Report Number: WUCS-91-1

1991-05-01

Applying Formal Verification Methods to Pure Rule-Based Programs

Rose F. Gamble, Gruia-Catalin Roman, William E. Ball, and H. Conrad Cunningham

Reliability, defined as the guarantee that a program satisfies its specifications, is an important aspect of many applications for which rule-based expert systems are suited. Verification refer to the process used to determine the reliability of the rule-based program. Because past approaches to verification are informal, guarantees of reliability cannot fully be made without severely restricting the system. On the other hand, by constructing formal specifications for a program and showing the program satisfies those specifications, guarantees of reliability can be made. This paper presents an assertional approach to the verification of rule-based programs. The proof logical needed for... **Read complete abstract on page 2.**

Applying Formal Verification Methods to Pure Rule-Based Programs

Rose F. Gamble, Gruia-Catalin Roman, William E. Ball, and H. Conrad Cunningham

Complete Abstract:

Reliability, defined as the guarantee that a program satisfies its specifications, is an important aspect of many applications for which rule-based expert systems are suited. Verification refer to the process used to determine the reliability of the rule-based program. Because past approaches to verification are informal, guarantees of reliability cannot fully be made without severely restricting the system. On the other hand, by constructing formal specifications for a program and showing the program satisfies those specifications, guarantees of reliability can be made. This paper presents an assertional approach to the verification of rule-based programs. The proof logical needed for verification is adopted from one already in use by researchers in concurrent programming. The approach involves using a language called Swarm, and requires one to express program specifications as assertions over the Swarm representation of the program. Among models the employ rule-based notation, Swarm is the first to have an axiomatic proof logic.

Applying Formal Verification Methods to Pure Rule-Based Programs

Rose F. Gamble
Gruia-Catalin Roman
William E. Ball
H. Conrad Cunningham[†]

WUCS-91-1

May 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

To appear in the International Journal of Expert Systems Research and Applications.

[†]H. Conrad Cunningham is associated with the Department of Computer and Information Science, University of Mississippi, Univeristy, MS 38677.

Applying Formal Verification Methods to Pure Rule-Based Programs

Rose F. Gamble[1]
Gruia-Catalin Roman
William E. Ball[2]
Department of Computer Science
Washington University
St. Louis, MO 63130

H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi
University, MS 38677

Key Words: rule-based programming, verification, formal specification, constraints, termination

Abstract

Reliability, defined as the guarantee that a program satisfies its specifications, is an important aspect of many applications for which rule-based expert systems are suited. *Verification* refers to the process used to determine the reliability of the rule-based program. Because past approaches to verification are informal, guarantees of reliability cannot fully be made without severely restricting the system. On the other hand, by constructing formal specifications for a program and showing the program satisfies those specifications, guarantees of reliability can be made.

This paper presents an assertional approach to the verification of rule-based programs. The proof logic needed for verification is adopted from one already in use by researchers in concurrent programming. The approach involves using a language called Swarm, and requires one to express program specifications as assertions over the Swarm representation of the program. Among models that employ rule-based notation, Swarm is the first to have an axiomatic proof logic.

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

Contents

1 Introduction

Rule-based expert systems are generally used in applications where all or part of the reasoning performed by a human expert can be emulated. Such systems can be successful in applications where *reliability*, defined as the guarantee that a program satisfies its specifications, is not a dominant factor. For this reason, often expert systems are used as "voting partners" and require extensive human interaction or monitoring during execution. However, reliability is a major concern in applications involving critical decisions and in stand-alone systems. In these domains, expert systems cannot yet be utilized effectively, even though they may be well suited conceptually.

Reliability requires the *validation* and *verification* of an expert system. Validation of expert systems refers to determining if the specifications of the system accurately represent the knowledge of an expert in a particular problem domain [22]. Verification is the process of guaranteeing a system meets the given specifications. In this paper, we concentrate on verification only, and make the assumption that those specifications of the program that can be expressed formally have already been validated. We recognize that there exist cases in which the expert's assessment of the problem cannot be quantified precisely.

Currently, verifying an expert system is an informal process. The guarantees that can be made require testing on various aspects of the expert system, such as whether there are cycles or redundancies in the rules, or inconsistencies in the data [19, 21, 17, 4]. Because of the difficulty in making guarantees of correct behavior in traditional systems, some approaches to verification choose to restrict the rules to having single actions and/or only adding to the set of facts, not deleting [29]. In current approaches, input test cases are generated to exercise the system and to determine if output results are accurate [5]. Though many input test cases can be generated, there may still be some cases, not tested, that cause anomalies to occur. If the constraints on the input are ill-defined, then it is not known when the system may give inaccurate output results. Finally, many of the algorithms used in testing the expert system are intractable, making it necessary to use approximation algorithms for most domains. The benefit of such approaches is the automation capabilities. But this benefit must be weighed against the fact that guarantees for reliability cannot be fully given.

Formal verification of rule-based programs can make guarantees that the program satisfies a given set of specifications which define its correct behavior [6, 27, 32]. This paper shows that rule-based programs may be formally verified using assertional methods and that the proof logic needed for verification can be adopted from one already in use by researchers in concurrent programming. The proof logic we will be using was originally developed by Chandy and Misra for UNITY [3], a concurrency model based on conditional multiple assignment statements to shared variables. This proof logic was later generalized by Cunningham and Roman [7] 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 [12]. By using the Swarm proof logic, a general approach to proving certain properties of rule-based programs that utilize tasking and context switching has been developed [11]. Moreover, the Swarm proof logic is applicable to both sequential and concurrent rule-based programs.

Swarm proofs involve two kinds of program-wide properties: *safety properties* that guarantee the program does nothing wrong, and *progress properties* that guarantee the program does something useful. Because the development of rule-based programs often involves incomplete knowledge about the application domain as given by a human expert, there may be improper interactions among rules that are not readily apparent. With the development process in mind, the main verification goals addressed by others focus on the general areas of: consistency of information, generation of correct output, and termination. In order to utilize the verification methodology given in Swarm, these goals must be reformulated as safety and progress properties to be satisfied by the program.

The goal of verifying consistency includes many issues. It may be referred to as rule inconsistency or data inconsistency. McGuire [19] defines the goal of rule inconsistency detection as checking if contradictory information can be derived from the assumed presence of some set of facts. The DEVA system [29] uses a generated fact-based scenario to search for rule inconsistency. The types of rule inconsistency searched for are: the ability to satisfy a pair of rules which conclude contradictory information, the ability to prove a literal and its explicit negation, and the ability to prove the existence of a literal and the deletion of the same literal. Waldinger [32] defines the goal of verifying consistency with respect to the data, in that it should be proven that no input produces an inconsistent fact-base. Verifying a consistent fact-base involves describing the information that is conflicting and showing it cannot occur simultaneously. Rushby and Whitehurst [27] give a more rigorous definition of this view of consistency. They rely on model theory and extend consistency to require that inputs that are close together should generate outputs that are close together. Swarm proofs view consistency as a constraint. Therefore, it is expressed as a safety property, which simply states that information considered conflicting cannot be present in the same program state. Such descriptions would include but not be restricted to disallowing data and its negation to occur simultaneously.

A program cannot be considered reliable if the output resulting from the program execution is inaccurate. In general, to achieve this verification goal, an expert system is provided with a set of inputs. The outputs it prescribes are compared to the human expert's response to the same inputs [22]. Some quantitative analysis is performed to determine if the inputs are close enough to the expert's to be considered accurate. Chang et al [4] automatically generate multiple input test cases and compare the output of the expert system to the expected output. Any discrepancies cause a revision of the rules. They admit that it is not feasible to generate all possible input test cases. Therefore, they state input and output constraints on individual components of a system and prove the output constraints are satisfied given the input constraints. They can perform these proofs by restricting their rules to Horn clauses, and only allowing addition to the fact-base, not deletion. To determine if the output of a rule-base program is accurate, we also detail constraints on both the input and output, and prove that the output generated meets these constraints. This method allows flexibility in the range of accuracy of the output, but still places quantitative bounds on it. It is not necessary to restrict the rules to single actions and only additions. The specification of output constraints requires the use of both safety and progress properties. Safety properties are involved in characterizing the expected output of a program, while progress properties state the requirement that the output must be actually produced.

Most rule-based programs are required (by their specifications) to terminate. In previous research, verifying termination for a particular rule-based program can be given only by testing the program on multiple input test cases. The termination criteria for a program in Swarm is formulated as a progress condition. Progress properties are necessary because a termination state is a state to occur in the future. It must be shown that this state is eventually reached provided the program is started in some initial state under certain input constraints.

The verification concerns discussed previously all utilize the program state to determine if the system is behaving correctly. These can easily be expressed as safety and progress properties, and be readily proven. Aside from the type of verification goal that involves the behavior of the program in a particular program state, there is another type of verification goal that involves the syntactic structure of the rules. Many concerns dealing with the syntactic structure of the rules of the system, e.g., completeness and redundancy, can also be expressed as safety and progress properties, but require the introduction of auxiliary variables, for proof purposes.

As the rules of an expert system are developed, experts are consulted and changes are made according to the experts' evaluation of the system. Since rigorous verification is not performed during the cycle of designing the rules and consulting experts, additional rules or changes to old rules may cause unnecessary dependencies and destroy useful dependencies. Changes to the data may also occur without updating the rules to reflect these changes. Completeness and redundancy are of interest due to this manner in which most large rule-based expert systems are developed. A complete rule-based system is one in which there are no gaps in the knowledge base that result from missing rules, unreachable clauses and deadend clauses [21]. A rule is believed missing if there is a range of possible data values for some attribute, yet the range is not totally covered by the existing rules. An unreachable clause is one that exists in the knowledge base and can never be utilized by the current system. A deadend clause contains a condition that can never be instantiated no matter what data is used with the system. Exhaustive checking for completeness is not tractable and algorithms have been developed to detect as many of the problems resulting in and from an incomplete system as feasible. This checking may involve creating a data flow diagram and checking for the connectedness of all components [21] or looking at all the possible instantiations of rules in a particular system using logical decision table checking [31]. The Expert System Validation Associate [30] extends this approach by exploiting knowledge about constraints and dependencies within the system to eliminate rows and columns from the decision table to make the checking more feasible.

A rule is redundant if there exists a consequent on the right-hand side of the rule which can be independently derived without the rule in question being used in the inference process [19]. Problems may arise if there is a program state that can be reached by redundant rules that is deemed incorrect later in the system's development. If only one of the redundant rules is altered, the incorrect state can still be independently derived without using the now altered rule, and it may go undetected. Verifying the absence of redundancy involves instantiating the system on certain facts and simulating the inference, as in the process of generate-and-test. McGuire [19] describes a method called a residue approach, which attempts to derive those test case facts that may cause redundancy, restricting the checking to a more feasible level.

We believe that the concern with completeness and redundancy is motivated by limitations in the previous approaches to rule-based program development and are not fundamental problems. Therefore a systematic formal approach will eliminate the need for the current algorithmic checks. Proofs of completeness and lack of redundancy are not addressed in this paper because they have distinct objectives that diverge from the verification goals we have considered.

Section 2 gives the notation for rule-based programs in Swarm. Next, Section 3 introduces a sample program, and its representation in Swarm. Section 4 describes the method of formal verification used in this paper. The actual verification of the sample program begins in Section 5 by defining what is meant by the structural and behavioral constraints of a program and giving an example proof of such constraints related to our sample program. Section 6 details how consistency is viewed and gives an example proof in relation to the sample program. Sections 7 and 8 are concerned with verifying the output of a program. Section 9 discusses the formulation of the termination criteria of a Swarm program and its proof for the sample program. Section 10 presents the related work in the area of formal verification of rule-based programs. Section 11 contains a brief summary and conclusion.

2 Notation

Swarm belongs to a class of languages and models that use tuple-based communication. Other languages and models in this class are Linda [2], Associons [24], GAMMA [1], and Tuple Space Smalltalk [18]. There are many important features of Swarm that make it amenable to verifying rule-based programs. First, Swarm provides a number of constructs used by rule-based programming languages. One such construct is content-based addressing of data. Another construct is that transactions in Swarm act like rules in that a successful query results in changes to a database of content-addressable entities. Swarm also provides an inference mechanism to cycle through the execution of multiple rules. Usual rule-based programming constructs such as context switching or tasking can easily be represented in Swarm.

There are important differences between Swarm and traditional rule-based programming languages, such as OPS5 [9]. First, and most important, Swarm provides an assertional proof logic which can be used to verify rule-based programs. Swarm was developed for research in concurrent programming. Thus, control over serial execution in the form of conflict resolution is not directly available in Swarm. At this time, conflict resolution must be made explicit within the transaction queries. If conflict resolution cannot be made explicit, the program must be shown to meet the intended goals of the desired conflict resolution strategy. Transactions in Swarm are dynamic. Hence, all possible transactions may not be available for execution at all times, as rules are in a rule-based program. Also, transactions are represented by a class name which makes proving properties about their actions easier.

2.1 Working Memory and Production Memory

The Swarm dataspace is partitioned into two parts[3]. The *tuple space* or working memory (WM) contains a set of tuples or working memory elements (WMEs) of the form

$$\mathbf{class(a1,a2,...,an)}$$

where **class** is the class name of the tuple and **a1, ..., an** are non-variable attribute values, i.e., WMEs are grounded.

