not guarantee correct behavior in all possible test cases. To show a program is reliable, it is desirable to construct formal specifications for the program and to prove that it obeys those specifications. This paper presents an assertional approach to the verification of a class of rule-based programs characterized by the absence of conflict resolution. The proof logic needed for verification is already in use by researchers in concurrent programming. The approach involves expressing the program in a language called Swarm, and its specifications as assertions over the Swarm program. Among models that employ rule-based notation, Swarm is the first to have an axiomatic proof logic. A brief review of Swarm and its proof logic is given, along with an illustration of the formal verification method used on a simple rule-based program.

Introduction

Rule-based (RB) programs have been very successful in applications where speed and total reliability are not significant factors. However, speed and reliability are important in applications involving critical real-time decisions. The issue of execution speed is currently being addressed by work on parallel production systems (Ishida 1990; Schmolze and Goel, 1990) which is concerned with parallel implementations of existing sequential rule-based programs. Reliability, defined as the guarantee that a program satisfies its specifications, has not been addressed as yet. For this purpose, a set of formal specifications must be given for the program and the program must be expressed in a language with an associated proof theory to show that the program obeys its specifications. Such a proof theory is not associated with current languages used to express RB programs, such as OPS5 (Forgy, 1981).

The principal contributions of this paper are to show that many rule-based programs may be formally verified using assertional methods and that the proof logic needed for verification is already in use by researchers in concurrent programming. This proof logic is applicable to both sequential rule-based programs and to concurrent rule-based programs, i.e., to programs that exhibit logical parallelism independent of their implementation.

The proof logic we will be using was originally developed by Chandy and Misra for UNITY (Chandy and Misra, 1988), a concurrency model based on conditional multiple assignment statements to shared variables. This proof logic was later generalized by Cunningham and Roman (Cunningham and Roman, 1990) for use with Swarm, a concurrency model based on atomic transactions over a set of tuple-like entities. Because Swarm uses tuples to represent the entire program state and builds transaction definitions around a rule-based notation, certain programs written in traditional rule-based programming languages have *direct* correspondents in Swarm and may be subjected to formal verification. Because of space limitations, we assume familiarity with the components of RB programs. A complete discussion can be found in (Winston, 1984). One class of RB programs that can be translated to Swarm without modification is the class of *pure production system programs.* In such programs no conflict resolution strategy is used. Instead, instantiations are chosen nondeterministically for execution from the conflict set. Many RB programs that depend on some form of conflict resolution can be reformulated as pure production system programs. The conversion of such programs is not addressed in this paper.

We begin the paper by presenting Swarm and its proof logic. In the next section, we demonstrate the proof theory on an example RB program. The final section gives a brief discussion and conclusion.

[*]The first and third authors were supported by Washington University's Center for Intelligent Computer Systems, whose primary sponsors are McDonnell Douglas Corporation and Southwestern Bell Corporation.

Swarm

Swarm (Roman and Cunningham, 1990) is a *shared dataspace* model and language, in which the principal means of communication is a common content-addressable data structure, called the dataspace. Swarm provides a small number of constructs that are at the core of a large class of shared dataspace languages, of which RB programs are a part.

Working memory directly corresponds to the Swarm *tuple space*, which is one part of the dataspace consisting of a set of data tuples. Each working memory element in a RB program is represented as a *tuple* in the tuple space[1]. Production memory maps to the Swarm *transaction space*, which is another part of the dataspace[2] consisting of a set of transactions that indicate possible actions to be taken by the program. Each transaction may be viewed as a parameterized rule. Simple transaction definitions involve a left-hand side (LHS) and a right-hand side (RHS) with the same meanings as in a rule in a RB program. Complex transaction definitions use a ||-operator to combine simple definitions of rules, also called *subtransactions*, into a single transaction. Note that Swarm makes a distinction between the definition of a transaction and its existence. Only transactions that exist in the transaction space may be executed.

The execution cycle of a Swarm program begins by choosing a transaction nondeterministically from the transaction space. The choice is fair in the sense that a transaction in the transaction space will eventually be chosen. As a by-product of its execution, the transaction is deleted from the transaction space, unless it explicitly reasserts itself. Once chosen, the LHS of all subtransactions are matched simultaneously. Those subtransactions whose LHSs are satisfied, execute their RHSs simultaneously, performing all deletions before additions. Only tuples may be deleted in the RHS of a subtransaction, but both tuples and transaction may be asserted. Termination occurs when no transactions are left in the transaction space. Figure 1 presents the tuple and transaction notation of Swarm.

