Report Number: WUCS-90-13

1991-01-01

Transforming a Rule-based Program

Rose F. Gamble

Conflict resolution is a form of global control used in production systems to achieve an efficient sequential execution of a rule-based program. This type of control is not used to parallel production system models [6,13]. Instead, only those programs that make no assumptions regarding conflict resolution are executed in parallel. Therefore, the initial sequential rule-based programs are either executed in parallel without their conflict resolution strategy, which normally results in incorrect behavior, or the programs are transformed in an ad hoc manner to execute on an particular parallel production system model. As a result, these programs do not exhibit...
Read complete abstract on page 2.

Transforming a Rule-based Program

Rose F. Gamble

Complete Abstract:

Conflict resolution is a form of global control used in production systems to achieve an efficient sequential execution of a rule-based program. This type of control is not used to parallel production system models [6,13]. Instead, only those programs that make no assumptions regarding conflict resolution are executed in parallel. Therefore, the initial sequential rule-based programs are either executed in parallel without their conflict resolution strategy, which normally results in incorrect behavior, or the programs are transformed in an ad hoc manner to execute on an particular parallel production system model. As a result, these programs do not exhibit the parallelism hoped for [10,13]. We believe that a second reason behind the lack of parallelism is that no formal methods of verifying the correctness of rule-based programs are utilized. By correctness, we mean verifying the behavior of the program meets the specifications given. Correctness is especially important when conflict resolution is no longer utilized. It is necessary to transform sequential rule-based programs into equivalent programs without conflict resolution. Also, the parallel execution of a rule-based program is more complex and demands these formal methods even more than its sequential counterpart. In this paper, we present preliminary ideas for an approach to designing and developing correct rule-based programs for parallel execution. We investigate the difficulty in transforming a simple sequential rule-based program to a new version of the program with no conflict resolution. Also, we investigate the use of a new programming paradigm and language that may result in more efficient programs which provably correct, and can be executed in parallel [2].

Transforming a Rule-based Program

Rose F. Gamble

WUCS-90-13

January 1991

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

A portion of this paper appeared in the *Proceedings of the Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems,* July 1990.

Contents

Transforming A Rule-based Program*

Rose Fulcomer Gamble
Department of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130
July 30, 1990

Revised December 14, 1990

Abstract

Conflict resolution is a form of global control used in production systems to achieve an efficient sequential execution of a rule-based program. This type of control is not used in parallel production system models [6, 13]. Instead, only those programs that make no assumptions regarding conflict resolution are executed in parallel. Therefore, the initial sequential rule-based programs are either executed in parallel without their conflict resolution strategy, which normally results in incorrect behavior, or the programs are transformed in an ad hoc manner to execute on a particular parallel production system model. As a result, these programs do not exhibit the parallelism hoped for [10, 13].

We believe that a second reason behind the lack of parallelism is that no formal methods of verifying the correctness of rule-based programs are utilized. By correctness, we mean verifying the behavior of the program meets the specifications given. Correctness is especially important when conflict resolution is no longer utilized. It is necessary to transform sequential rule-based programs into equivalent programs without conflict resolution. Also, the parallel execution of a rule-based program is more complex and demands these formal methods even more than its sequential counterpart.

In this paper, we present preliminary ideas for an approach to designing and developing correct rule-based programs for parallel execution. We investigate the difficulty in transforming a simple sequential rule-based program to a new version of the program with no conflict resolution. Also, we investigate the use of a new programming paradigm and language that may result in more efficient programs which are provably correct, and can be executed in parallel [2].

*This research has been conducted within the Washington University Center for Intelligent Computer Systems (CICS), whose primary supporters are McDonnell Douglas Corporation and Southwestern Bell Corporation.

1 Introduction

Production systems, also called rule-based systems, are very useful in automating certain human expert tasks, but the current technology exhibits many problems. Sequential production systems are far too slow to be useful for any complex problem where resources, such as time, are limited. To solve the speed problem, attempts at executing production systems in parallel have been made. Some researchers, such as [5, 14], have concentrated on increasing the speed of the match phase of the execution. Others have concentrated on executing the remainder of the system in parallel. Research in this area has resulted in synchronous models[6, 12], and more asynchronous models[8, 9, 13]. Both the synchronous and asynchronous systems do not use global control in the form of conflict resolution, but require that programs be written without regard to any conflict resolution strategy that was utilized in the existing sequential version. These researchers either execute programs without conflict resolution, even though the programs rely on it, and get incorrect results, or they transform the program to one which does not rely on conflict resolution in an ad hoc manner so that it can execute in parallel. The way these programs are executed or transformed is believed to limit the parallelism exploited in rule-based programs [10, 13].