The *transaction space* or production memory consists of a set of transactions that indicate possible actions to be taken by the program. Each transaction may be viewed as a parameterized rule. A simple transaction has the form

$$\mathbf{T(i) \equiv LHS \longrightarrow RHS}$$

[3]There is actually another partition in the dataspace called the *synchrony relation*, but it is not utilized for this paper.

where **T** is the class name of the transaction and **i** is an attribute value of that class. The **LHS** is made up of a conjunction of condition elements (CEs) and the **RHS** is made up of action elements (AEs). Every CE represents a pattern or template matched against WMEs. A positive CE is satisfied when there exists a matching WME. A negative CE succeeds when no matching WME can be found. Every AE is made up of WMEs and specifies modifications to WM. Positive AEs add to WM and negative AEs delete from WM. In Swarm, transactions can be defined without being in the transaction space. Swarm makes a distinction between the definition of a transaction and its existence. Only transactions that exist in the transaction space may be executed. For example, the transaction **T(i)** may be defined, but like a WME, **T(i)** must be grounded (by instantiating **i**) if the transaction is to be in the transaction space. This does not mean that the entire transaction must be instantiated, only the attribute values of the particular class of transactions. A transaction must be specifically asserted into the transaction space, either initially or as an AE during the execution of another transaction.

Complex transaction definitions use a parallel bar operator (||) to combine simple definitions of rules, also called *subtransactions*, into a single transaction. A complex transaction in Swarm is represented as follows.

$$
\begin{array}{ll}
T(j) \equiv & \\
 & LHS_1 \longrightarrow RHS_1 \\
\parallel & \\
\vdots & \\
\parallel & \\
 & LHS_n \longrightarrow RHS_n
\end{array}
$$

Complex transactions regard all their subtransactions simultaneously as explained in the next section.

2.2 Inference Engine

In OPS5 [9] and similar rule-based systems, the inference engine performs a **match-select-act** cycle until a halt statement (a type of AE) is reached or the match phase returns the empty set. One can think of the match phase as comparing the LHS of all rules to WM. A match for every CE in a rule constitutes an instantiation of that rule. A rule may have one or more instantiations. All instantiations from all rules are gathered at the end of the match phase to form a set. This set of instantiations, called the *conflict set*, is passed through the select phase. During this phase, a conflict resolution strategy determines a single instantiation, or some subset of the conflict set, which is passed to the act phase. The act phase performs the actions of the RHS of those instantiations passed. The results of this phase may be modifications to WM, and/or calling subroutines and/or modifications to PM[4]. Once all actions are performed the cycle begins again with the match phase.

In this paper we will concentrate on the class of *pure* rule-based programs. In such programs no conflict resolution strategy is used. Instead, instantiations are chosen nondeterministically for execution from the conflict set. Many rule-based programs that depend on some form of conflict resolution can be reformulated as pure rule-based programs. The conversion of such programs is not addressed in this paper.

[4]We will only concentrate on modifications to WM in this paper.

```
T(i) ≡
        X,Y : out(X) ∧ in(Y) ⟶ in(Y)†, out(Y)
        ||
        Z : in(Z) ⟶ T(i);
```

Description: (a) T(i) is the transaction name, for some variable i. (b) X, Y, Z are dummy variables. (c) out(X) is the first condition element 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 (⟶) separates the LHS and RHS. (e) In the RHS, the dagger (†) after the action element in(Y) 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 first condition element in the second subtransaction, which reasserts the transaction as an action element as long as in(Z) is in the tuple space.
For notational convenience, the above transaction can be rewritten as:

```
T(i) ≡
        X,Y : out(X), in(Y)† ⟶ out(Y)
        ||
        Z : in(Z) ⟶ T(i);
```

Figure 1: Example of Swarm Transaction.

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 every transaction in the transaction space will eventually be chosen. As a by-product of this choice, the transaction is deleted from the transaction space (although it may explicitly reassert itself or another transaction may reassert it). 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. A transaction is satisfied if one of its subtransactions is satisfied. Only tuples may be deleted from the tuple space, but both tuples and transaction may be asserted into the dataspace. Termination occurs when no transactions are left in the transaction space. Figure 1 presents the tuple and transaction notation of Swarm.

2.3 Direct Translation of Programs

Pure rule-based programs can be translated to Swarm without modification to the rules. Each rule in a pure rule-based program is represented as a subtransaction of a distinct transaction. In addition, the termination conditions of the pure rule-based 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. When the termination conditions are satisfied, each chosen transaction will *not* be reasserted into the transaction space. Since the non-deterministic choice of transactions is fair, the termination conditions of the rule-based program cause the Swarm program to terminate (though it still must be formally proven). Thus, each transaction in Swarm contains two subtransactions: (1) the direct translation of a single rule in the rule-based program and (2) the negated termination conditions of the rule-based program for reassertion of the transaction. If all possible transactions are initially placed in the transaction space, since a transaction is chosen nondeterministically and has an effect only if the LHS of a state-changing subtransaction is satisfied, the execution sequences produced are those of a pure rule-based program.

3 Sample Program

We use the Bagger problem [33] to illustrate how the Swarm proof logic may be employed for verifying rule-based programs. Bagger is a toy expert system to bag groceries according to certain characteristics, much like one would desire grocery items to be bagged in the store. The program is easily expanded to utilize items with many characteristics. We use the original example in which: (i) an item is one of three sizes (small, medium, or large), (ii) a large item may be contained in a bottle, and (iii) a small or medium item may be frozen. Winston gives the original rules, which rely on conflict resolution for the correct order in which the rules are executed. We have changed those rules only to make conflict resolution explicit in the rule's LHS and we have eliminated some extraneous information for notational convenience. The Bagger problem was chosen because: (1) it can be fully specified formally, (2) it can be stated as a pure rule-based program, and (3) it exhibits some basic properties of rule-based programming, such as tasking and context switching.

Specifically, Bagger is given a set of unbagged grocery items represented by tuples (WMEs) 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,C,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 contained in a bottle*. The third field gives *one of three possible weights that determines if the item is small, medium or large*. These weights are: *lgwgt, medwgt, smwgt* respectively. The last field corresponds to a boolean value representing *whether or not the item is frozen*. Execution of the program must place those items that are unbagged in a bag, in a predefined order, with large bottles first, then large items, followed by medium items, then small items. In the cases of medium and small items, those that are frozen are packed before those that are not.

Bags should only be created when needed. To represent a bag, a tuple of type **bag(N,W,A)** is placed in the tuple space. The first field is the *bag's unique identification number*. The next field is the *total weight of the bag*. The last field is a *sequence containing the identification numbers of the items placed in the bag*. 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*. A context element is normally present in the LHS of every rule in order to segregate the rules into tasks, or contexts. It is used often in rule-based programs along with a *control rule* that switches contexts or tasks, so that another group of rules can execute. If a context element is used, there is normally only one tuple in that class. Rules without a context element do not correspond to any task. In Bagger, the tuple **step(B)** is present in every rule and divides the rules into tasks 1, 2, 3, and 4, depending on the value of B. 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.

```
Program Bagger (maxwgt,lgwgt,medwgt,smwgt,maxitems,steps:

          natural(maxwgt); natural(lgwgt); natural(medwgt); natural(smwgt);
          maxwgt ≥ lgwgt ≥ medwgt ≥ smwgt;
          natural(maxitems); maxitems ≥ 0;
          steps = 4)

definitions

[ I : natural(I) ::
          item(I) ≡ 1 ≤ I ≤ maxitems;
          small-item(I) ≡ item(I) ∧ [∃ C,W,F : grocery(I,C,W,F) :: W = smwgt];
          medium-item(I) ≡ item(I) ∧ [∃ C,W,F : grocery(I,C,W,F) :: W = medwgt];
          large-item(I) ≡ item(I) ∧ [∃ W,F : grocery(I,false,W,F) :: W = lgwgt];
          large-bottle(I) ≡ item(I) ∧ [∃ W,F : grocery(I,true,W,F) :: W = lgwgt];
          frozen(I) ≡ item(I) ∧ [∃ C,W :: grocery(I,C,W,true)]
]

tuple types

[ I,C,F,W,A,N,B :
          A ∈ [• I : item(I) :: I]*, item(I), boolean(C), boolean(F),
          1 ≤ W ≤ maxwgt, natural(N), 1 ≤ B ≤ steps ::
                    grocery(I,C,W,F);
                    bag(I,W,A);
                    unbagged(I);
                    current(N);
                    step(B)
]
```

Figure 2: Bagger Program Definitions and Tuple Type Declarations

3.1 Swarm Representation of Bagger

Each of the rules in Bagger is directly translated to a single subtransaction of a distinct transaction whose class name corresponds to the rule name. As described in Section 2, a second subtransaction is added to every transaction. The LHS of this second subtransaction tests whether any items are still unbagged. If so, then the transaction is reasserted. This test corresponds to negating the termination conditions of Bagger: that all items are bagged. Figure 2 presents the program header, predicate definitions, and tuple type declarations for the Swarm version of the Bagger program[5]. Figure 3 gives the transaction definitions[6]. Figure 4 gives the initialization section of the program.

[5]We use a three part notation, e.g., [∀ I,C,W,F : grocery(I,C,W,F) :: 0 ≤ W ≤ *maxwgt*], defined as follows: the first part, up to the single colon, consists of a quantifier and a list of quantified variables; the middle part restricts the domain of values that may be assumed by the variables; the third part, after the double colon, consists of a predicate. If the single colon is missing, the domain is not restricted. Operators, such as • for concatenation, can be used in the place of quantifiers. In such cases, the third part of the notation defines a list of operands over which the operator is applied.

[6]The symbol <> represents the empty sequence.

Rules to bag large bottles

```
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),
        [∀ M,W,A : bag(M,W,A) :: W > maxwgt - lgwgt],
        current(N)†
        ——→
        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);
```

Rules to bag large items

```
Rule(4) ≡
    I,N,W,A :
        step(2),
        large-item(I), unbagged(I)†,
        bag(N,W,A)†, W ≤ maxwgt - lgwgt
        ——→
        bag(N, W+lgwgt, A • I)
||
    I :  unbagged(I) ——→ Rule(4);

Rule(5) ≡
    I,N :
        step(2),
        large-item(I), unbagged(I),
        [∀ M,W,A : bag(M,W,A) :: W > maxwgt - lgwgt],
        current(N)†
        ——→
        bag(N+1, 0, <>),current(N+1)
||
    I :  unbagged(I) ——→ Rule(5);

Rule(6) ≡
        step(2)†
        [∀ I : large-item(I) :: ¬unbagged(I)],
        ——→
        step(3)
||
    I :  unbagged(I) ——→ Rule(6);
```

Rules to bag medium items

```
Rule(7) ≡
    I,N,W,A :
        step(3),
        medium-item(I), frozen(I), unbagged(I)†,
        bag(N,W,A)†, W ≤ maxwgt - medwgt
        ——→
        bag(N, W+medwgt, A • I)
||
    I :  unbagged(I) ——→ Rule(7);

Rule(8) ≡
    I,N,W,A :
        step(3),
        medium-item(I), unbagged(I)†,
        [∀ J : medium-item(J) ∧ frozen(J) :: ¬unbagged(J)],
        bag(N,W,A)†, W ≤ maxwgt - medwgt
        ——→
        bag(N, W+medwgt, A • I)
||
    I :  unbagged(I) ——→ Rule(8);

Rule(9) ≡
    I,N :
        step(3),
        medium-item(I), unbagged(I),
        [∀ M,W,A : bag(M,W,A) :: W > maxwgt - medwgt],
        current(N)†
        ——→
        bag(N+1, 0, <>), current(N+1)
||
    I :  unbagged(I) ——→ Rule(9);

Rule(10) ≡
        step(3)†,
        [∀ I : medium-item(I) :: ¬unbagged(I)]
        ——→
        step(4)
||
    I :  unbagged(I) ——→ Rule(10);
```

Rules to bag small items

```
Rule(11) ≡
    I,N,W,A :
        step(4),
        small-item(I), frozen(I), unbagged(I)†,
        bag(N,W,A)†, W ≤ maxwgt - smwgt
        ——→
        bag(N, W+smwgt, A • I)
||
    I :  unbagged(I) ——→ Rule(11);

Rule(12) ≡
    I,N,W,A :
        step(4),
        small-item(I), unbagged(I)†,
        [∀ J : small-item(J) ∧ frozen(J) :: ¬unbagged(J)]
        bag(N,W,A)†, W ≤ maxwgt - smwgt
        ——→
        bag(N, W+smwgt, A • I)
||
    I :  unbagged(I) ——→ Rule(12);

Rule(13) ≡
    I,N :
        step(4),
        small-item(I), unbagged(I),
        [∀ M,W,A : bag(N,W,A) :: W > maxwgt - smwgt],
        current(N)†
        ——→
        bag(N+1, 0, <>), current(N+1)
||
    I :  unbagged(I) ——→ Rule(13);
```

Figure 3: Bagger Transaction Type Declarations

```
initialization

The following tuples represent a sample grocery list with maxwgt = 15, lgwgt = 6, medwgt = 4, smwgt = 2 and maxitems = 8.

    grocery(1,false,2,false);
    grocery(2,false,2,true);
    grocery(3,false,4,false);
    grocery(4,false,4,true);
    grocery(5,false,4,false);
    grocery(6,flase,6,false);
    grocery(7,false,6,false);
    grocery(8,true,6,false);

    [I : 1 ≤ I ≤ 8 :: unbagged(I)];

    current(0);

    step(1);

    [t : 1 ≤ t ≤ 13 :: Rule(t)]
```

Figure 4: Sample Initialization Section for Bagger

4 Formal Verification

The goals of this paper are to illustrate that many interesting properties of a rule-based program can be proven and to give a sense of how to approach the proofs of these properties. Properties that can be formalized as part of the original problem specification are generally the most interesting and should be proven in all programs. Such properties are normally concerned with the output results. Termination may be implied by the problem specification or explicitly stated. As the actual program is formulated, other interesting and necessary properties may be added to the program specification. These properties include characterizing the inputs to the system, called the *input constraints*, constraints that the program must uphold because of its specific design, called *structural constraints*, and constraints that the program must uphold in order for the program to satisfy the original specifications, called *behavioral constraints.* For example, in Bagger the original specification consists of three main properties: (a) the items are ordered in a bag according to their weight and whether they are frozen or not, (b) bags are created as needed, (c) all bags cannot exceed some maximum weight. There are many different rule formulations that can be used to satisfy these properties. An input constraint in our formulation of Bagger is that the tuple step(1) be present initially in working memory. An example of a structural constraint associated with our program is that at all times there can be only one tuple of type step(B) for restricted values of B. A behavioral constraint specific to our program states that once a grocery item is placed in a bag, it must remain in the same bag in the original position in which it was placed. This property is necessary for proving properties (a) and (c) above, because our program assumes an item is properly packed in a bag the first time. Another program may randomly bag the groceries and then check the ordering of the items in the bag. If the order is not correct, the contents of the bag are dumped out and repacked. Thus, the previously stated behavioral constraint would not apply. Consistency constraints may be part of any of the previous categories of properties, but are considered as a separate verification concern. Since we use the same basic rules that Winston [33] described for Bagger, we have a set of specifications that are biased to this encoding.