Each rule in a pure production system program is represented as a subtransaction of a distinct transaction. In addition, the termination conditions of the pure production system program are defined, negated, and placed as a second subtransaction in each transaction. This ensures that the transaction is reasserted into the transaction space as long as the termination conditions are not satisfied. Thus, each transaction in Swarm contains two subtransactions: (1) the direct translation of a single rule in the RB program and (2) the negated termination conditions of the RB program for reassertion of the transaction. Since a transaction is chosen nondeterministically, and has an effect only if its LHS is satisfied, the execution sequences produced are those of a pure production system program.

[1]If working memory is a multiset, it must be encoded as a set.

[2]There is a third and final part of the dataspace, the *synchrony relation*, but we do not elaborate on it in this paper.

$$
\begin{aligned}
&T(i) \equiv \\
&\qquad X,Y : out(X) \land in(Y) \longrightarrow in(Y)\dagger, out(Y) \\
&\qquad \| \\
&\qquad Z : in(Z) \longrightarrow T(i);
\end{aligned}
$$

Description: (a) T(i) is the transaction name, with variable i.. (b) X, Y, Z are dummy variables. (c) out(X) is the 1st tuple in the LHS of the first subtransaction in T(i), where "out" is the class name and X represents an attribute value of that class. (d) The arrow ($\longrightarrow$) separates the LHS and RHS. (e) In the RHS, the dagger (†) means "delete this tuple from the tuple space." No dagger means "add this tuple to the tuple space." (f) The parallel bars (||) separate the subtransactions. (g) in(Z) is the 1st tuple in the second subtransaction, which reasserts the transaction as long as in(Z) is in the tuple space. For notational convenience, the above transaction can be rewritten as:

$$
\begin{aligned}
&T(i) \equiv \\
&\qquad X,Y : out(X), in(Y)\dagger \longrightarrow out(Y) \\
&\qquad \| \\
&\qquad Z : in(Z) \longrightarrow T(i);
\end{aligned}
$$

Figure 1: Swarm Sample Transaction.

Proof System

In this section, we briefly summarize the Swarm proof logic (Cunningham and Roman, 1990). This proof logic is built around assertions that express program-wide properties. Such properties encompass the entire knowledge base and database of a RB program. The Swarm proof logic is based on the UNITY (Chandy and Misra, 1988) proof logic, and uses the same notational conventions. Informally, the meaning of the assertion {p} t {q} for a given Swarm program, is whenever the precondition p is *true* and transaction instance t is in the transaction space, all dataspaces which can result from execution of t satisfy postcondition q.

As in UNITY's proof logic, the basic safety properties of a program are defined in terms of **unless** relations.

$$\frac{[\forall t : t \in TRS :: \{p \land \neg q\}\ t\ \{p \lor q\}]}{p \ \textbf{unless}\ q}$$

where the bar represents inference and TRS is the set of all transactions that can occur in the transaction space. Informally, if p is *true* at some point in the computation and q is not, then, after the next step, either p remains *true* or q becomes *true*. From this definition, the properties **stable** and **invariant** can be defined as follows,

$$\textbf{stable}\ p \equiv p\ \textbf{unless}\ \text{false}$$
$$\textbf{invariant}\ p \equiv (\text{INIT} \Rightarrow p) \land \textbf{stable}\ p$$

where INIT is a predicate which characterizes the valid initial states of the program. Informally, a stable predicate once *true*, remains *true*, and invariants are always *true*. The symbol $\Rightarrow$ represents logical implication.

The **ensures** relation is the basis of the progress properties. This relation is defined as follows,

$$\frac{p \textbf{ unless } q \wedge [\exists t : t \in TRS :: (p \wedge q \Rightarrow [t]) \wedge \{p \wedge \neg q\}\ t\ \{q\}]}{p \textbf{ ensures } q}$$

where [t] means that the transaction t is actually present in the transaction space. Informally, if p is *true* at some point, then (1) p will remain *true* as long as q is *false*, and (2) if q is *false*, there is at least one transaction in the transaction space which can establish q as *true*.

For the **leads-to** ($\longmapsto$) property, the assertion p $\longmapsto$ q is *true* if and only if it can be derived by a finite number of applications of the following inference rules.