We believe there is a second reason for the difficulty. No formal language exists in which to express rule-based programs. Therefore, no verification techniques are utilized to prove properties which guarantee correct parallel execution [2]. If rule-based programs are to be executed with little or no human interaction, then the behavior of the program must be validated. Though much research has been done in the area of verification of sequential and concurrent programs, it has not yet been applied to rule-based programming. There are no design heuristics to aid a rule-based programmer in developing and proving the correctness of the programs. Designing a rule-based program for parallel execution demands such formal methods, due to the complexity of the execution, to ensure correct behavior.

1.1 Simple Production System

A simple production system consists of production memory (PM), which is defined by a set of rules, and working memory (WM), which is a database of assertions. A production rule has the following form:

$$\textbf{LHS} \longrightarrow \textbf{RHS},$$

where 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 working memory elements (WMEs). WMEs are data objects with class and attribute-value pairs. A positive CE is satisfied when there exists a matching WME. A negative CE is one that 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.

Execution of a serial production system consists of repeatedly executing a **match-select-act** cycle until a halt statement (a type of AE) is reached or the match phase returns the empty set. The match phase compares 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. Once all actions are performed the cycle begins again with the match phase.

1.2 Conflict Resolution

Production systems use conflict resolution strategies to control the search for a solution, and to achieve an efficient execution of a rule-based program. Most production systems have a conflict resolution strategy built in. Thus, programs encoded within a system make explicit use of its strategy. The strategies may come in different forms to accommodate any particular rule-based system design. In general, a conflict resolution strategy is made up of a combination of conflict resolution rules. McDermott and Forgy[7] detail thirteen of these rules, grouping them into five classes: 1) **production order**–based on a priority relationship defined among the rules, 2) **special case**–based on subsumption and redundancy of rules, 3) **recency**–based on the time of assertion for WMEs, 4) **distinctiveness**–based on history of rules fired, 5) **arbitrary decision**–based on random selection of a single instantiation. Some programmers use variations on these thirteen rules or detail another type of conflict resolution rule for a particular type of rule-based program.

For example, the LEX strategy in OPS5[4] is defined as the following combination of conflict resolution rules: **D2** $\rightarrow$ **R5** $\rightarrow$ **SC3** $\rightarrow$ **AD1**, where the arrows indicate the order in which the rules are applied to the conflict set. **D2**, from class 4 above, distinguishes if either the production or the data of the two instantiations are different. It uses the entire past history of firings, and prefers instantiations distinct from those fired. **R5**, from class 3, orders two instantiations by comparing their most recent WME. If equal, it compares next most recent, and so on. **SC3**, from class 2, determines if one instantiation is a special case of another instantiation. If the first instantiation contains as a proper subset all of the WMEs the second instantiation contains, and, if the production rule from the first has more CEs than the production rule from second, then the first instantiation is chosen. **AD1**, from class 5, randomly chooses a single instantiation. In this initial investigation, we concentrate on the use of the **SC3** conflict resolution rule.

1.3 Problems with Eliminating Conflict Resolution

To execute a production system in parallel and asynchronously, conflict resolution must be eliminated, simulated or encoded locally. These problems stem from the elimination of global control and from the attempt to execute a rule-based program in parallel.

We are concerned with three main problems that must be addressed when developing production system programs. Other problems that arise in parallel production system models will be briefly discussed in section 7. The first problem is **solution correctness**. In sequential systems, a global control mechanism, such as conflict resolution, is responsible for directing the search for a solution. If the search space is too complex, more control is utilized to prune it. Using this control, the solutions reached are acceptable. Guaranteeing a solution path will converge is difficult without global control, unless formal methods are utilized. It is also difficult to prove the solution reached is among the set of solutions deemed satisfactory. Formal methods require a formal representation of the program. Another problem is **termination detection**. Processors operating asynchronously

with only working memory as their medium may not have a way to determine when a solution is reached or when no solution can be found. Also, they may follow an incorrect solution path indefinitely. Finally the problem of **task-order dependence** arises when conflict resolution is eliminated and the rules of the program are executed asynchronously. Some tasks depend on the execution or non-execution of other tasks. With no global scheduler, such dependence must be encoded within the rules or working memory. Also, initial task ordering may be more the result of sequential programming than of necessity to achieve correct results. Our goal is to develop a methodology for designing rule-based programs that exhibit correct behavior without global control.

2 Bagger Problem

The Bagger problem [15] is a simple program to bag groceries. In the original OPS5 version of the program, each grocery class in WM has attributes: name, container, size, and frozen. The frozen attribute is a boolean and size is restricted to the values: small, medium and large. Each bag class in the original program has a weight and a contents. Each bag can hold a maximum weight of 15 pounds. Every large item weighs 6 lbs, every medium item weighs 4 lbs, and every small item weighs 2 lbs. Appendix A contains the full OPS5 version of Bagger.