In this section, we describe the Swarm proof theory [7]. Definitions will be given for the formal meaning of a safety and progress property. This proof logic is built around assertions that express program-wide properties. Such properties encompass the entire knowledge base and database of a rule-based program. The Swarm proof logic is based on the UNITY [3] 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.

Safety properties are used to express constraints on the program, whether they are output constraints, structural constraints, behavioral constraints, or consistency constraints. As in UNITY's proof logic, the basic safety properties of a program are defined in terms of **unless** relations.

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

where 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,

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

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. We shorten the reference to an invariant property by using **inv**.

The generation of output and program termination are expressed as progress properties. 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 t is 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,

$$\frac{[\forall\ m : m \in W :: p(m) \longmapsto q]}{[\exists\ m : m \in W :: p(m)] \longmapsto q}$$

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*.

The next several sections are devoted to applying the Swarm proof logic to verify that the safety and progress properties expressed for our formulation of Bagger hold. For each category of constraints, we present the properties corresponding to that category and give an example proof. Also, we prove the three main properties involved in characterizing the output of Bagger, and show the program terminates. Proofs for properties not shown can be found in [13].

5 Input Constraints

In rule-based programs, the inputs to the program are known and can be characterized in the form of initial constraints. The constraints are assumed to be met by all inputs. In a real application, there may be a possibility of inputs contradicting each other such as in sensor readings. In this case, some preprocessing must be available to determine which input is given to the system. This step is necessary since any conclusion can be implied by inputs that do not meet the initial conditions. These conditions must be as complete as necessary to ensure those safety and progress properties in the specification of the program that rely on a defined initial program state. We let INIT represent the initial conditions or input constraints of a program. In Bagger, INIT is defined as follows (with informal descriptions in italics):

$$\begin{array}{ll}
\text{INIT} \equiv & [\Sigma\ \text{I,C,W,F} : \text{grocery(I,C,W,F)} :: 1] = maxitems \\
& \qquad \textit{The total number of grocery items equals } \text{maxitems.} \\
& \wedge\ [\forall\ \text{I} : \text{item(I)} :: [\exists\ \text{C,W,F} :: \text{grocery(I,C,W,F)}]\,] \\
& \qquad \textit{There exists a grocery tuple for each item.} \\
& \wedge\ [\forall\ \text{I,C,W,F} : \text{grocery(I,C,W,F)} :: \text{W} \leq \text{maxwgt}] \\
& \qquad \textit{Each grocery item can weigh at most } \text{maxwgt.} \\
& \wedge\ [\forall\ \text{I} :: [\exists\ \text{C,W,F} :: \text{grocery(I,C,W,F)}] \Leftrightarrow \text{unbagged(I)}] \\
& \qquad \textit{For every grocery tuple (i.e., grocery(I,B,W,F)), there is a} \\
& \qquad \textit{corresponding unbagged tuple (i.e., unbagged(I)).} \\
& \wedge\ [\forall\ \text{N} :: \text{current(N)} \Leftrightarrow \text{N} = 0] \\
& \qquad \textit{The tuple current(0) is the only one of its type present.} \\
& \wedge\ [\forall\ \text{B} :: \text{step(B)} \Leftrightarrow \text{B} = 1] \\
& \qquad \textit{The tuple step(1) is the only one of its type present.} \\
& \wedge\ [\forall\ \text{t} :: 1 \leq \text{t} \leq 13 \Leftrightarrow \text{Rule(t)}] \\
& \qquad \textit{All transactions are present in the transaction space.}
\end{array}$$

When proving a safety or progress property, the effects of all transactions on the current program state must be considered. The proof process is less tedious and easier to explain if we instead look only to those transactions that can possibly violate the property. In Bagger, we can partition the transaction space into three sets of transactions, characterized by their state-changing subtransactions on the program state: (i) those that pack items in bags, (ii) those that create new bags, and (iii) those that act as control rules and switch contexts. We call these sets of transactions *packers*, *creators*, and *switchers* respectively. The transactions are partitioned as follows.

$$packers = [\cup\ \text{t} : \text{t} \in \{1,4,7,8,11,12\} :: \{\text{Rule(t)}\}] \qquad \textit{transactions that add items to bags.}$$

creators = [$\cup$ t : t $\in$ {2,5,9,13} :: {Rule(t)}]	*transactions that create new bags.*
switchers = [$\cup$ t : t $\in$ {3,6,10} :: {Rule(t)}]	*transactions that change contexts.*

Throughout the proofs, we will refer to a transaction as being *successful*. For this program, in particular, we mean that the state-changing subtransaction in each transaction is satisfied. Recall that each transaction in Bagger contains two subtransactions. If the second subtransaction is satisfied, the transaction is reasserted into the transaction space. If this is the only subtransaction satisfied, no state-change occurs, but the transaction is still considered satisfied, because one of its subtransactions is satisfied. Thus, we use the term successful to distinguish between satisfying the state-changing subtransaction, and the non-state-changing subtransaction. This term is used only if the previously described approach to converting a pure rule-based program to Swarm is used (Section 2).

The tuple of type **grocery(I,C,W,F)** for some I,C,W, and F is the data structure that holds all of the information pertaining to a grocery item. There are certain restrictions that must be placed on the manipuation of tuples of this type. We present these restrictions here, not because they are considered input constraints, but because they define the characteristics of specific input data throughout the execution of the program. The following safety properties express these restrictions.

Property 1. **inv** [Σ I,C,W,F : grocery(I,C,W,F) :: 1] = maxitems
The total number of grocery items equals maxitems
Property 2. **inv** [$\forall$ I : item(I) :: [Σ C,W,F : grocery(I,C,W,F) :: 1] = 1]
For every item, there is exactly one grocery item

Property 1 is part of INIT. Since no transactions add or delete **grocery(I,C,W,F)** for all values of I,C,W, and F, the invariant holds.

For Property 2, initially the invariant holds because it is implied by INIT. The first conjunct of INIT states that there are a total of *maxitems* grocery tuples. The second conjunct of INIT states that for each item I, where $1 \leq I \leq$ *maxitems*, there is at least one grocery tuple. Since there can only be *maxitems* of such tuples, there must be exactly one for each I, such that item(I). Since no transactions add or delete **grocery(I,C,W,F)** for all values of I,C,W, and F, the invariant is not violated.

6 Consistency Constraints

A property that constrains the classes of working memory elements or the individual working memory elements that can exist simultaneously in working memory at any time during program execution is a consistency constraint. These constraints may fall into any of the categories of input, structural, behavioral or output constraints, depending on the working memory elements that are constrained. Thus, there is overlap between these constraints and the other categories of constraints. Because of the previous usage of consistency, we maintain this category only for characterizing those classes of working memory elements that may be semantically contradictory. By semantically contradictory, we mean that two distinct classes of working memory elements have opposite meanings and the presence of certain elements of both classes simultaneously in working memory yields an inconsistent state.

In Swarm, a working memory element either exists or does not exist, so there can be no consistency problems of having a literal and its negation exist simultaneously in working memory. Therefore, in Swarm programs, consistency constraints are necessary to describe those classes of working memory elements that are semantically contradictory. For example, in Bagger, whatever the formulation of the program, it would not be suitable for a particular grocery item to be unbagged and bagged at the same time. In our particular program, we use the working memory class **unbagged(I)** for some item I to represent an unbagged item, and the tuple **bag(N,W,A)** for some N,W,A, to represent a bag in which the item I is packed. Under certain conditions, working memory elements within these two classes are semantically contradictory. Also, The absence of an element in some class does not necessarily mean its negation. For example, if ¬**unbagged(I)** is true, then the tuple does not exist in the tuple space. If $I < 0$ or $I > maxitems$, then the non-existence of this tuple is not of concern. But if item(I) is true, i.e., $1 \leq I \leq maxitems$, then the non-existence of this tuple should mean that the item has been packed in a bag. Thus, the consistency constraint not only expresses that an item cannot be unbagged and bagged simultaneously, but also gives meaning to the negation of a working memory element in a particular class. Below, the consistency constraint is formally stated as a safety property and is followed by its proof.

Property 3. **inv** [∀ I : item(I) :: ¬bagged(I) ⇔ unbagged(I)]
At any time, a grocery item is either inside or outside of a bag
where bagged(I) *is defined by*
bagged(I) ≡ [∃ N,n :: placed-in(I,N,n)]
and
placed-in(I,N,n) ≡ [∃ W,A : bag(N,W,A) :: A.n = I]

Initially **unbagged(I)** is true for all I such that item(I) and there are no tuples of type **bag(N,W,A)** for all N,W and A. Thus, the invariant holds initially.

The proof of equivalence is somewhat obvious. No transactions add a tuple of type **unbagged(I)** to the tuple space. The tuple **unbagged(I)** may be deleted by a transaction in the set *packers* (henceforwared called a *packer* transaction) if and only if it is placed in a bag yielding

$$[\exists\ N,W,A,n : bag(N,W,A) \land n \leq |A| :: A.n = I]$$

which by definition is bagged(I). The notation |A| means "the size of sequence A." This yields the equivalence of

$$\neg unbagged(I) \equiv bagged(I)$$

which guarantees the invariant.

7 Structural and Behavioral Constraints

Structural and behavioral constraints are properties that restrict the data structures in working memory and dictate the strategy the program follows in order to satisfy its goals. Therefore, these constraints are program specific. In this section, we define both structural and behavioral constraints, normally expressed as safety properties, and give those constraints associated with the formulation of Bagger that are used. An example proof is given for one of each type of constraint.

7.1 Structural Constraints

A *structural constraint* is defined as a property that constrains the organization of the data structures in working memory in order for the program to execute correctly. There are four safety properties that express structural constraints in Bagger.

Property 4. **inv** $[\Sigma\ B : \text{step}(B) :: 1] = 1$
There is exactly one context element always present

Property 5. **inv** $[\Sigma\ N : \text{current}(N) :: 1] = 1$
There is exactly one current tuple always present

Property 6. **inv** $[\forall\ N :: [\Sigma\ W,A : \text{bag}(N,W,A) :: 1] \leq 1]$
The bags are identified by unique natural numbers

Property 7. **inv** $[\forall\ N,W,A :: \text{bag}(N,W,A) \Rightarrow [\forall\ M : 1 \leq M \leq N :: [\exists\ W',A' :: \text{bag}(M,W',A')]\,]\,]$
The bags are ordered sequentially, beginning with the number 1

Looking at Property 6, one may be tempted to include this property in the category of consistency constraints since working memory is barred at all times from having two distinct tuples **bag(N,W,A)** and **bag(N,W',A')**. But there is no comparison between distinct classes of working memory elements that may cause a semantic inconsistency. Thus, Property 6 remains in the category of structural constraints.

As an example, we give the proof of Property 4 above. The proof of this property is straightforward. As stated in the Section 5, the tuple **step(1)** is part of the initial conditions, and defined in INIT. No transactions assert multiple tuples of type **step(B)**. A transaction from the set *switchers* (henceforward called *switcher* transactions) may delete **step(B)** if and only if **step(B+1)** is reasserted. Thus, the invariant of Property 4 is maintained.

7.2 Behavioral Constraints

Behavioral constraints are those properties which dictate the strategy or reasoning process followed by the program to accomplish its goals. These constraints depend on the particular formulation of the specifications and are program specific. These constraints differ from structural constraints in that they are not targeted precisely to the format or structure of working memory. Instead, they are related to the actions the program performs in order to accomplish its goals.

In the formulation of Bagger presented in Section 3, there are two behavioral constraints expressed as safety properties.