(1) $\dfrac{p \textbf{ ensures } q}{p \longmapsto q}$

(2) $\dfrac{p \longmapsto r \wedge r \longmapsto q}{p \longmapsto q}$

(3) For any set W, $\dfrac{[\forall\ m : m \in W :: p(m) \longmapsto q]}{[\exists\ m : m \in W :: p(m)] \longmapsto q}$

Informally, p $\longmapsto$ q means once p becomes *true*, q will eventually become *true*, but p is not guaranteed to remain *true* until q becomes *true*.

Illustrating a Correctness Proof

We use the Bagger problem (Winston, 1984) to illustrate the use of Swarm proof logic for verifying RB programs. Bagger is a toy expert system to bag groceries according to their container types and weights. This program was chosen because: (1) it can be fully specified formally, (2) it can be stated as a pure production system program, and (3) it exhibits some basic properties of a RB program, such as tasking and context switching. For notational convenience, we have eliminated some extraneous information in the original program.

Bagger is given a set of unbagged grocery items represented by tuples of the type **unbagged(I)**, where I, ranging from 1 to *maxitems*, denotes a *unique item number*. The value of *maxitems* is determined by the number of unbagged items given initially. For each unbagged tuple in the tuple space, the program is given a description of that item in the form of a tuple of type **grocery(I,B,W,F)**. The first field of this tuple type corresponds to the *unique item number*. The next field corresponds to a boolean value representing *whether or not the item is a bottle*. The third field gives *one of three possible weights that determines if the item is small, medium or large*. These weights are: **smwgt, medwgt, lgwgt** respectively. The last field corresponds to a boolean value representing *whether or not the item is frozen*.

Execution of the program must place unbagged items in a bag, in a predefined order. Bags should only be created when needed. To represent a bag, a tuple of type

1. There is exactly one step tuple present at all times.
2. The total number of grocery items equals maxitems.
3. For every item, there is exactly one grocery item.
4. At any time, a grocery item is either inside or outside of a bag.
5. A bagged item exists once in only one bag.
6. A bagged item remains bagged in the same bag and in the same position.
7. The items in each bag are ordered as follows:
   (a) large bottles (b) large items
   (c) medium frozen items (d) medium non-frozen items
   (e) small frozen items (f) small non-frozen items
8. At any time, the weight of every bag cannot exceed the maximum weight allowed.
9. The bags are ordered sequentially, beginning with the number 1.
10. The bags are identified by unique natural numbers.
11. At any time, the first item in bag N, cannot fit in any bag M, where M < N.
12. All unbagged items are eventually bagged
13. Eventually every bag has at least one item.
14. Once all items are bagged, all remain bagged.
15. Once all items are bagged, the program terminates.

Figure 2: Informal Specifications of Bagger.

bag(N,W,A) is placed in the tuple space, in which N is the *bag's unique identification number*, W is the *total weight of the bag*, and A is a *sequence containing the identification numbers of the items placed in the bag so far*. A bag can only reach a certain weight, called **maxwgt**. Since bags are created dynamically, a tuple of type **current(N)** keeps track of the number of bags created.

Another tuple of type **step(B)**, is used as a context element to divide the rules into tasks 1, 2, 3, and 4, depending on the value of B. The context element in Bagger is a single working memory element that is present in the LHS of every rule making each rule contribute to some task. It is also always present in working memory. In each task, a control rule is used to switch contexts, according to the predefined task ordering. The tasks are: (1) bag large bottles, (2) bag large items, (3) bag medium items, and (4) bag small items. Bagger terminates when all unbagged items are bagged. Figure 2 informally details the full specifications of Bagger.

Translation of Bagger to Swarm

Each rule in the original Bagger was translated to a Swarm transaction as discussed in the earlier Swarm section. Figure 3 shows the Swarm transactions for task 1, bag large bottles. The symbol • represents concatenation and <> represents the null sequence. Figure 4 shows the English translation of these same transactions. We will concentrate on this task for the remainder of the paper. The formal specifications, the Swarm program in its entirety, and the proof of Bagger can be