The object of the program is as follows. Given a bag, large bottles must be placed in a bag first until no more can fit. If there are other large items, they should be placed next in the bag until no more can fit. Then medium items are placed in a bag until no more can be packed, with those that are frozen packed first. Finally small items are packed, with those that are frozen first, until the bag is as full as possible. If there are no items of some type, that stepped is skipped. There should be no extraneous bags, so they are created on an as needed basis.

Figure 1 displays the English rules for bagging medium items in Bagger, where **bmi** means "bag-medium-items".

There are four main tasks in Bagger. The first task is to bag large bottles before every other item. This task gives purpose to the container attribute. One of two assumptions is made. Either there are no bottles of other sizes or it is of no concern when bottles of other sizes are bagged among items of the same or different size. Of course many other rules and situations can be added to the program, but it exhibits many useful properties for research as it is.

The second task is to bag the rest of the large items, those that are not bottled. There is no other ordering for bagging these items. The next task is to bag medium items. This task distinguishes between frozen and non-frozen medium items, such that medium frozen items are bagged before medium non-frozen items. The last task is to bag small items, which are handled the same as medium items.

A *control variable* is used to separate the rules into tasks so that only those applicable to the current task enter the conflict set. The working memory element **step** is a control variable in this program. In Bagger, **step** clusters the rules into categories of bagging an item in a particular task. As the program executes, **step** will change to denote which type item is being bagged. Thus, there are four values for **step**: bag-large-bottles, bag-large-items, bag-medium-items, bag-small-items. The value of **step** is changed to the next task when its current task is complete. The program begins with **step** having the value bag-large-bottles.

In Bagger, each set of rules representing a task makes explicit use of the special case conflict resolution rule to choose the most specific rule of that task for execution. In particular, the **SC3**

Rule **bmi1**:

IF The step is bag-medium-items
AND there is a grocery item of medium size and frozen
AND there is a bag with weight $\leq$ 11
THEN
Put the medium frozen item in the bag
AND increase the weight of the bag by 4
AND remove the grocery item from working memory.

Rule **bmi2**:

IF The step is bag-medium-items
AND there is a grocery item of medium size
AND there is a bag with weight $\leq$ 11
THEN
Put the medium item in the bag
AND increase the weight of the bag by 4
AND remove the grocery item from working memory.

Rule **bmi3**:

IF The step is bag-medium-items
AND there is a grocery item of medium size
THEN
Create a new empty bag

Rule **bmi4**:

IF The step is bag-medium-items
THEN
change the step to bag-small-items

Figure 1: Rules for Bagging Medium Items

rule as described in Section 1.2 is used. Using Figure 1 as an example, given **step** is bag-medium-items, bmi2 will fire if there is any medium item to bag and an available bag in which to place it. The rule bmi1 is has more constraints that bmi2, since it only fires if there is an unbagged frozen medium item and an available bag. Thus, when bmi1 enters the conflict set, bmi2 will also. The **SC3** rule will always chose bmi1 over bmi2, because bmi1 is more specific than bmi2. This same rule will choose bmi1 over bmi3 and bmi4 which will also be in the conflict set whenever bmi1 is. **SC3** will choose bmi2 over bmi3 and bmi4, and it will chose bmi3 over bmi4. Therefore, an explicit ordering of the rules within a task is defined.

If this program were to execute on a parallel production system, the **SC3** rule would not be available to make its choice. In transforming Bagger to a program which does not rely on global control, we must determine the role this conflict resolution rule plays and whether or not it is necessary to emulate it for the program to execute correctly in parallel. If necessary, we must determine methods in which to emulate it and whether or not general methods exist or only domain specific ones.

Also, we must decide if the use of a control variable is needed. Due to each task's reliance on the **SC3** rule, each is sequential in nature. Only one rule will fire at a time during the execution of each task. Therefore, as the program is written, we would not achieve better results executing it in parallel if we emulate **SC3** unless the tasks could be executed simultaneously. Using the control variable disallows the parallel execution of tasks. In Bagger, as it is written, only one rule from one task can be executed at a time.

If one or both of the conflict resolution strategy and control variable is taken away from the program with no emulation in the translation, the program would exhibit incorrect behavior. Therefore, we must ensure any translation of the original program to one which exploits parallelism is correct. To ensure correctness, we must be able to express the program formally and use verification techniques. We believe that the process of translation and verification will be step by step, first simulating, expressing and proving the sequential version of the program, before moving on to exploiting parallelism and proving the correctness of these parallel programs. In the following sections, we provide a glimpse of the process. Each step must be researched in detail to develop the methodology for creating the programs.

3 Use of Swarm

Swarm [3] is a formal language that serves as a vehicle of investigation for this research. The language is based on the *shared dataspace* paradigm [11]. The term shared dataspace refers to the general class of programming languages in which the principal medium among concurrent components of programs is a common, content-addressable data structure called a *dataspace*. The state of a Swarm program is represented by such a dataspace. Swarm extends the UNITY[1] model to one which has a dynamically varying set of statements. To incorporate these and other features to perform rule-based reasoning, some proof rules in UNITY are replaced or extended. Except for these rules, the programming logics are identical.