Property 8. $[\forall\ I,N,n :: \textbf{stable}\ \text{placed-in}(I,N,n)]$
A bagged item remains bagged in the same bag and in the same position

Property 9. **inv** $[\forall\ N,N',W,W',A,A',n,n',m : \text{bag}(N,W,A) \wedge \text{bag}(N',W',A') \wedge N \neq N' \wedge 1 \leq n < m \leq |A| \wedge 1 \leq n' \leq |A'| :: A.n \neq A.m \wedge A.n \neq A'.n']$
A bagged item exists once in only one bag

These two properties deal precisely with characterizing an item in a bag, and are necessary for the proofs of the output constraints, output generation, and termination. They also directly relate to the way the program is formulated to accomplish its goals. For example, as stated earlier, it is possible to have a program that randomly places items in a bag and checks their ordering. If the ordering is incorrect, the items are dumped out and repacked, until correct. Then, Property 8, for example, would not be applicable to the program.

As an example, we give the proof of Property 9 above. For the proof, we assume that Property 8 holds, where the predicates bagged(I) and placed-in(I,N,n) are defined in Section 6. (The proof can be found in the appendix). Initially, Property 9 holds because there are no tuples of type **bag(N,W,A)** in the tuple space, for any N,W and A. By the definition of **invariant** in Section 2, we can assume that Property 9 holds prior to the next execution step, in which a transaction is non-deterministically chosen from the transaction space and executed. Only a *packer* transaction can affect the invariant by: (i) packing an item in multiple bags simultaneously or (ii) by packing an item already packed in a bag.

For case (i), an item cannot be added to two bags simultaneously, because no transaction asserts multiple **bag(N,W,A)** tuples for different values of N into the tuple space.

For case (ii), we are given

$$
\begin{array}{ll}
[\exists \text{ N,W,A,n} : \text{bag(N,W,A)} \wedge \text{n} \leq |\text{A}| :: \text{A.n} = \text{I}] & \\
\equiv & \textit{by definition} \\
\text{bagged(I)} & \\
\equiv & \textit{by Property 3} \\
\neg\text{unbagged(I)} &
\end{array}
$$

Assuming that Property 8 above has been proven to hold, we know

$$\textbf{stable } \text{bagged(I)} \equiv \textbf{stable } \neg\text{unbagged(I)}$$

Thus, case(ii) also cannot occur, proving the invariant in Property 9.

8 Output Constraints

Formally characterizing the output of a system requires the use of both safety and progress properties. Safety properties constrain the output to a certain format and range. These properties, called output constraints, are discussed in this section. Progress properties are used to ensure that output is generated. These properties will be discussed in the next section. Normally there is a routine that gives the output result and it is this result that is examined to determine if it meets the specified constraints. In Swarm however, there is no output mechanism, so we must look at how the desired result is characterized in working memory before an output routine is called upon. For example, there are three output constraints associated with Bagger.

Property 10.	**inv** $[\forall$ N,W,A,n,m : bag(N,W,A) $\wedge 1 \leq n < m \leq |A|$ :: f(A.n) $\leq$ f(A.m)]
	The items in each bag are ordered by the function f as follows: (a) large bottles, (b) large items, (c) medium items (frozen items first), (d) small items (frozen item first)
Property 11.	**inv** $[\forall$ N,W,A : bag(N,W,A) :: W $\leq$ *maxwgt*]
	At any time, the weight of every bag cannot exceed the maximum weight allowed
Property 12.	**inv** $[\forall$ N,N',W,W',A,A' : bag(N,W,A) $\wedge$ bag(N',W',A') $\wedge$ N' < N :: W' + weight(A.1) > *maxwgt*]
	At any time, the first item in bag N, cannot fit in any bag M, where M < N and weight(I) = [Σ C,W,F : grocery(I,C,W,F) :: W]

If the appropriate routine existed, the contents of a bag and its total weight may be output in any number of ways, from a simple list to a detailed graphics. Instead of using these representations to verify the output satisfies the constraints, we utilize the working memory elements directly.

The three properties above are considered the most interesting because they comprise the original problem specification. Property 12 expresses a constraint on why bags are created; that there exists some restriction on bag creation. The property is expressed this way because the program state must be used in the safety property. The output constraint proven in this section is Property 10, a safety property that characterizes how the items are ordered in bags. The function f can be informally defined as a mapping of the different size items to the order in which they should be placed in the bag.

Definition 1 Let *items* = [$\cup$ I : item(I) :: {I}] and *priorities* = [$\cup$ J : $1 \leq J \leq 6$ :: {J}]. Then f: *items* $\rightarrow$ *priorities,* such that:

f(I) = 1 if large-bottle(I)
f(I) = 2 if large-item(I)
f(I) = 3 if medium-item(I) $\bigwedge$ frozen(I)
f(I) = 4 if medium-item(I) $\bigwedge$ $\neg$frozen(I)
f(I) = 5 if small-item(I) $\bigwedge$ frozen(I)
f(I) = 6 if small-item(I) $\bigwedge$ $\neg$frozen(I)

The definitions of the above predicates are found in Figure 2. The function g maps the values in the range of f into the values used by B in the tuple **step(B)** as defined by *steps* in Figure 2..

Definition 2 Let g: *priorities* $\rightarrow$ *steps,* such that:

g(J) = 1	if J = 1
g(J) = 2	if J = 2
g(J) = 3	if J = 3 $\bigvee$ J = 4
g(J) = 4	if J = 5 $\bigvee$ J = 6

The composition of f and g can be viewed informally as a mapping of items to the steps in which they are placed in a bag.

The invariant of property 10 holds initially since there are no tuples of class **bag(N,W,A)** for N,W and A. Only a *packer* transaction can violate the invariant. We assume the invariant holds prior to execution of a *packer* transaction, i.e., the ordering of the items currently in all bags is correct. We must show that every *packer* transaction preserves the ordering.

Because we assume the items are correctly ordered prior to the next execution step, it is only necessary to show that the next item packed in any bag does not violate that ordering. Throughout the proof, we rely on Property 3 (from Section 5), and Properties 8 and 9 (from Section 7.2). Then the ordering can be violated only if the next item packed is not ordered correctly with respect to the last item in each bag. Thus, the invariant of Property 10 is reduced to the following proof obligation for all **unbagged(I)**.

$$\textbf{inv}\ [\forall\ N,W,A,n : \text{bag}(N,w,A) \wedge n = |A| :: f(A.n) \leq f(I)] \qquad (1)$$

The following two definitions are given to facilitate the proof of (1).

Definition 3 min-f $\equiv$ [min I,s : unbagged(I) $\wedge$ f(I) = s :: s]

Definition 4 min-f-item $\equiv$ [min I : unbagged(I) $\wedge$ f(I) = min-f :: I]

The items are ordered from 1 to *maxitems*. Thus, we can use the appropriate item number that is minimum without loss of generality, i.e., min-f-item, when referring to any item with an f value of min-f. If there are no tuples of type **unbagged(I)** left in the tuple space, the values of min-f and min-f-item are undefined. Given a *packer* transaction is successful by matching some I such that **unbagged(I)**, we know that either: (i) f(I) = min-f or (ii) f(I) > min-f. In case (i), the invariant is maintained because of the earlier assumption that the ordering of the items in the bags is correct prior to the execution of a *packer* transaction. If min-f-item is packed, all unbagged items have an equal or greater f-value, so the correct ordering is maintained. But in case (ii), the invariant may be violated, since an item with a lesser f-value may be packed at a later time, causing an incorrect order. Hence, to show (1) and therefore Property 10, we must show that the only *packer* transaction that can bag an item must have a precondition that matches **unbagged(I)** with f(I) = min-f. Without loss of generality, we refer to the item that is matched as min-f-item.

For every value, $1 \leq$ min-f ≤ 6, there is exactly one *packer* transaction that can bag min-f-item. For example, if min-f = 4, then we know that medium-item(min-f-item) and $\neg$frozen(min-f-item) are true. Only Rule(8) can pack min-f-item into an available bag (see Figure 3). We will refer to the transaction that can bag min-f-item as min-f-bagger, and show that if an item can be packed, then only min-f-bagger can be successful. To show case (ii) cannot occur, we must show that min-f-item is the only item that can be packed. To do this, we show that the only *packer* transaction that can be successful is min-f-bagger.

The first step in the proof is to show the implications the tuple **step(B)** has on which *packer* transactions are successful.

Lemma 1 **inv** step(B) $\Rightarrow$ B $\leq$ g(min-f)

Initially, **step(1)** is in the tuple space and there are no I, such that **unbagged(I)** and g(f(I)) < 1. Assuming the invariant holds prior to the next execution step, there are two cases to consider that may violate it: (a) g(min-f) is altered by a *packer* transaction and (b) B in **step(B)** is altered by a *switcher* transaction.

Case (a): B $\leq$ g(min-f). The value g(min-f) may be altered if

$$[\Sigma\ I : \text{unbagged(I)} \land \text{g(f(I))} = \text{min-f} :: 1] = 1$$

and min-f-item is packed in the next step by a *packer* transaction. If this occurs, min-f will increase by at least 1. Since the value of B in **step(B)** cannot be altered by any *packer* transaction, the invariant is not violated in this case.

Case (b): B ≤ g(min-f). If B, such that B < g(min-f), is altered by a *switcher* transaction it may be incremented by 1 to at most equal g(min-f). Since, the value of g(min-f) cannot be altered by a *switcher* transaction, the invariant holds. If B = g(min-f), no *switcher* transaction is successful because the precondition of every *switcher* transaction can be restated as (step(B) ∧ [∀ I : g(f(I)) = B :: ¬unbagged(I)). Therefore, the invariant is not violated. **End Lemma 1.**

We know that only those *packer* transactions that match the current **step(B)** can pack an item. By Lemma 1, we know that B ≤ g(min-f). If B < g(min-f), no *packer* transaction is successful, including min-f-bagger. (Actually, only *switcher* transactions can be successful, but that is not a concern.) If B = g(min-f), then at most two transactions can be successful. In the case of 1 ≤ B ≤ 2, only one transaction can be successful, and this transaction is min-f-bagger. In the case of 3 ≤ B ≤ 4 (in which frozen items are bagged before nonfrozen items), it is possible for two transactions to have successful queries because of the value of B. One of these transactions matches **unbagged(I)** such that f(I) = k for 3 ≤ k ≤ 6, and the other matches **unbagged(J)** such that f(J) = k + 1. The precondition of the second transaction can be restated as

$$\text{step(B)} \land \text{unbagged(J)} \land \text{f(J)} = \text{k} + 1 \land [\forall\ I : \text{f(I)} = \text{k} :: \neg\text{unbagged(I)}]$$

The first transaction has no such restriction in its precondition. Thus, if **unbagged(I)** and **unbagged(J)** exist with f(I) = k and f(J) = k + 1, then only the first transaction can be successful, and this transaction is clearly min-f-bagger. If there are no **unbagged(J)** with f(J) = k + 1, then again only the first transaction can be successful, and again this is min-f-bagger. Since the second transaction can be successful only in the absence of **unbagged(I)** with f(I) = k, it is clearly min-f-bagger. Hence, we have shown that if a *packer* transaction is successful, in all cases it can only be min-f-bagger, which only packs min-f-item. Thus, case (ii) cannot occur, proving the invariant of Property 10.

9 Output Generation

In the previous section we showed how output can be characterized using constraints. These constraints, expressed as safety properties, restrict the representation of output in working memory by constraining the format and allowable range of values for certain working memory elements. The proofs guarantee that if output is generated, it satisfies the given constraints. In this section, we concentrate on the generation of output. As stated earlier, progress properties guarantee that the program will do something useful. In this case, progress properties are used to guarantee that output is generated.

There are three progress properties relating to output generation of items.

Property 13. $[\forall \text{ I : item(I) :: unbagged(I)}] \longmapsto [\forall \text{ I : item(I) :: bagged(I)}]$
All unbagged items are eventually bagged

Property 14. $[\forall \text{ N :: current(N)} \wedge [\exists \text{ I : unbagged(I)}$
$\text{:: } [\forall \text{ M,W,A : bag(M,W,A)} \wedge \text{M} \leq \text{N :: weight(I) + W} > \textit{maxwgt}]$
$\longmapsto$
$\text{current(N+1)} \wedge \text{bag(N+1,0,<>)}]$
Bags are created when needed.

Property 15. $[\forall \text{ N :: bag(N,0,<>)} \longmapsto [\exists \text{ W,A : bag(N,W,A) :: W} > 0 \wedge \text{A} \neq \text{<>}]\,]$
Eventually every bag has at least one item

These properties are implied by the program specifications. First, there is some result given by the program, such as bagged items. The implicit goal of Bagger is to bag all the items it is given. We assume these items meet the input constraints of Section 5. Properties 14 and 15 extend the specification that a bag should be created only if there is at least one item to be packed in the bag. To give an example of proving a progress property associated with output generation, we give the proof of Property 13. Because Bagger uses tasking and context switching, a common paradigm in rule-based programming, the program is divided into tasks which follow a predefined order. The approach used to prove the correctness of this property is outlined below. This approach is useful for those rule-based programs that utilize this paradigm. It is not restricted to only those programs whose tasks are have a specific sequential ordering as in Bagger.

1. Specify the initial conditions and termination conditions of each individual task.
2. Prove each task eventually satisfies its termination conditions when started at its initial conditions.
3. Prove the execution ordering of the tasks is correct.