Swarm

```
Rule(1) ≡
 I,N,W,A :
    step(1), large-bottle(I), unbagged(I)†,
    bag(N,W,A)†, W ≤ maxwgt - lgwgt
    ⟶
    bag(N, W+lgwgt, A • I)
‖
 I :unbagged(I) ⟶ Rule(1);

Rule(2) ≡
 I,N :
    step(1), large-bottle(I), unbagged(I), current(N)†,
    [∀ M,W,A : bag(M,W,A) :: W > maxwgt - lgwgt]
    ⟶
    bag(N+1, 0, <>), current(N+1)
‖
 I :unbagged(I) ⟶ Rule(2);

Rule(3) ≡
    step(1)†, [∀I : large-bottle(I) :: ¬unbagged(I)],
    ⟶
    step(2)
‖
 I :unbagged(I) ⟶ Rule(3);
```

Figure 3: Transactions to bag large bottles in Bagger.

English

When Rule(1) is chosen, if the current step is 1 and there exists a large unbagged bottle, and an available bag, then delete the unbagged tuple and add the item and its weight to the bag. Any unbagged item in the tuple space causes the transaction to be reasserted.

For Rule(2), if the step is 1 and there exists a large unbagged bottle and the current number of bags is N, and no bag can hold the item, then create a new bag, changing the current number of bags.

For Rule(3), if the step is 1 and all large bottles are bagged, then delete the current step tuple and insert the tuple step(2), which enables large items to be bagged.

Figure 4: English Version of Swarm Transactions to bag large bottles.

found in (Gamble *et al.,* 1991).

Sample Proof

In this section, we prove a single progress property to demonstrate how the proof logic of Swarm can be applied to a rule-based program. A progress property is normally expressed as a **leads-to** relation between two predicates.

The proof presented in this section will show that task 1, bag large bottles, fulfills its objective. Every task is characterized formally by its initial and termination conditions. The termination conditions of a task must eventually be reached from the initial conditions. When the termination conditions are reached, the objectives of the task should be met. Let *init(1)* represent the initial conditions of task 1, and *term(1)* its termination conditions. Task 1 is activated only when the tuple step(1) is in the tuple space. Also part of *init(1)* is that all large bottles are unbagged. The termination condition of task 1 is that all large bottles are bagged. Since this condition must occur when step(1) is in the tuple space, it is also part of *term(1)*. The proof obligation for task 1 is stated as follows.

Prove: While in task 1, all large bottles are eventually bagged.

Formally, this is stated:

$$(1) \qquad init(1) \longmapsto term(1).$$

which expands to:

$$(2) \quad \text{step}(1) \wedge [\forall\ I : \text{large-bottle}(I) :: \text{unbagged}(I)] \\ \longmapsto \\ \text{step}(1) \wedge [\forall\ I : \text{large-bottle}(I) :: \text{bagged}(I)]$$

where large-bottle(I) is **true** if there is a grocery tuple representing a large bottle with I as the unique identifier. Hence, large-bottle(I) is defined by:

$$\text{large-bottle}(I) \equiv \\ \text{item}(I) \wedge \\ [\exists\ W,F : \text{grocery}(I,\textit{true},W,F) :: W = \text{lgwgt}];$$

item(I) is defined by:

$$\text{item}(I) \equiv [1 \leq I \leq \text{maxitems}];$$

and bagged(I) is *true* if the item is in some bag:

$$[\forall\ I : \text{item}(I) :: \text{bagged}(I) \equiv \\ [\exists\ N,W,A,n : \text{bag}(N,W,A) :: A.n = I]\ ]$$

Before we prove (1), we must first show that the initial conditions of the task are actually established sometime during execution. In the case of task 1, the predicate *init(1)* should be implied by the initial conditions of the program, represented by the predicate INIT. We will assume INIT satisfies the following properties stated informally below.

A. The total number of grocery items equals *maxitems*.
B. There exists one grocery tuple for each item.
C. For every grocery tuple (i.e., grocery(I,B,W,F)), there is a corresponding unbagged tuple (i.e., unbagged(I)).

D. There are no tuples of type bag(N,W,A) present.
E. The tuple step(1) is the only one of its type present.
F. All transactions are present in the transaction space.