3.1 Rule-based Programming in Swarm

We can make a correlation of a Swarm program to a rule-based program. For the purpose of this paper, the shared dataspace is divided into two subsets: the *tuple space* and the *transaction space.*

The tuple space in Swarm is equivalent to the working memory of a production system, if the working memory contains a **set** of working memory elements, i.e., no duplicates. A tuple is simply a working memory element. A transaction in the transaction space has a rule-like format. It consists of one or more subtransactions that have the form **LHS** $\longrightarrow$ **RHS**. The subtransactions of a transaction attempt to match their LHSs simultaneously. Those whose LHSs match, execute their RHSs simultaneously, with deletions from the tuple space preceding additions. Thus, production memory has no direct equivalent in Swarm. For example, a rule in production memory may be a transaction or a context may be a transaction with its rules as subtransactions. The distinction depends on the independence of the production rule from other production rules in the program and also on the way that the transaction space is organized.

The Swarm execution cycle, in the context of production systems is as follows:

i. Non-deterministically choose a transaction from the transaction space. A by-product of the choice is that the transaction is deleted.
ii. Match subtransactions simultaneously against the tuple space.
iii. Execute simultaneously all subtransactions that evaluate to **true** and change the dataspace, with all deletions preceding all additions. Transactions may be asserted but not deleted.

Termination of a serial production system occurs when the conflict set produced by the match phase is empty. Termination in Swarm occurs when the transaction space is empty. Thus, it will be necessary to carefully define when a transaction reasserts itself and when it does not, in order for the Swarm program to correctly terminate. This definition will come from the termination conditions of the original rule-based program.

3.2 Advantages of Swarm

It is necessary to express the program formally in order to prove properties to guarantee correctness. Our ultimate goal is to use a formal language for defining, developing, and proving correct rule-based programs that execute in parallel. Swarm is a formal language that has primitives similar to those in productions system. One major benefit of using this language is that we can use its associated proof theory. Thus, we will be able to show correctness of the behavior of a program during its development.

In Swarm, there is no general structure for a global control module or global scheduler as in production systems. During the execution cycle of Swarm no conflict set is produced. Any rule that is chosen and has subtransactions that are matched successfully, will be executed. There is no control mechanism to decide which transaction to chose for matching. Therefore, rule-based programs produced in Swarm will not rely on conflict resolution for rule execution order. This lack of control will force an analysis of the importance of global control in parallel rule-based programs.

There are other properties in Swarm that make it a desirable model. Because Swarm is architecture independent, concentration can be placed on developing solutions to the problems instead

of placing concern on the specific execution of the program on some architecture. Swarm is more expressive than a production system because reasoning can be performed over the entire dataspace, instead of just the tuple space or working memory. For example, a subtransaction may be satisfied on the basis of a transaction being present or absent from the transaction space. Also, asynchronous computation can be simulated and synchronous computation can be performed dynamically.

3.3 Encoding Bagger in Swarm

Figure 2 is a direct translation to Swarm of the rules to bag medium items from the English version in Figure 1, and the OPS5 version in Appendix A. The entire Swarm listing for this program can be found in Appendix B. Some notational explanation is necessary. Variables are in uppercase letters. The variable **I** is some grocery item, **S** is a size, **N** is a natural number used as a unique identifier, **W** is the weight, and **A** is a set representing the current contents of the bag. An additional tuple, **current**, is used to give the bags their unique identification. Such identification is necessary to maintain a set of tuples. Looking at **bmi1**, there is a dagger (†) after the **unbagged** tuple. This is a notational convenience meaning that if that subtransaction is satisfied, then the deletion of the **unbagged** tuple matched will be performed in the right-hand side of the executing subtransaction. The parallel bars (||) separate the subtransactions of a transaction. The subtransaction

unbagged(I) $\longrightarrow$ **bmi1**

reasserts the transaction **bmi1** into the transaction space as long as there are unbagged items left to bag (in the tuple space). We make the assumption that both the grocery tuples and unbagged tuples are unique. Also, we assume that *initially* for every grocery tuple, there is *exactly one* unbagged tuple.

Because all conflict resolution is eliminated in the direct translation, the results and behavior of the original program can no longer be guaranteed. For example, assume that there is a medium sized grocery item and an available bag at some point in the state of the computation. Then if any one of **bmi1, bmi2, bmi3** or **bmi4** is chosen, that transaction will match and execute. Since the choice is non-deterministic, **bmi4** could be chosen, matched and executed, changing **step** tuple to **bag-small-items** and never allowing the medium grocery item to be bagged.