1. Specify the initial conditions and termination conditions of each individual task. As stated earlier, Bagger uses the tuple **step(B)** as a context element to segregate the rules for each of the 4 tasks corresponding to the value of B. Property 4 in Section 7.1, states that there is exactly one tuple **step(B)** at all times in the tuple space, meaning only one task can be active at a time. Let *init(B)* represent the initial conditions of task B, and let *term(B)* represent its termination conditions. In [12], we showed how an individual task in Bagger is proven. Such a proof can be done specifically for each task, but it is easier if the proof is generalized. Fortunately, Bagger has the characteristic that *init(B)* and *term(B)* can be generalized over all B, $1 \leq \text{B} \leq 4$. Thus, only one proof is needed for all four tasks, though it is slightly more abstract.

$$\begin{aligned} \textit{init(B)} \quad \equiv \quad & \text{step(B)} \\ & \wedge [\forall \text{ I : g(f(I)) = B :: unbagged(I)}] \end{aligned} \tag{2}$$

$$\begin{aligned} \textit{term(B)} \quad \equiv \quad & \text{step(B)} \\ & \wedge [\forall \text{ I : g(f(I)) = B :: bagged(I)}] \end{aligned} \tag{3}$$

where g(f(I)) is the composition of functions defined in Section 8.

2. Prove each task eventually satisfies the termination conditions when started at its initial conditions. This is stated as the following **leads-to** relation.

$$\mathit{init}(B) \longmapsto \mathit{term}(B) \quad \text{for } 1 \leq \text{B} \leq 4 \tag{4}$$

Using (2) and (3), this expands to

$$\begin{array}{l} \text{step(B)} \wedge [\forall \text{ I} : \text{g(f(I))} = \text{B} :: \text{unbagged(I)}] \\ \longmapsto \\ \text{step(B)} \wedge [\forall \text{ I} : \text{g(f(I))} = \text{B} :: \text{bagged(I)}] \end{array} \tag{5}$$

for $1 \leq \text{B} \leq 4$. For each task, we are only concerned with the items that can be bagged in that task. To show (4), we use induction on the number of items that can be bagged in task B, i.e., given the current value of B in the tuple **step(B)**. From Section 6, $1 \leq \text{B} \leq 4$

$$\begin{array}{l} [\forall \text{ I} : \text{g(f(I))} = \text{B} :: \text{bagged(I)}] \Leftrightarrow \\ \qquad [\Sigma \text{ I} : \text{g(f(I))} = \text{B} \wedge \text{unbagged(I)} :: 1] = 0 \end{array} \tag{Property 3}$$

Since it is clear that

$$(\text{step(B)} \wedge [\Sigma \text{ I} : \text{g(f(I))} = \text{B} \wedge \textbf{unbagged(I)} :: 1] = 0) \equiv \mathit{term(B)} \tag{6}$$

we need to show that $\mathit{init(B)} \longmapsto (6)$. If there initially are no I, such that g(f(I)) = B for **step(B)**, then the proof of (4) for that value of B is trivial. Assume that for each task there are items to be bagged, i.e.,

$$[\forall \text{ B} : 1 \leq \text{B} \leq 4 :: [\Sigma \text{ I} : \text{g(f(I))} = \text{B} \wedge \text{unbagged(I)} :: 1] > 0] \tag{7}$$

We define for $1 \leq \text{B} \leq 4$

$$\begin{array}{rl} \text{must-bag(B,}\alpha\text{)} \equiv & \text{step(B)} \\ & \wedge [\Sigma \text{ I} : \text{g(f(I))} = \text{B} \wedge \text{unbagged(I)} :: 1] = \alpha \\ & \wedge\ \alpha \geq 0 \end{array} \tag{8}$$

Then, given our assumption in (7),

$$\mathit{init(B)} \Rightarrow (\text{must-bag(B,}\alpha\text{)} \wedge \alpha > 0) \tag{9}$$

and

$$(\text{must-bag(B,}\alpha\text{)} \wedge \alpha = 0\ ) \Rightarrow \mathit{term(B)} \tag{10}$$

for $1 \leq \text{B} \leq 4$. Both (9) and (10) can be stated as **leads-to** relations. The proof obligation becomes

$$(\text{must-bag(B,}\alpha\text{)} \wedge \alpha \geq 0) \longmapsto \text{must-bag(B,0)} \tag{11}$$

for $1 \leq \text{B} \leq 4$. We use induction on the number of unbagged items to show (11). The implication in (10) can be considered the base as case of the induction proof. The remainder of the proof is to show that the number of items that can be packed during each task decreases by one, i.e., the induction step. This is stated as:

$$(\text{must-bag}(B,\alpha) \wedge \alpha > 0) \longmapsto \text{must-bag}(B,\alpha - 1) \qquad \text{for } 1 \leq B \leq 4 \tag{12}$$

Using (12), we can apply the transitivity of **leads-to** to (9), (10), and (11) to conclude (4).

To prove (12), two cases must be addressed: (1) if the unbagged item considered (i.e., the item I with g(f(I)) = B for the current step(B)), *does not fit* in any available bag, or (ii) if the item considered *does fit* in an available bag. We define for all I and for $1 \leq B \leq 4$

$$\begin{aligned} \text{fits}(I,B) \equiv\ & \text{unbagged}(I) \\ & \wedge\ g(f(I)) = B \\ & \wedge\ [\exists\ N,W,A : \text{bag}(N,W,A) :: W + \text{weight}(I) \leq \mathit{maxwgt}] \end{aligned} \tag{13}$$

The two cases can then be stated as:

$$\begin{aligned} &C_1\text{: must-bag}(B,\alpha) \wedge \alpha > 0 \wedge \neg\text{fits}(I,B) \\ &C_2\text{: must-bag}(B,\alpha) \wedge \alpha > 0 \wedge \text{fits}(I,B) \end{aligned}$$

We know that logically, for $1 \leq B \leq 4$

$$(\text{must-bag}(B,\alpha) \wedge \alpha > 0) \longmapsto C_1 \ \vee \ C_2 \tag{14}$$

If we can show that C_1 and C_2 both independently lead to must-bag(B,α - 1), for $1 \leq B \leq 4$, then we can prove (12). (This can also be stated using the disjunction property of **leads-to** in Section 2.) Thus, we must show both:

$$C_1 \longmapsto C_2 \tag{15a}$$
$$C_2 \longmapsto \text{must-bag}(B,\alpha - 1) \tag{16a}$$

which can be restated as

$$\begin{gathered} \text{must-bag}(B,\alpha) \wedge \alpha = k \wedge \neg\text{fits}(I,B) \\ \longmapsto \\ \text{must-bag}(B,\alpha) \wedge \alpha = k \wedge \text{fits}(I,B) \end{gathered} \tag{15b}$$

and

$$\begin{gathered} \text{must-bag}(B,\alpha) \wedge \alpha > 0 \wedge \text{fits}(I,B) \\ \longmapsto \\ \text{must-bag}(B,\alpha - 1) \end{gathered} \tag{16b}$$

for some value k and $1 \leq B \leq 4$. We use **ensures** to prove (15). Relying again on Properties 8 and 9, all transactions either maintain the LHS of the **ensures** or establish the RHS, thus proving the **unless** part of the **ensures**. For every B, such that $1 \leq B \leq 4$, there is a transaction (such as Rule(2) for B = 1) that establishes the RHS of the **ensures** in (15) if the LHS holds in the state of computation prior to the next execution step.

The **leads-to** in (16) is a little more difficult to prove than the one in (15). We first look at the case in which for the tuple **step(B)**, $1 \leq B \leq 2$, i.e., consider the tasks of bagging large bottles and large items only. In this case, all the transactions either maintain the LHS of (16) or establish the RHS, thus proving the **unless** part of an **ensures**. Also, there is exactly one transaction that establishes must-bag(B,α - 1). The **leads-to** relation in (16) for the case of $1 \leq B \leq 2$ holds.

For the case in which $3 \leq B \leq 4$, (16) is more complicated to show because these tasks are each concerned with bagging more than one type of item, i.e., frozen and non-frozen items. Another case analysis is needed and the conditions must be augmented to the LHS of (16). The cases are: (i) if the number of frozen items (for must-bag(B,α) to hold) is greater than zero and frozen(I) is true (for fits(I,B) to hold), and (ii) if the number of frozen items equals zero and $\neg$frozen(I) is true. Remember, we are not concerned about the order in which the items are bagged. That is restricted using the output constraints in Section 8. Here we are concerned only with getting all the items in a bag. We will not go into further detail, but assume (16) can be concluded for $1 \leq \mathrm{B} \leq 4$.

As stated earlier, by proving (15) and (16), we prove (12) using the transitivity of **leads-to**. Then using the transitivity of **leads-to** again on (9), (10), (11), and (12), we have proven (4).

3. Prove the execution ordering of the tasks is correct. In the case of Bagger, the tasks must follow a predefined sequential order. The correctness criteria for this portion of the proof of Property 13 is stated as follows.

$$[\forall \mathrm{B} : 1 \leq \mathrm{B} \leq 3 :: \mathit{term(B)}\ \textbf{ensures}\ \mathit{init(B{+}1)}] \tag{17}$$

The **ensures** in (17) can be proven easily because for each task (except task 4) there is a control rule, or transaction that changes contexts, i.e. for $1 \leq \mathrm{B} \leq 3$, there exists Rule(t) for $1 \leq \mathrm{t} \leq 13$ that causes *init(B+1)* to be true.

At this point in the proof of Property 13 we have shown:

$$\mathit{init(B)} \longmapsto \mathit{term(B)} \qquad \text{for } 1 \leq \mathrm{B} \leq 4 \tag{4}$$

and

$$\mathit{term}(B) \longmapsto \mathit{init}(B+1) \qquad \text{for } 1 \leq \mathrm{B} \leq 3 \tag{17}$$

It is clear that

$$[\forall \mathrm{I} : \mathrm{item(I)} :: \mathrm{unbagged(I)}] \Rightarrow \mathit{init(1)} \tag{18}$$

Then, by (1) and the transitivity **leads-to**, we can conclude

$$[\forall \mathrm{I} : \mathrm{item(I)} :: \mathrm{unbagged(I)}] \longmapsto \mathit{term(1)} \tag{19}$$

and

$$\mathit{term}(1) \longmapsto \mathit{term}(2) \longmapsto \mathit{term}(3) \longmapsto \mathit{term}(4) \tag{20}$$

We rely on Properties 3, 8 and 9 to state that for $1 \leq \mathrm{B} \leq 4$

$$\mathit{term}(4) \Rightarrow [\Sigma\ \mathrm{I} : \mathrm{g(f(I))} = \mathrm{B} :: \mathrm{unbagged(I)}] = 0 \qquad \text{for } 1 \leq \mathrm{B} \leq 4 \tag{21}$$

which can be restated as

$$\mathit{term}(4) \Rightarrow [\forall \mathrm{I} : \mathrm{item(I)} :: \mathrm{bagged(I)}] \tag{22}$$

Then Property 13 can be concluded from (18)-(22).

Though proving a program satisfies it specifications is normally complicated, there are other reasons which contributed to the complexity of the proof of Bagger. One of the main problems is the task variable has no semantic meaning within the actual task performed. This task variable plays an integral part in the behavior of the program, and hence, the proof. To show the program progresses to termination, we must show it progresses through the tasks in the predefined order. To do this, we also have to show that what is accomplished inside a task is correct, and the task itself eventually is completed. Therefore, we had to define a relationship between the task name and work performed under that name. The approach used was to prove that each task executed to completion first, and then prove the order of the tasks was correct. It was very concise for the Bagger program, because the task ordering was strictly sequential. There may be choices involved in task ordering, which would complicate the proof further, but the same approach can still be used. This approach also shows that individual tasks can be shown to satisfy their specifications without showing the entire program satisfies its specifications.

Another reason for the complexity of the proof is the representation of the program we used. The proof showed many places in which the Bagger program could be improved, making the proof more concise and straightforward. For example, if the transaction that creates a new bag also bagged the item tested in its precondition, there would be no need to introduce the predicate *fits* into the progress property proof. Finally, the irregularity of the goals of each task causes unnecessary case analysis, because the first two tasks bag one type of item and the last two tasks bag two types of items. The proof presented in this section would be considerably less complex if these corrections to the program were made. It is often the case in which showing a program meets its specifications offers insight into a better representation of the specifications, and hence, and easier proof. This leads us to believe that incremental proving and development of a rule-based program are essential to creating a correct and efficient program.

10 Termination

There are many problems, such as in diagnostic trouble shooting, for which non-terminating rule-based programs can be utilized. For these programs, it is more important to prove certain goals are reached during program execution and that the cycle of tasks to be continuously repeated by the program is executed in the correct sequence. These proof obligations are similar to those in Section 9, where it was shown that the individual tasks terminate and are correctly ordered during execution. For other programs, eventually reaching a state in which the program terminates is implied by the specification of the program. In this section, we address these programs by presenting the termination criteria of a Swarm program expressed as a progress property, and give the proof of this criteria for Bagger.

As stated in Section 2, a Swarm program terminates when there are no transactions left in the transaction space. A transaction can only be deleted from the transaction space if it is chosen for execution. It can reassert itself or may be reasserted by another transaction. The termination for a Swarm program is formally stated below.

Property 16. $\text{INIT} \longmapsto \text{TERM}$

where

$\text{TERM} \equiv [\forall\ t : t \in \text{TRS} :: \neg[t]\ ]$

For Bagger, TERM can be defined more specifically as

$$TERM_{Bagger} \equiv [\forall\ t : 1 \leq t \leq 13 :: \neg[\text{Rule}(t)]\ ]$$

Our proof obligation is then

$$\text{INIT} \longmapsto \mathit{TERM}_{Bagger} \tag{23}$$

In Section 9, we showed that all unbagged grocery items are eventually bagged by proving each task completes its goal of bagging the appropriate items for that task. By the transitivity of the **leads-to** relation and (4) and (17) in Section 9, we have

$$\mathit{init(1)} \longmapsto \mathit{term(4)} \tag{24}$$

We know *init(1)* is the first executing task because **step(1)** is initially in the tuple space. By Property 4 in Section 7.1, we know there is always exactly one tuple of type **step(B)** where $1 \leq \text{B} \leq 4$. We prove the following lemma stating that the value of B in the tuple **step(B)** is non-decreasing and if it increases, it does so in sequential order, bounded above by 4.