From the definition of INIT, it should be clear that

(3) $\text{INIT} \Rightarrow \textit{init(1)}$

A property of **ensures** is that $\frac{p \Rightarrow q}{p \textbf{ ensures } q}$ (Chandy and Misra, 1988). Then, by the first inference rule of **leads-to** we have:

(4) $\text{INIT} \longmapsto \textit{init(1)}$

To show (1), we use induction on the number of large-bottles. By property 4, in Figure 2, we know:

$[\forall\ \text{I} : \text{large-bottle(I)} :: \text{bagged(I)}] \Leftrightarrow$
$[\Sigma\ \text{I} : \text{large-bottle(I)} \wedge \text{unbagged(I)} :: 1]^3 = 0.$

Then, it is clear that

(5) $\text{step(1)} \wedge [\Sigma\ \text{I} : \text{large-bottle(I)} \wedge \text{unbagged(I)} :: 1] = 0 \equiv \textit{term(1)}$

Then we need to show *init(1)* $\longmapsto$ (5). If there are no large bottles, then the proof is trivial. Assume that initially the number of large bottles is non-zero. We define

(6) $\text{must-bag}(1,\alpha) \equiv$
$\text{step(1)} \wedge$
$[\Sigma\ \text{I} : \text{large-bottle(I)} \wedge \text{unbagged(I)} :: 1] = \alpha \wedge$
$\alpha \geq 0$

Then

(7) $\textit{init(1)} \Rightarrow (\text{must-bag}(1,\alpha) \wedge \alpha > 0)$

and

(8) $(\text{must-bag}(1,\alpha) \wedge \alpha = 0) \Rightarrow \textit{term(1)}$

are *true*. Both (7) and (8) can be stated as **leads-to** relations. The implication in (8) represents the base case of the induction. The remainder of the proof of (1) is to show that the number of unbagged bottles eventually decreases by one, i.e., the induction step.

(9) $(\text{must-bag}(1,\alpha) \wedge \alpha > 0) \longmapsto \text{must-bag}(1,\alpha\text{-}1)$

Then we can apply the transitive property of **leads-to** to (7), (8), and (9).

In the proof of (9), two cases are possible: (i) if the unbagged large bottle does not fit in any available bag, or (ii) the unbagged large bottle does fit in an available bag. We define

$\text{fits(I)} \equiv$
$[\exists\ \text{N,W,A} : \text{large-bottle(I)} \wedge \text{unbagged(I)} \wedge \text{bag(N,W,A)}$
$:: \text{W} + \text{lgwgt} \leq \text{maxwgt}]$

The two cases can be stated as:

Case (i) $\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)}$
Case (ii) $\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}$

We know that

(10) $(\text{must-bag}(1,\alpha) \wedge \alpha > 0) \longmapsto$
$([\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)}] \vee$
$[\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}])$

since must-bag(1,α) $\wedge$ $\alpha > 0$ logically implies the disjunction of case (i) and case (ii). Using the transitivity of **leads-to**, the proof of (9) is complete if we show: case (i) **leads-to** case (ii), and case (ii) **leads-to** must-bag(1,α-1). This is done by showing the following:

(11) $(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)})$
ensures
$(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)})$

and

(12) $(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)})$
ensures
$\text{must-bag}(1,\alpha\text{-}1)$

To prove the **ensures** relation in (11) and (12), it must be shown that all transactions either maintain the LHS property of the **ensures** after execution, or change the state of computation to satisfy the RHS property, and that there is at least one transaction that changes the state to the RHS property. Only those transactions that match the tuple step(1) can affect the **ensures** in (11) and (12). When step(1) is in the tuple space, any transaction that does not match step(1) maintains the LHS property. (We are using property 1 in Figure 2 and the fact that every transaction contains a query for the step tuple.) Therefore, for the purpose of this proof, we need only prove (11) and (12) for those transactions in Figure 3.

The first step of the proof for (11) is to show:

$[\forall\ \text{Rule(t)} : 1 \leq \text{t} \leq 3 ::$
$\{(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)}) \wedge$
$\neg(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)})\}$
Rule(t)
$\{(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)}) \vee$
$(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)})\}]$

$\equiv$

$[\forall\ \text{Rule(t)} : 1 \leq \text{t} \leq 3 ::$
$\{\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)}\}$
Rule(t)
$\{\text{must-bag}(1,\alpha) \wedge \alpha > 0\}\ ]$

This property clearly holds since in a state where a large bottle does not fit in any bag, none of the rules do any bagging. Both Rule(1) and Rule(3) maintain the LHS property because their LHSs are not satisfied under the property. Rule(2) creates a new bag upon execution, but does not reduce the number of unbagged bottles. Hence, we have proven

(13) $(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)})$
unless
$(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)})$

[3] Count 1 for each time the predicate is satisfied. Read "the number of I, such that I is a large bottle and I is unbagged."