This problem results from encoding the original program to rely explicitly on the **SC3** conflict resolution rule. The benefit of using this rule is that implicitly the absence of an instantiation in the conflict set represents the absence of a class of working memory elements in working memory. Though, the entire working memory is checked for this absence by the attempt to match the more specific rule, it is not shown explicitly. For example, **bmi2** actually relies on the absence of an available bag, though there is no checking to determine this absence. In general, production rules relying on **SC3** must be executed in a certain sequential order for the program to maintain correct behavior. In Bagger, this order conforms to the way groceries are normally bagged. You want to place groceries in an available bag while you can. When you can't because the bag is full, you get another bag and begin packing again. Therefore, the production rules associated with bagging a particular size item cannot be executed simultaneously, if the program exhibits the stated behavior.

Pasik[10] uses a *macrorule* to transform rules that always execute in the same sequential order into a single rule that performs all the actions of the sequential rule. He was able to increase the amount of parallelism available using macrorules. But this transformation technique will not work

```
bmi1 ≡
   I,C,N,W,A:
        step(bag-medium-items),
        grocery(I,medium,C,true), unbagged(I)†,
        bag(N,W,A)†, W ≤ 11
        ⟶
        bag(N, W+4, A ∪ I)
   ||
   I:   unbagged(I) ⟶ bmi1;

bmi2 ≡
   I,C,N,W,A:
        step(bag-medium-items),
        grocery(I,medium,C,false), unbagged(I)†,
        bag(N,W,A)†, W ≤ 11
        ⟶
        bag(N, W+4, A ∪ I)
   ||
   I:   unbagged(I) ⟶ bmi2;

bmi3 ≡
   I,C,F,N:
        step(bag-medium-items),
        grocery(I,medium,C,F), unbagged(I),
        current(N)†
        ⟶
        bag(N+1, 0, φ), current(N+1)
   ||
   I:   unbagged(I) ⟶ bmi3;

bmi4 ≡
        step(bag-medium-items)† ⟶ step(bag-small-items)
   ||
   I:   unbagged(I) ⟶ bmi4
```

Figure 2: Direct translation of OPS5 Bagger to Swarm

here, because these rules do not execute in the same sequential order. Their ordering depends on the number of items of each size available, and the number of bags generated.

We show two ways to transform a program, such as Bagger, which relies on the **SC3** rule. The first is to create a control variable or use an existing one in the program (such as **step**). By increasing the detail of a control variable, we can allow only one rule to match and execute at a time. The manner in which the control variable is changed must correspond to the flow of control among the rules, as dictated by **SC3**. This will be discussed in more detail below. The second method is to group the rules which interact which each other because of **SC3** (such as the **bmi1, bmi2, bmi3**) and make each rule mutually exclusive, in its LHS, of the other rules in the group. This will cause explicit test for absent information. This technique is used in the programs presented in Figures 4 and 6.

Figure 3 corrects the control problem of Figure 2 by applying the first technique above and making **step** more specific. The entire listing for this program can be found in Appendix C. A number corresponding to the rule that should execute is placed inside **step**, thereby allowing only a single rule at a time to match and execute. To change the number in **step**, the flow of control among the rules had to be analyzed. The **NAND** construct in Swarm provides a mechanism to correctly change **step**. Looking at **bmi1**, if one or both of the subtransactions are not satisfied, the **NAND** predicate will be satisfied, and the rest of that subtransaction will then be matched. The logic behind using the **NAND** in **bmi1** is that if the first subtransaction matches, then we don't want to change **step**, because we may still have a grocery item and available bag. If this subtransaction doesn't match, then we either don't have a frozen, unbagged, medium item or we don't have a bag, so we change **step** to allow **bmi2** to determine whether there are no frozen items or no bags. If the second subtransaction of **bmi1**, where the transaction is reasserted doesn't match, then control should be changed so that eventually **bmi4** will be executed to change **step**. There is a problem with this solution. If new rules are to be added, the flow of control between the rules must be re-analyzed to determine where the new rule would fit in. But this example shows that a program can be created that does not implicitly rely on the **SC3** conflict resolution. The program does not use **SC3** as stated since no conflict set is generated, but it could be said that the rule is emulated, since the task rules execute with the same behavior.

Pasik[10] found that the use of a control variable also limits the amount of parallelism in the program. Since some tasks must execute in a sequential order, he proposes organizing rules which can execute simultaneously with each other into a *rule set*, then have a global mechanism to determine when to switch rule sets for execution. For our purposes, this method is not useful because it trades one form of global control for another.

Swarm has the capability to add transactions to the transaction space during execution. All transactions must be defined, but all do not have to be in the transaction space initially or at the same time. Thus, we can simulate the effect of the control variable in a sequential program, by having the ending of one task assert the transactions for the next task.