Lemma 2 $[\forall\ \text{B} : 1 \leq \text{B} \leq 4 :: \text{step(B)}\ \textbf{unless}\ \text{step(B + 1)}] \wedge \textbf{stable}\ \text{step(4)}$

> To prove the **unless** relation holds, we need only to show that all transactions either maintain the LHS of the relation or establish the RHS. All *packer* and *creator* transactions maintain the LHS since they do not affect the tuple **step(B)**. If a *switcher* transaction changes the value of B, it does so by incrementing it by 1, thus establishing the RHS. There are no transactions that modify the tuple **step(4)**, proving the **stable** property. **End of Lemma 2.**

Using this lemma, we know that *term(4)* must be the last executing task. The progress property in (24) concludes that the first executing task leads to the last executing task. This property is an integral part of the proof for termination. We can divide the remainder of the proof into the following two statements.

$$\text{INIT} \longmapsto \mathit{init(1)} \tag{25}$$

$$\mathit{term(4)} \longmapsto \mathit{TERM}_{Bagger} \tag{26}$$

The proof of (25) is trivial from the definition of INIT in Section 5, i.e., we know

$$\text{INIT} \Rightarrow [\forall\ \text{I} : \text{item(I)} :: \text{unbagged(I)}] \tag{27}$$

and it is clear

$$[\forall\ \text{I} : \text{item(I)} :: \text{unbagged(I)}] \wedge \text{step(1)} \Rightarrow \mathit{init(1)} \tag{28}$$

From Section 9, we know

$$\mathit{term(4)} \Rightarrow [\forall\ \text{I} : \text{item(I)} :: \text{bagged(I)}] \tag{22}$$

To prove (26), we can use (22) and reduce the proof obligation to

$$[\forall\ \text{I} : \text{item(I)} :: \text{bagged(I)}] \longmapsto \mathit{TERM}_{Bagger} \tag{29}$$

To show (29), we use induction on the number of transactions left in the transaction space. We define

$$\text{rules-left}(\beta) \equiv ([\Sigma\ \text{t} : 1 \leq \text{t} \leq 13 \wedge \text{Rule(t)} :: 1] = \beta) \tag{30}$$

We know

$$[\forall\ \text{I} : \text{item(I)} :: \text{bagged(I)}] \wedge \text{rules-left(0)} \\ \longmapsto \\ TERM_{Bagger} \tag{31}$$

The proof obligation is then to show

$$[\forall\ \text{I} : \text{item(I)} :: \text{bagged(I)}] \wedge \text{rules-left}(\beta) \wedge \beta \geq 0 \\ \longmapsto \\ \text{rules-left(0)} \tag{32}$$

The property in (31) can be considered as the base case of the induction. The remainder of the proof of (32) is to show that the number of transactions in the transaction space decreases by 1, i.e., the induction step.

$$[\forall\ \text{I} : \text{item(I)} :: \text{bagged(I)}] \wedge \text{rules-left}(\beta) \\ \longmapsto \\ \text{rules-left}(\beta\ \text{-}\ 1) \tag{33}$$

Since Property 8 is a **stable** property, we do not have to state [∀ I : item(I) :: bagged(I)] in the RHS of the **leads-to** relation in (33). The proof is straightforward as it relies on the second subtransaction of each transaction from the translation described in Section 3. Because of this subtransaction, all transactions will establish the RHS. In other words, no matter which transaction is chosen for execution, it will not be reasserted into the transaction space all unbagged items are bagged and remain bagged (relying on Property 3).

Since at least one transaction establishes the RHS of (33), the **leads-to** relation holds proving (32). The proof of (32) establishes the proof of (29), which was necessary to conclude (26). Because both (25) and (26) hold, we can conclude that in Bagger, INIT $\longmapsto$ TERM.

11 Related Work

In this section we discuss independent research focusing on the formal verification of rule-based expert systems. The informal approaches referenced in Section 1 will not be presented. Though they are significant in the overall process of verification and validation of real-world expert systems, their philosophy is somewhat removed from the formal approaches.

11.1 Experimental Approaches to Development

In many applications, rules that emulate an expert's decision process can be easily generated to form a prototype system. These early versions of an expert system are often developed without the benefit of complete specifications. The specifications become more defined through repeated interaction between users, designers, and experts. Often multiple designers perform different work on the system not knowing fully what has been done by others.

The problems caused by this experimental development methodology relate directly to the difficulties of validation and verification. First, due to the absence of initial specifications, the system cannot be fully verified early in its development. Validation and verification during this time is done by letting the expert see what the system outputs. Second, if different people write different portions of the system, one designer is often unaware of rule dependencies used by another and may inadvertently destroy those dependencies or create redundant rules or rules that yield conflicting information. Finally, because there is no incremental validation and verification, if is difficult to subject a large cumbersome system to stringent testing, be it of a formal or informal nature.

We have taken the position that systems developed using an experimental methodology are likely to be less reliable than systems developed incrementally with validation and verification applied at each step. The reasoning is that if, after the fact, informal methods of validation and verification are used, then reliability can not be fully guaranteed, and if formal methods are used, proofs will be extremely cumbersome. Also, the systematic application of verification techniques during program development often provides valuable insight into the nature of the computation and the required rules.

It is highly unlikely that some minimal characterization of the input and output of an expert system cannot be given for initial development. Specifications centered around these characterizations can be formulated. For example, termination may be one of these initial specifications. Then, prior to the system being evaluated by the expert, the program can be shown to meet the specifications. As the designers build upon the system by adding rules and incorporating new data structures, they should ensure that the changes do not violate the current specifications. As the specifications become more defined, the program will have to reevaluated the rules in order to guarantee they maintain the new specifications. We contend that it is this type of methodology that must be used for those rule-based systems from which reliability is demanded.

11.2 Swarm Program Development

Swarm was initially developed to serve as a vehicle for investigating the shared dataspace approach to concurrent computation [25]. This research was motivated by the view that the content-addressable approach seemed to encourage higher degrees of concurrency and more flexible connections of components, but verification techniques for such languages were not yet developed. The results of the investigation up to this point show that Swarm has the ability to bring a variety of programming paradigms such as shared variable and message passing under a single unified model. It was recognized early on that because of its content-addressable nature and the use of rules to transform program states, verification results could be transferred to rule-based languages within the Swarm framework [7, 25]. These ideas later became more concrete when it was shown that the class of sequential pure rule-based programs were a proper subset of the class of Swarm programs [10]. Because of these properties, Swarm has been a useful vehicle for this investigation.

In the following sections we present two other independently developed approaches to formal verification, one that focuses on safety properties alone and one that transforms a rule-based program into a guarded iteration. The advantages of Swarm are its generalits and its proof logic that eliminates the need to consider the history of rule executions. Though in the previous sections we have presented some complex proofs, they have relatively the same complexity as similar proofs in the alternative approaches to be presented.

Because there are other resources in Swarm not yet tapped by the rule-based programming paradigm, there is still more investigation needed. For example, Swarm can be used to specify synchronous and asynchronous processing modes, and can accommodate highly dynamic program and data structures, possibilities not fully explored in rule-based programming. Swarm also provides an environment for investigating new methodologies for developing rule-based programs. Our research into the verification of rule-based programs stems from the ease of using Swarm and applying its proof theory, and the desire to investigate further how concurrency can be exploited in these programs.

11.3 The Use of Invariants

Rushby and Whitehurst [27] cite several common reasons why they believe that expert systems are not amenable to formal verification: the specifications are normally vague; the current development methodology (as discussed in Section 11.1) is experimental in nature; conventional verification is difficult to apply because it is concerned with algorithmic software; the current rule-based languages have not been developed with formal analysis in mind. Their paper addresses each of these reasons by giving an example of how invariants can be used for verification in certain classifications of expert system programs. They use their own rule-based language to facilitate the formulation of the program, combined with an automated theorem prover to verify the invariants, once they are defined.

They rely on the axiomatic rule for an invariant over an **if-then** statement. Using only pure rule-based system programs, they attempt to prove the invariant for every rule in the system, and guarantee that it is true for valid inputs. The invariants defined are those that verify the *minimum competency requirements* of a program. These requirements differ from *service requirements*, which are assumed to be easily subjected to formal verification as in conventional software. Competency requirements represent the vague requirements of a expert system that relate to how the expert quantifies output. Competency requirements are further divided into those that are *desired*, such as wanting an optimal solution, and those that are *minimum*, such as a bound on how bad the results can be before the system is considered unreliable.

To utilize and express minimum competency requirements as safety properties in a rule-based expert system program, they first classify the application areas where these programs may be useful into two categories, *bounded constraint satisfaction* and *model inversion*. To solve a bounded constraint satisfaction problem, it is necessary to find an assignment of values to the variables that satisfy the bounded constraint defines how far an acceptable solution can be from optimal. They believe that expert systems concerned with applications such as planning, scheduling, configuration and design can be formulated as bounded constraint satisfaction problems in which the invariant specifies the bounded constraint on the output. The model inversion problem is described as finding the causes (identified as the inputs to the system) that explain observed effects (the outputs of the system). If heuristics are used to suggest valid input values, Rushby and Whitehurst believe that expert systems concerned with applications such as monitoring equipment and fault diagnosis can be characterized as model inversion problems, where the model that is inverted must constructed explicitly to produce the specifications. The invariant for this problem specifies that the "effect" from certain "causes" must be supported by the explicit model that is created, in which it is stated that these "causes" produce the "effect."

They recognize that due to the restricted formulation of constraint satisfaction and model inversion problems, general expert systems exhibit better performance. They contend that the use of constraint satisfaction and model inversion problem formulations are necessary if reliable expert systems are to be developed.

11.4 Reduction to Guarded Iteration

Waldinger and Stickel [32] do not advocate a change in the current development methodology of rule-based expert systems. Their goal is to subject final programs, in the form of pure rule-based programs, to formal verification using an automated theorem prover. A set of criteria is given to define the **correct behavior** of a rule-based program, i.e., that the program obeys a given set of specifications. Each of these criteria is called a **validation task.** For every positive validation task, there is a negative counterpart.

Verification:	A property is proven to be always satisfied.
Fault detection:	Exhibiting an input that causes a system to fail to satisfy a given condition.
Termination:	Proving that a system will always terminate.
Loop detection:	Exhibiting an input that will cause a system to fail to terminate.
Firing:	Exhibiting an input that causes a given rule to fire.
Unreachability:	Proving that no input will cause a given rule to fire.
Consistency:	Proving that no input can produce an inconsistent working memory.
Inconsistency:	Exhibiting an input that will produce an inconsistent working memory.

They use a deductive approach to performing each of the validation tasks defined above. A conjecture is associated with each validation task. If the validity of the conjecture can be established in the system theory, then the associated validation task is performed. To prove their conjectures Waldinger and Stickel [32], use a model and notation similar to OPS5, but without conflict resolution. For a given system of rules, a **system theory** is developed. The system theory is defined by a set of axioms that express the actual behavior of the system. As in Swarm, the approach taken by Waldinger and Stickel treats working memory as a variable that is changed as rules are executed. The execution of the inference mechanism is reduced to an iteration of guarded commands [8]. Specifically, a **rule system** is defined as an unordered set of rules. A rule system is applied to a working memory by repeatedly applying any of the rules (selected nondeterministically) to working memory until no rule can be applied, i.e. satisfied. The final working memory is the result of applying the system.

An axiom based on Dijkstra's weakest precondition [8] is formulated for each rule in the rule system, where each predicate symbol in a rule is represented by a function symbol in the system theory. A function is defined that returns the result of working memory after a rule is applied. The entire system is applicable to a working memory if some rule in the system with some instantiation is applicable. A working memory to which no rule is applicable is called the *final working memory.* A *history* is defined as a description of a finite initial segment of a possible computation of the system. A history is applicable to a working memory if there is a finite sequence of working memories such that each memory is obtained from the previous one by applying the corresponding rule from the history. A history is called a *terminating history* if applied to the initial working memory results in the final working memory. The axioms for defining the rules that form a history, defining the working memory that results by applying a history, and defining a terminating history are all biased toward the use of an automated theorem prover.

One of their goals for this research is to go beyond the simple yes/no answer to whether or not a conjecture can be established. They wish to identify the faults of the system by extracting information out of the proof process. This may be done if the proof is restricted to be sufficiently constructive, allowing the extraction of a description of a case in which the condition to be proven fails to hold; basically, an example is constructed. To prove the negative validation tasks, they either attempt to prove that the positive condition does not hold, or find an example. Not all validation conjectures put forward can be satisfied. For example, they have not yet experimented with proving nontermination.

12 Discussion and Conclusion

This paper presents an assertional approach to the verification of a class of rule-based programs characterized by the absence of conflict resolution. The verification method is borrowed directly from work in concurrent programming. We show how program specifications are expressed as safety and progress properties. These properties are then classified according to their relation to the original problem specification and to the actual program. A toy example program illustrates the use of safety and progress properties to express the desired requirements to be met by the program. Proofs of certain properties give the reader an idea of how verification is accomplished. Some of these proofs show the complexity involved in formally verifying a rule-based program.