Since every transaction is reasserted as long as:

$$[\exists : \text{item(I)} :: \text{unbagged(I)}]$$

we know

(14) $\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)} \Rightarrow \text{Rule}(2)$

and based on its definition, Rule(2) actually establishes the RHS of (11) as shown in (15) below.

(15)
$$\{(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits(I)}\} \;\; \text{Rule}(2) \;\; \{(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}\}$$

By (13), (14), and (15), we have proven (11).

The proof of (12) is similar to the previous proof. Again we look only at the transactions to bag large bottles, task 1, since these are the only transactions that can have any effect on (12), because they match **step(1)**.

(16)
$$\{\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}\} \;\; \text{Rule}(1) \;\; \{\text{must-bag}(1,\alpha\text{-}1)\}$$

(17)
$$\{(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}\} \;\; \text{Rule}(2) \;\; \{(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}\}$$

(18)
$$\{\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}\} \;\; \text{Rule}(3) \;\; \{\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}\}$$

In (17) and (18), it is shown that Rule(2) and Rule(3) maintain the LHS of (12). Rule(1) decreases the number of unbagged large bottles, because it will pack an item in a bag. From (16), (17), and (18), we know:

(19)
$$(\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)}) \;\; \textbf{unless} \;\; \text{must-bag}(1,\alpha\text{-}1)$$

Again, all transactions are reasserted into the transaction space as long as there is an unbagged item left in the tuple space. Thus,

(20) $\text{must-bag}(1,\alpha) \wedge \alpha > 0 \wedge \text{fits(I)} \Rightarrow \text{Rule}(1)$

In (16) Rule(1) establishes the RHS of (12). Therefore, by (16), (19), and (20), have proven (12), and hence shown proof of (2).

The proof above gives the flavor of the entire proof of the Bagger program, but only encompasses a small part: that of proving a single task executes correctly. The approach we used in proving progress properties of the entire program was to show first, *that each individual task executed according to its specifications* and then to show, *the ordering of the tasks was correct*[4]. Since each task has a control rule to switch contexts when the task completed, the proof of correct task ordering followed directly. This approach would be typical of RB programs structured using a context element and control rule for tasking. In Bagger, because the tasks were similar, we were able to simplify the proof by generalizing some program properties. The result was a single proof that covered all four tasks.

[4]For example, if task 2 followed task 1, the proof obligation would be: *term(1)* $\longmapsto$ *init(2)*.

Conclusion

This paper presents an assertional approach to the verification of a class of RB programs characterized by the absence of conflict resolution. The verification method is borrowed directly from work in concurrent programming. This work raises two important questions. First, can we eliminate conflict resolution from rule-based programs for the sake of achieving reliability through the application of formal verification methods? Second, can we extend program verification techniques to cover those forms of conflict resolution that appear to be essential to RB programming?

References

Chandy, K.M., and Misra, J. 1988. *Parallel Program Design: A Foundation.* Reading, Mass.: Addison Wesley.

Cunningham, H.C., and Roman, G.-C. 1990. A UNITY-style Programming Logic for a Shared Dataspace Language. *IEEE Transactions on Parallel and Distributed Systems* 1(3):365-376.

Forgy, C.L. 1981. OPS5 User's Manual. Technical Report, CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon University.

Gamble, R.F.; Roman, G.-C.; and Ball, W.E. 1991. On Extending the Application of Formal Specification and Verification Methods to Rule-Based Programming. Technical Report WUCS-91-1, Dept. of Computer Science, Washington University, St. Louis.

Ishida, T. 1990. Methods and Effectiveness of Parallel Rule Firing. *Proceedings of the 6th IEEE Conference on Artificial Intelligence Applications.* Washington, D.C.: IEEE Computer Society Press.

Roman, G.-C., and Cunningham, H.C. 1990. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Transactions on Software Engineering* 16(12):1361-1373.

Schmolze, J.G., and Goel, S. 1990. A Parallel Asynchronous Distributed Production System. *8th National Conference on Artificial Intelligence*, 65-71. Cambridge, Mass.: MIT Press.

Winston, P.H. 1984. *Artificial Intelligence, 2nd Edition.* Reading, Mass.: Addison-Wesley.