For example in Figure 3, **bmi4** can be eliminated and the subtransaction with the **NAND** would then read:

NAND $\longrightarrow$ bsi1, bsi2, bsi3

Another way to avoid the use of the SC3 rule is to explicitly state what information should be absent for the rule to execute. In this manner, the rules within each task become mutually exclusive and

```
bmi1 ≡
   I,C,N,W,A:
        step(bag-medium-items,1),
        grocery(I,medium,C,true), unbagged(I)†,
        bag(N,W,A)†, W ≤ 11
        ⟶
        bag(N, W+4, A ∪ I)
   ||
   I:   unbagged(I) ⟶ bmi1;
   ||
        NAND, step(bag-medium-items,1)†
        ⟶
        step(bag-medium-items,2)

bmi2 ≡
   I,C,N,W,A:
        step(bag-medium-items,2),
        grocery(I,medium,C,false), unbagged(I)†,
        bag(N,W,A)†, W ≤ 11
        ⟶
        bag(N, W+4, A ∪ I)
   ||
   I:   unbagged(I) ⟶ bmi2;
   ||
        NAND, step(bag-medium-items,2)†
        ⟶
        step(bag-medium-items,3)

bmi3 ≡
   I,C,F,N:
        step(bag-medium-items,3)†,
        grocery(I,medium,C,F), unbagged(I),
        current(N)†
        ⟶
        bag(N+1, 0, φ), current(N+1), step(bag-medium-items,1)
   ||
   I:   unbagged(I) ⟶ bmi3;
   ||
        NAND, step(bag-medium-items,3)†
        ⟶
        step(bag-medium-items,4)

bmi4 ≡
        step(bag-medium-items,4)† ⟶ step(bag-small-items,1)
   ||
   I:   unbagged(I) ⟶ bmi4;
```

Figure 3: Correcting the problems in Figure 2

```
MD ≡                                        COMMENTS
  I,N,W,A:
      grocery(I,medium)†,                   used in place of unbagged(I) in Fig. 3
      bag(N,A,W)†, W ≤ 11
      ⟶
      bag(N,W+4, A ∪ I), MD                 reassert the transaction for this task
  ||
  I,N:
      grocery(I,medium),
      [∀ M,W,A: bag(M,W,A): W > 11],        make sure all existing bags will not work
      current(N)†
      ⟶
      bag(N+1,0,φ), current(N+1), MD
  ||
      [∀ I :: ¬grocery(I,medium)] ⟶ SM;     change tasks by asserting the next task's transaction
```

Figure 4: Using a specific Swarm mechanism

independent of one another. Figure 4 uses this method among the task rules[1]. The entire program listing for Figure 4 can be found in Appendix D. Along with using this method, we have combined all the task rules to form a single transaction. This method shows how an entire task can be mapped to a transaction with the task rules as subtransactions. Along with this mapping, we allow only the transaction for the current task to be in the dataspace. When the task is complete, the transaction will assert the transaction corresponding to the next task to be performed. Therefore, in this version of Bagger, only the transaction for the current task to bag one size item is in the transaction space. Therefore, it is the only one to be chosen. Because of this type of control only one size item can be bagged at a time. Therefore, if the transaction corresponding to bagging large items adds the transaction that bags medium items, and the medium transaction does the same for the transaction bagging small items, we have the same flow of control and task ordering generated by **step** variable in the original Bagger. Additionally, the program is concise and understandable. The only problem is that given numerous rules, each relying on something not matching in another rule, making them mutually exclusive will be cumbersome, unless the process can be automated.

4 Transformation

Some of the problems that arise when rule-based programs are executed in parallel may be a result of informal problem description and lack of capability to guarantee certain results. Currently a desirable solution for a rule-based program is guaranteed to be achieved through multiple executions of the program. The use of conflict resolution controls how solution paths are traversed, in order to guarantee such paths converge. Neither multiple executions nor explicit control strategies prove program correctness.

[1]For clarity, we have simplified the specifications and representation of Bagger in Swarm. The grocery(I,S) tuple will be matched and deleted in place of unbagged(I). Also, we are no longer concerned with the container of the item or whether or not it is frozen. This leaves only 3 main tasks in bagger, and no rules for frozen items.

4.1 Necessity of Formal Description and Proof

There does not exist a commonly accepted formal system in which a rule-based program can be expressed and proven correct. It is possible that a new way to express and prove rule-based programs, as in Swarm, may lead to a change in how these programs are designed. It is our belief that through the use of the Swarm proof theory, we can show equivalence of the globally controlled rule-based programs and their counterparts in Swarm.

By examining the rule-based version of Bagger, we can express properties of Bagger formally. Once the specifications have been determined, the next step is to prove the program meets these specification. A specification or property is called invariant if it is true initially and throughout the program execution. An invariant is proven in Swarm by showing it holds initially and by assuming it holds prior to the execution of a transaction and showing it holds after the execution of a transaction, for all transactions in the transaction space.

Other important properties that are part of the specification are called progress properties. These properties guarantee the program will progress from its initial state to its final state. A sampling of the specification for Bagger is shown in Figure 5.