One purpose of this paper is to take an already formulated program, Bagger [33], show that it can be encoded in the formal Swarm language, that its specifications can be expressed as safety and progress properties, and that the Swarm proof theory can be applied to this domain by showing Bagger satisfies its specifications. From this exercise, another purpose emerged; showing that a new development methodology is necessary to incorporate formal verification.

We recognize that development techniques for experts systems cannot change overnight, but changes will have to be made if reliable expert systems are to become commonplace. If we were to develop Bagger from its specification and incrementally apply verification techniques, immediate improvements would be made to the program. First, using a single task per item type makes the tasks more regular, i.e., disallowing one task to bag frozen and non-frozen items of the same size. This improvement reduces the complexity of the proof of Property 14, in which two functions must be used to relate the task to the value of the step tuple and to relate the item type to its bag ordering. By making each task responsible for one item type, only one function is needed. Also, it is feasible to create a bag and place something in that bag during the same rule execution. This change is also suggested by the proof of Property 14, in which an otherwise unnecessary predicate 'fits' must be used. Another improvement would be to utilize a single context switching rule. Each of these improvements are due to the insight provided by the proof. After verifying the program, if we were to add rules or entire tasks, these changes would have to be incorporated into the proofs. But such incremental changes are not nearly as complex as proving the properties after the program is assumed to be complete. Also, with incremental development and verification, error in the rules or working memory can be detected and corrected more easily.

The original Bagger program was geared toward the use of conflict resolution. A task can easily bag frozen and non-frozen items of the same size by using the specificity rule. Then there would be no need for extra context switching, as there would be in our changes above. This brings up several important points. First, how important is the role of conflict resolution in rule-based expert systems? What do we lose by eliminating the use of conflict resolution for the sake of achieving reliability through the application of formal verification methods? Finally, can we extend program verification techniques to cover those forms of conflict resolution that appear to be essential to rule-based programming? As seen in Section 11, it is generally acceptable to use programs that do not rely on conflict resolution for the purpose of formal verification. Rushby and Whitehurst [27] believe a pure rule-based program gives only the approximate semantics, but can still be used effectively. Waldinger and Stickel [32] believe that only certain types of conflict resolution interfere with correctness, and those can be simulated explicitly in the LHS of the rules. We believe that conflict resolution has merit in sequential rule-based expert systems in both the execution efficiency and the development process. Therefore, we are analyzing the goals that conflict resolution meets in order to better answer the previous questions. Initially we are looking for ways to incorporate the conflict resolution strategies already within a rule-based program to be proven reliable by: (1) formalizing translation rules to transform a program that relies on conflict resolution into a pure rule-based program, and (2) translating the program and its conflict resolution strategy into Swarm, and formalizing the strategy in Swarm, thus eliminating the need for the original program to be a pure rule-based program.

Execution speed is another problem plaguing rule-based expert systems. Slow speed bars such systems from being utilized in many domains. Large programs are too slow to be useful in most real-world applications. Work is ongoing to use parallel computation as a possible scalable solution to the problem. This solution is plausible because most expert systems are complex enough to have tasks which can execute independently from one another, as well as rules which can execute in parallel. For example Gupta [14] and Miranker [20] have developed parallel matching strategies to speed up one portion of the execution process. Others, such as Ishida and Stolfo [15], Schmolze [28], Pasik [23] and Kuo [16] have attempted to produce speed up by parallel rule executions during another portion of the execution process of a rule-based system, but it is difficult to extract parallelism when there is none. Such systems are concerned with the parallel implementation of existing sequential systems.

By contrast, our research is directed toward the formal derivation of rule-based programs that exploit concurrency [26]. Ongoing work is based on deriving programs from their specifications with an eye on concurrency. The derivation includes guaranteeing the program obeys the specifications even as they are updated during the derivation. Heuristic derivation techniques borrowed from research in program derivation will be applied to this domain. Problems such as minimizing contention and rule interference will be addressed, along with the role of conflict resolution techniques in concurrent programs. Our hope is to derive highly concurrent reliable rule-based programs capable of being executed on a variety of parallel architectures.

References

[1] J.P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15, 1990.

[2] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, 1989.

[3] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Reading, MA, 1988.

[4] C. Chang, R. Stachowitz, and J. Combs. Testing integrated knowledge-based systems. In *IEEE International Workshop on Tools for AI*, October 1989.

[5] C.L. Chang and R.A. Stachowitz. Testing expert systems. In *Proceedings of Space Operations Automation and Robotics Workshop (SOAR)*, July 1988.

[6] H.C. Cunningham and G.-C. Roman. Toward formal verification of rule-based systems: A shared dataspace perspective. Technical Report WUCS-89-28, Washington University, June 1989.

[7] H.C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.

[8] E.W. Dijkstra. *A Disipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[9] C.L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[10] R.F. Gamble. Transforming rule-based programs: from the sequential to the parallel. In *3rd International Conference on Industrial and Engineering Applications of Artificial Intelligence and Systems*, New York, NY, July 1990. ACM Press.

[11] R.F. Gamble and W.E. Ball. An approach to verifying rule-based programs. In *Proceedings of the 3rd Midwest Artificial Intelligence and Cognitive Science Society Conference*, 1991.

[12] R.F. Gamble, G.-C. Roman, and W.E. Ball. Formal verification of rule-based programs. In *Proceedings of the 9th National Conference on Artificial Intelligence*, Menlo Park, CA, July 1991. AAAI Press. Also Technical Report WUCS-91-16, Dept. of C.S., Washington University, St. Louis.

[13] R.F. Gamble, G.-C. Roman, and W.E. Ball. On extending the application of formal specification and verification methods to rule-based programming. Technical Report WUCS-91-1, Washington University, 1991.

[14] A. Gupta. *Parallelism in Production Systems.* Pitman Publishing, London, England, 1987.

[15] T. Ishida and S.J. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of the IEEE International Conference on Parallel Processing*, Washington, D.C., 1985. IEEE Computer Society Press.

[16] S. Kuo, D. Moldovan, and S. Cha. Control in production systems with multiple rule firings. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 247–251, 1990.

[17] R. Loganantharaj. Static analysis of consistency and redundancy of a rule based expert system. In *2nd Int'l Conference on Industrial Applications of AI and Expert Systems*, June 1989.

[18] S. Matuoka and S. Kawai. Using tuple space communication in distribued object-oriented languages. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, 1988. *ACM SIGPLAN Notices*, Vol. 23, No. 11, pp 276-284.

[19] J. McGuire. Uncovering redundancy and rule inconsistency in knowledge bases via deduction. Technical Report A-45, Lockheed Missiles and Space Company, Inc, 1989.

[20] D.P. Miranker. TREAT: a better match algorithm for AI production systems. In *Proceedings of the Sixth national Conference on Artificial Intelligence (AAAI-87)*, pages 42–47, July 1987.

[21] T.A. Nguyen, W.A. Perkins, T.J. Laffey, and D. Pecora. Checking an expert systems knowledge base for consistency and completeness. In *9th Int'l Joint Conference on Artificial Intelligence*, 1985.

[22] T.J. O'Leary, M. Goul, K.E. Moffitt, and A.E. Radwan. Validating expert systems. *IEEE Expert*, pages 51–58, June 1990.

[23] A. Pasik. A methodology for programming production systems and its implementations on parallelism. Technical Report Technical report and Ph.D dissertation, Columbia University, 1988.

[24] M. Rem. Associons: A program notation with tuples instead of variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, July 1981.

[25] G.-C. Roman and H.C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, December 1990.

[26] G.-C. Roman, R.F. Gamble, and W.E. Ball. Seeking concurrency in rule-based programming. Technical Report WUCS-91-17, Dept. of C.S., Washington University, St. Louis, January 1991.

[27] J. Rushby and R.A. Whitehurst. Formal verification of AI software. Technical report, SRI International, Computer Science Laboratory, February 1989.

[28] J.G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *8th National Conference on Artificial Intelligence*, pages 65–71, Cambridge, Mass., 1990. MIT Press.

[29] R. Stachowitz and J. Combs. Validation of expert systems. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 686–695, January 1987.

[30] R. Stachowitz and J. Combs. Completeness checking of expert systems. Technical Report A-60, Lockheed Missiles and Space Company, Inc., 1990.

[31] M. Suwa, A.C. Scott, and E.H. Shortliffe. Completeness and consistency in a rule-based system. In B. G. Buchanan and E. H. Shortliffe, editors, *Rule-Based Expert Systems*, pages 159–170. Addison-Wesley Publishing Co., Reading, MA, 1984.

[32] R.J. Waldinger and M.E. Stickel. Proving properties of rule-based systems. In *Proceedings of the 7th IEEE Conference on Artificial Intelligence Applications*, February 1991.

[33] P.H. Winston. *Artificial Intelligence, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Mass., 1984.

A More Proofs of Swarm Bagger

This appendix contains the proofs of those properties stated earlier in the paper, but not proven. The properties retain the same numbering and categorization as in the previous sections.

A.1 Structural Constraints

Property 5: inv $[\Sigma\ N : \text{current}(N) :: 1] = 1$

There is exactly one current tuple always present in working memory.

The invariant holds initially by definition of INIT. Only a *creator* transaction can violate the invariant. In these transactions, the tuple **current(N)** for some N is deleted if and only if **current(N+1)** is reasserted into the tuple space. Thus, the invariant is preserved.

Property 6: inv $[\forall\ N :: [\Sigma\ W,A : \text{bag}(N,W,A) :: 1] \leq 1]$

The bags are identified by a unique natural number.

The invariant holds initially because there are no bag tuples. The invariant can be violated if a transaction asserts a **bag(N,W',A')** for some N,W' and A', when **bag(N,W,A)** exists. *Packer* transactions delete **bag(N,W,A)** if and only if **bag(N,W',A')** is asserted, and hence the invariant is preserved. *Creator* transactions assert **bag(N,0,<>)** and **current(N)**. The value of N in **current(N)** is incremented after each successful execution of a *creator* transaction. To show the invariant holds it must be shown that N is a unique identification number when **bag(N,0,<>)** is asserted. This is done by proving the following lemma.

Lemma 3 *The value of N, in the tuple current(N), is non-decreasing.*

current(N) **unless** current(N + 1) for all N

> *Packer* and *creator* transactions maintain the LHS of the **unless** and *switcher* transactions either maintain the LHS or establish the RHS. **End of Lemma 4.**

Since the value of N in **current(N)** does not decrease, but increases only when a new bag tuple is to be asserted, the identifications numbers are unique.

Property 7: inv $[\forall\ N,W,A :: \text{bag}(N,W,A) \Rightarrow [\forall\ M : 1 \leq M \leq N :: [\exists\ W',A' :: \text{bag}(M,W',A')]]]$

The bags are ordered sequentially, beginning with the number 1.

Initially there are no tuples of type **bag(N,W,A)** for any N,W and A, and **current(0)** exists in the tuple space.

Lemma 4 *The value of N in the tuple* **current(N)** *dictates the identification numbers of bags.*

$$[\forall\ N,M,W,A : \text{current}(N) \Leftrightarrow \text{bag}(M,W,A) \wedge M \leq N]$$

New tuples of type **bag(M,W,A)** for some M,W,A can only be asserted by *switcher* transactions. A *switcher* transaction asserts **bag(N,0,<>)** if and only if it asserts **current(N)**. There is only one **current(N)** tuple as stated in Property 5. By lemma 3, the value of N in **current(N)** is non-decreasing and when it does increase, it does so sequentially. Thus, all bags created must have an identification number of at most N for **current(N)**.

Using lemma 4, we know that the value of N in the tuple **current(N)** dictates the bag identification number. If a *creator* transaction with precondition **current(N)** is successful, it asserts **bag(N+1,0,<>)** and **current(N+1)**, causing the numbers to be in sequential order.

A.2 Behavioral Constraints

Property 8: **inv** [∀ I,N,n :: **stable** placed-in(I,N,n)]
A bagged item remains bagged in the same bag and in the same position in that bag.

where placed-in(I,N,n) is defined in Section 6, under the description of Property 3.

Given I,N,n. Assume [∃ W,A :: bag(N,W,A) ∧ A.n = I], there are 3 parts to show for the proof: (i) no bag tuples are deleted from the tuple space, (ii) the contents of a bag are not deleted, and (iii) items do not change positions with in a bag. Lemmas 1, 2, and 3 provide the proofs to each part respectively.

Lemma 5 *Once a bag exists, it continues to exist.*

$$[\forall\ N :: \textbf{stable}\ [\exists\ W,A :: \text{bag}(N,W,A)]]$$

Assume for some N,W and A, **bag(N,W,A)** exists. A *creator* transaction deletes **bag(N,W,A)** if and only if for some W' and A', **bag(N,W',A')** is asserted, where W $\neq$ W' and A $\neq$ A'. No other transactions modify a tuple of type **bag(N,W,A)**. Thus, the lemma holds. **End of Lemma 5.**

Lemma 6 *The contents of a bag are never deleted.*

```
[∀ N,α ::
    [∃ W,A : bag(N,W,A) :: A = α]
    unless
    [∃ W,A,I : item(I) ∧ bag(N,W,A) :: A = α • I] ]
```

All *creator* and *switcher* transactions maintain the LHS of the lemma. *Packer* transactions either maintain the LHS or establish the RHS. **End of Lemma 6.**

Lemma 7 *The positions of the items in a bag are never changed.*

[∀ N,α :: **stable** [∃ W,A : bag(N,W,A) :: α is a prefix of A]]

Only *packer* transactions alter the contents of A by adding an item to A, but they do not change α. **End of Lemma 7.**

A.3 Output Constraints

Property 11: Invariant $[\forall$ N,W,A : bag(N,W,A) :: W $\leq$ *maxwgt*$]$
At any time, the weight of every bag cannot exceed the maximum weight allowed.

Initially there exists no N,W and A such that **bag(N,W,A)** is in the tuple space. Possible violating transactions could be either *packer* or *switcher* transactions. Given the existence of **bag(N,W,A)** for some N,W and A, such that W $\leq$ *maxwgt*. Given I, such that **unbagged(I)**, and A.n $\neq$ I for $1 \leq$ n $\leq |$A$|$. Assume W + weight(I) $>$ *maxwgt*. For

$$\mathbf{bag(N,W + weight(I), A \bullet I)}$$

to be asserted into the tuple space, it must be true that W $\leq$ *maxwgt* - weight(I). Therefore, I would not be added to **bag(N,W,A)** by a *packer* transaction. *Switcher* transactions assert **bag(N,0,<>)** for some N. Therefore, the invariant is maintained.

Property 12: inv $[\forall$ N,N',W,W',A,A' : bag(N,W,A) $\wedge$ bag(N',W',A') $\wedge$ N' $<$ N $\wedge$ A $\neq$ <>
:: W' + weight(A.1) $>$ *maxwgt*$]$
The first item in bag N, does not fit in any bag M < N.

where weight(I) is defined as follows.

Definition 5 *The weight of an item.*

$[\forall$ I : item(I) :: weight(I) = $[\Sigma$ C,W,F : grocery(I,C,W,F) :: W$]$ $]$

Switcher transaction have no effect on bag tuples. *Creator* transactions cannot violate the invariant because they create only empty bags. Only *packer* transactions can violate the invariant. We assume the invariant holds prior to the execution of a *packer* transaction. Given **unbagged(I)** and **bag(N,W,A)** exist in the tuple space, where N = [max X,Y,Z : bag(X,Y,Z) :: X], there are three computation states that can occur prior to the execution of a *packer* transaction. We rely on Properties 6, 7, and 11.

Case (i):
If A $\neq$ <> and
$[\exists$ T,U,V : bag(T,U,V) $\wedge$ T $<$ N :: U + weight(I) $\leq$ *maxwgt*$]$
then the invariant is not violated because a successful *packer* transaction cannot place I in the first position of any bag.

Case(ii):
If A $\neq$ <> and
$[\forall$ T,U,V : bag(T,U,V) :: U + weight(I) $>$ *maxwgt*$]$
then no *packer* transaction is successful and the invariant is not violated.

Case(iii):
If A = <> and
$[\exists$ X,Y,Z : bag(X,Y,Z) $\wedge$ X $<$ N :: Y + weight(I) $\leq$ *maxwgt*$]$
then a *packer* transaction *can* violate the invariant by adding I to A when there exists a bag with room for I.

To show the invariant is not violated, we must show case (iii) cannot occur. We utilize the definitions of min-f, min-f-item, and min-f-bagger from the proof of Property 10 in Section 8. We know from this proof that a successful *packer* transaction that places an item in a bag is min-f-bagger, and this transaction packs min-f-item. Thus, the proof that case (iii) cannot occur is formulated as the following proof obligation.

$$\begin{aligned}&\textbf{invariant } [\forall\ \mathrm{N} :: \mathrm{bag(N,0,<>)} \Rightarrow \\ &\qquad [\forall\ \mathrm{M,W,A} : \mathrm{bag(M,W,A)} \wedge \mathrm{M} < \mathrm{N} \\ &\qquad\quad :: \mathrm{W} + \mathrm{weight(min\text{-}f\text{-}item)} > \mathit{maxwgt}]]\end{aligned} \tag{12.1}$$

Initially, there are no bag tuples so the invariant holds. To prove the invariant is preserved, we have to show that those transactions that (1) make **bag(N,0,<>)** true, (2) cause W + weight(min-f-item) to be less than *maxwgt*, and (3) cause min-f-item to be undefined, do not violate the invariant.

Creator transactions can cause **bag(N,0,<>)** to become true. The precondition of all *creator* transactions can be stated more generally as

$$\mathrm{step(B)} \wedge \mathrm{g(f(I))} = \mathrm{B} \wedge [\forall\ \mathrm{M,W,A} : \mathrm{bag(M,W,A)} :: \mathrm{W} + \mathrm{weight(I)} > \mathit{maxwgt}] \wedge \mathrm{current(N)}$$

Therefore, if a *creator* transaction is successful causing **bag(N,0,<>)** to be true, the RHS of the implication in (12.1) is also true.

Packer transactions may cause the RHS of the implication in (12.1) to become false if (a) the weight of any **bag(M,W,A)** with M < N decreases, (b) the weight of min-f-item decreases such that the weight of **bag(M,W,A)** where M < N can hold min-f-item or (c) min-f-item becomes undefined. We assume the invariant holds before the execution of a *packer* transaction (which must be min-f-bagger as shown earlier).

For case (a), there are no transactions that decrease the weight of a bag. For case (b), no transaction decreases the weight of an item directly. In order for the weight of min-f-item to decrease, the program state prior to the execution of min-f-bagger must include

$$\mathrm{I} = \mathrm{min\text{-}f\text{-}item} \wedge [\Sigma\ \mathrm{J} : \mathrm{unbagged(J)} \wedge \mathrm{f(J)} = \mathrm{min\text{-}f} :: 1] = 1$$

After successful execution of min-f-bagger, bagged(I) is true and weight(min-f-item) < weight(I). But, since the invariant is also a precondition of the transaction that bags I (previously min-f-bagger), "bag N" was the only bag that could hold I. Thus, I can only be placed in "bag N", maintaining the invariant.

For case (c), a *packer* transaction cannot be successful unless there is a tuple of type **unbagged(I)** for some I in the tuple space. If min-f-bagger is successful and packs the final unbagged item, we have shown the invariant to hold because the LHS is false. A *creator* transaction cannot make **bag(N,0,<>)** true unless there is an unbagged item left, thus it also does not violate the invariant.

Since the invariant of (12.1) is preserved, we have proven case (iii) cannot occur, which shows Property 12 is also preserved.

A.4 Output Generation

Property 14: $[\forall\ \mathrm{N} : \mathrm{current(N)} \wedge [\exists\ \mathrm{I} : \mathrm{unbagged(I)} :: [\forall\ \mathrm{M,W,A} : \mathrm{bag(M,W,A)} \wedge \mathrm{M} \leq \mathrm{N} :: \mathrm{weight(I)} + \mathrm{W} > \mathit{maxwgt}]\ ] \longmapsto \mathrm{current(N+1)} \wedge \mathrm{bag(N+1,0,<>)}]$

Bags are created when needed.

Using the definition of min-f and min-f-item from the proof of Property 10, we know that

$$[\forall\ \mathrm{I} : \mathrm{unbagged(I)} :: \mathrm{weight(min\text{-}f\text{-}item)} \geq \mathrm{weight(I)}]$$

Then we only have to prove Property 14 when min-f-item doesn't fit in any bag, We know from Property 5 that there is only one tuple of type **current(N)** for all N in the tuple space at all times and that from lemma 4

$$[\forall\ \mathrm{N,M,W,A} :: \mathrm{current(N)} \Rightarrow \mathrm{bag(M,W,A)} \wedge \mathrm{M} \leq \mathrm{N}]$$

From these properties, the proof obligation can be restated as

$$\begin{array}{l} \mathrm{current(N)} \wedge [\exists\ \mathrm{I} : \mathrm{unbagged(I)} \\ \qquad :: [\forall\ \mathrm{M,W,A} : \mathrm{bag(M,W,A)} \\ \qquad\qquad :: \mathrm{weight(min\text{-}f\text{-}item)} + \mathrm{W} > \mathit{maxwgt}]] \\ \longmapsto \\ \mathrm{current(N+1)} \wedge \mathrm{bag(N+1,0,<>)} \end{array} \tag{14.1}$$

for all N. We rely on Lemma 1 in the proof of Property 10 and Lemma 2 in the proof of Property 16 and break the proof into two steps. First we show that for all N and $1 \leq \mathrm{B} \leq 4$

$$\begin{array}{l} \mathrm{current(N)} \wedge [\exists\ \mathrm{I} : \mathrm{unbagged(I)} \\ \qquad :: [\forall\ \mathrm{M,W,A} : \mathrm{bag(M,W,A)} \\ \qquad\qquad :: \mathrm{weight(min\text{-}f\text{-}item)} + \mathrm{W} > \mathit{maxwgt}] \\ \qquad \wedge\ \mathrm{step(B)} \wedge \mathrm{B} \leq \mathrm{g(min\text{-}f)} \\ \longmapsto \\ \mathrm{step(B)} \wedge \mathrm{B} = \mathrm{g(min\text{-}f)} \end{array} \tag{14.2}$$

and then we show that

$$\begin{array}{l} \mathrm{current(N)} \wedge [\exists\ \mathrm{I} : \mathrm{unbagged(I)} \\ \qquad :: [\forall\ \mathrm{M,W,A} : \mathrm{bag(M,W,A)} \\ \qquad\qquad :: \mathrm{weight(min\text{-}f\text{-}item)} + \mathrm{W} > \mathit{maxwgt}] \\ \qquad \wedge\ \mathrm{step(B)} \wedge \mathrm{B} = \mathrm{g(min\text{-}f)} \\ \longmapsto \\ \mathrm{current(N+1)} \wedge \mathrm{bag(N+1,0,<>)} \end{array} \tag{14.3}$$

To show (14.2), we must show that in this case the value of B in **step(B)** increases by one until B = 4. Then, since $\mathrm{g(min\text{-}f)} \leq 4$ by definition, B will eventually equal 4.

$$\begin{array}{l} \text{current(N)} \wedge [\exists \text{ I : unbagged(I)} \\ \quad\quad :: [\forall \text{ M,W,A : bag(M,W,A)} \\ \quad\quad\quad\quad :: \text{weight(min-f-item)} + \text{W} > \textit{maxwgt}] \\ \quad \wedge \text{ step(B)} \wedge \text{B} \leq \text{g(min-f)} \\ \textbf{ensures} \\ \text{step(B+1)} \end{array} \quad (14.4)$$

Both *packer* and *creator* transactions maintain the LHS of the **ensures**, and *switcher* transactions either maintain the LHS or establish the RHS, depending on the value of g(min-f). Thus, the *unless* part of the proof holds. If B < g(min-f) then by definition of min-f we have

$$[\forall \text{ I : g(I)} = \text{B} :: \neg\text{unbagged(I)}]$$

and we know $1 \leq B \leq 3$ by INIT and earlier invariants. There exists a *switcher* transaction for **step(B)** such that $1 \leq B \leq 3$. Thus, at least one *switcher* transaction will be successful on the next execution step, establishing the RHS of (14.4). Using the transitive property of **leads-to** and (14.4), we establish (14.2).

The next step of the proof is to show (14.3) holds. This property can be restated as follows.

$$\begin{array}{l} \text{current(N)} \wedge \neg\text{fits(min-f-item,B)} \\ \textbf{ensures} \\ \text{current(N+1)} \wedge \text{bag(N+1,0,<>)} \end{array} \quad (14.5)$$

Packer and *switcher* transactions maintain the LHS of the **ensures**. Depending on the value of B, a *creator* transaction will either maintain the LHS or establish the RHS of (14.5), and at least one *creator* will establish the RHS on the next execution step.

Property 15: $[\forall \text{ N} :: \text{bag(N,0,<>)} \longmapsto [\exists \text{ W,A : bag(N,W,A)} :: \text{W} > 0 \wedge \text{A} \neq \text{<>}]\ ]$
Eventually every bag has at least one item.

We know by (12.1) in the proof of Property 12 that

$$\begin{array}{l} \textbf{invariant } [\forall \text{ N} :: \text{bag(N,0,<>)} \Rightarrow \\ \quad\quad [\forall \text{ M,W,A : bag(M,W,A)} \wedge \text{M} < \text{N} \\ \quad\quad\quad :: \text{W} + \text{weight(min-f-item)} > \textit{maxwgt}]] \end{array} \quad (12.1)$$

We need another invariant given in the following lemma.

Lemma 8 *The identification number of an empty bag equals the value of N in current(N).*

$$\textbf{invariant } [\forall \text{ N} :: \text{bag(N,0,<>)} \Rightarrow \text{current(N)}]$$

A *creator* transaction is the only transaction that can violate the invariant. But this transaction, if successful, makes bag(N,0,<>) true if and only if **current(N)** is made true, for some N. thus, the invariant is preserved. **End of Lemma 8.**

Using (12.1) and lemmas 4 and 8, we can restate Property 15 as the following **ensures** relation.

$$\begin{array}{l} \text{bag(N,0,<>)} \wedge \text{current(N)} \\ \qquad\qquad \wedge\ [\forall\ \text{M,W,A : bag(M,W,A)} \wedge \text{M} \leq \text{N} :: \text{W} + \text{weight(min-f-item)} > \mathit{maxwgt}] \\ \textbf{ensures} \\ [\exists\ \text{W',A' : bag(N,W',A')} :: \text{W} > 0 \wedge \text{A} \neq \text{<>}] \end{array} \tag{15.1}$$

for all N. If a *switcher* transaction is successful, the LHS will be maintained. A *creator* transaction cannot be successful, so again the LHS of the **ensures** is maintained. All *packer* transactions except min-f-bagger also maintain the LHS. However, min-f-bagger establishes the RHS. Therefore, the **ensures** holds, proving Property 15.