As parallelism is introduced, the initial specifications may be changed or updated incrementally to express the behavior of the program in more general terms.

4.2 Proof Sketch for Program in Figure 4

Using the Swarm proof theory, we can show that the program in Figure 4 maintains the properties and invariants in Figure 5. This program is the easiest of the Bagger programs developed to show correctness because it is sequential in nature and we have restricted the program such that only a single transaction exists in the transaction space at any time. There are three transactions in the entire program: **LG** (to bag large items), **MD** (as shown), and **SM** (to bag small items). All three are similar in construction. SM is missing the last subtransaction, (similar to MD's in Figure 4) because when there are no small grocery items left, everything should be bagged and termination should occur.

Though not all the properties and invariants are represented in Figure 5, they give the flavor of the program behavior. The following is a proof sketch centered on the transactions for bagging medium items. It is not a formal proof. The LG transaction is initially in the transaction space alone and asserts the MD transaction, without reasserting itself, when there are no more large grocery items left in the tuple space.

Looking at the MD transaction in Figure 4, we can show it does not violate the invariants in Figure 5. Invariant 1 states that a medium item should be present in a bag only if there are no large items left. MD can violate this invariant only upon execution of its first subtransaction, since this is where an item is bagged. We assume the other invariants hold. By invariant 3 and 4, if a large item is bagged it remains bagged and is deleted from the tuple space. By construction of the program we know that:

$$(\forall \text{ I:: } [\text{MD}] \Rightarrow \neg\text{grocery(I,large)})$$

Definitions

Definition 1: Bagging an item.
[$\forall$ I: I $\in$ item: bagged(I) $\equiv$ [$\exists$ N,W,A: bag(N,W,A): I $\in$ A]]

Definition 2: Size of an item.
[$\forall$ I,S: grocery(I,S): size(I) = S]

Properties

Termination Condition:
POST $\equiv$ [$\forall$ I, S: I $\in$ item $\wedge$ S $\in$ size: $\neg$grocery(I,S) $\bigwedge$ bagged(I)]

Property 1: Bag Completion: INIT $\longmapsto$ POST
where **INIT** represents the initial conditions of the program.

Property 2: Bag Stability: **stable** POST

Bagger Invariants

Invariant 1: Large items are bagged before medium items
[$\forall$ I: I $\in$ item: [$\exists$ N,W,A,I': bag(N,W,A) $\wedge$ I' $\in$ A: size(I') = medium] $\Rightarrow$ $\neg$grocery(I,large)]

Invariant 2: The weight of every bag is at most maxweight.
[$\forall$ N,A,W: bag(N,A,W): W $\leq$ maxweight]

Invariant 3: Once an item is bagged, it remains bagged.
stable bagged(I)

Invariant 4: An item is deleted if and only if it is bagged.
[$\forall$ I,S: grocery(I,S) $\in$ INIT: bagged(I) $\Leftrightarrow$ $\neg$grocery(I,S)]

Figure 5: Example of Bagger properties and invariants

where [MD] means "MD is present in the transaction space". This holds because MD is asserted only when there are no large items left. Thus the invariant is not violated, since the first part of the disjunction is always satisfied.

In the program, we explicitly used the fact that **maxweight**=15 and the weight of a medium item is 4. Invariant 2 can be violated only if the first subtransaction in MD matches and increases the bag weight (represented by W) above 15. For the match to succeed, W cannot be greater than 11 and the weight added to the bag is at most 4, so the invariant is not violated.

There are two ways to violate invariant 3. Assuming bagged(I) was true for some item I, one way is to delete I from A for some bag(N,W,A) or another way is to delete bag(N,W,A) from the tuple space where $I \in A$. As in MD, no subtransaction in any transaction performs the deletion of an item from a bag. The first subtransaction in MD does delete bag(N,W,A) but reasserts it by adding to W and A, before the execution of the subtransaction is complete. Thus invariant 3 is not violated.

A grocery item must be bagged and not deleted, or deleted and not bagged, to violate invariant 4. No transaction assert grocery tuples into the tuple space. The first subtransaction of MD is the only one that can violate this invariant. It does not, since when executed, the deletion of a grocery item is coupled with the bagging of that item.

The most difficult part of the proof is to show that progress is made during the execution of the program, as represented by property 1 in Figure 5. We present a brief overview of the proof. To show property 1 holds, we show that the program beginning at its initial conditions (INIT) will eventually lead to ($\longmapsto$) a satisfaction of the termination conditions (POST). The LG transaction is part of the initial conditions, along with the grocery(I,S) tuples. The proof is broken down into parts. First, it is shown that the execution of the program with only the LG transaction in the transaction space eventually leads to having only the MD in the transaction space ([LG] $\longmapsto$ [MD]). A similar transition from MD to SM is shown also.

Decomposing the proof further, to show the transition of the transaction space contents from LG to MD, we show that the number of large groceries decreases eventually to zero, causing MD to be asserted into the transaction space. Once MD is in the transaction space, we show the same decrease in medium items. Then we can prove the transition is made from MD to SM in the transaction space. With SM in the transaction space, we show that the number of small items decreases to zero. At this point in the program execution, using the invariants, we can show no grocery items remain in the tuple space. Thus, the POST is reached. Then we must show that this property is stable, again using the invariants.

5 Exploiting Parallelism

Bagger does have properties that give it potential for parallel execution. First, there are multiple items of different sizes. Can the same size items be packed simultaneously? Can the packing of different size items overlap, while maintaining the correctness criteria? Second, normally the result will contain multiple bags. Can these bags be packed simultaneously? If a correct program can be developed to incorporate the above characteristics, then we believe the program would exploit most of its potential parallelism.

```
MD ≡                                                    COMMENTS
  I,N,W,A:
      grocery(I,medium)†,
      bag(N,W,A,medium)†, W ≤ 11
      ⟶                                                 put a medium item in a medium bag
      bag(N,W+4,A ∪ I,medium)
  ||
  I,N:
      [∀ N',W',A' :: ¬bag(N',W',A',large)],             create a new bag only if there are
      [∀ I' :: ¬grocery(I',large)],                     no large bags and no large items
      grocery(I,medium),
      [∀ M,W,A: bag(M,W,A,medium): W > 11]
      current(N)†
      ⟶
      bag(N+1,0,φ,medium), current(N+1)
  ||
  N,W,A:
      bag(N,W,A,large)†, 9 < W                          change large bags that can't hold
      ⟶                                                 more large items, to medium bags
      bag(N,W,A,medium), MD
  ||
  N,W,A:
      [∀ I :: ¬grocery(I,large)], bag(N,W,A,large)†     when there are no more large items
      ⟶                                                 change all large bags to medium bags
      bag(N,W,A,medium), MD
  ||
  I,S: grocery(I,S) ⟶ MD;
```

Figure 6: Parallel Bagger

5.1 A Pipeline Version of Bagger

There are several ways to write this program, both Swarm-specific ways and general ways. The goals for the parallel program are 1) to maintain correctness criteria, 2) to not waste computation time unnecessarily, and 3) to correspond closely to how we view the program should execute in the real world. One such program is to incorporate a generate and test mechanism. For example, bags are randomly packed with groceries and tested. If they are not correctly bagged, then the groceries are dumped out and the process continues. We recognize that invariant 3 would have to be changed for this process. This program seems to waste computation and really does not correspond to the way groceries are bagged. Thus, we do not consider it a viable solution for this problem. Another example is to calculate and generate the minimum number of necessary bags, then pack large items in the bottom of each simultaneously, then pack medium items simultaneously, and so on. This is a synchronous type of solution, but does not allow items of different sizes to be packed simultaneously. But such a solution would be satisfactory according to the above goals.

Figure 6 presents the MD transaction of a program which allows the parallelism discussed above and also achieves the goals. The entire Swarm listing can be found in Appendix E. In this program, it is possible that medium and small items can be bagged while large items exist in the dataspace,

by allowing each bag to be in control of what is packed. The bag is "reserved" for a certain size item to be packed into it. Initially, let bag_1 be reserved for large items only. When it can no longer hold large items, it creates bag_2 to hold the large items and the bag_1 becomes reserved for medium items. Medium items are bagged in bag_1 until it can no longer hold medium items, then it is reserved for small. Meanwhile, bag_2 has reached a limit on its large items, so bag_3 is created for large items and bag_2 becomes reserved for medium items. At this point, the program has three bags currently being bagged with each size item. This solution exploits parallelism in the problem because bags can be packed asynchronously, in a pipeline. Also, different size items can be packed simultaneously, exploiting even more parallelism. The best solution would be a combination of the synchronous and asynchronous solutions since it would encompass all possible parallelism.

Note that due to the parallelism some invariants may have to change. For example, invariant 1 needs to be updated to: A medium item can be present in a bag only if there are no large items in the tuple space OR all bags cannot hold more large items.

5.2 Mapping Bagger to Parallel Architectures

There are easy ways to map the program in Figure 6 to a parallel architectures. Given a parallel architecture with multiple processors operating on a shared memory dataspace, we can have each processor execute the Swarm cycle asynchronously and at its own pace, with appropriate locks placed on the elements of the tuple space. When a transaction is chosen from the transaction space, it is deleted, so two processors cannot access the same transaction simultaneously. Some constraints on which transactions can execute simultaneously may be necessary.

Assume we have an architecture in which the processors can have a local transaction space, but the tuple space is shared among them. Then each transaction, (LG, MD, SM), can be placed at a separate processor where it is constantly executed on the tuple space. Again, with the appropriate locks, each transaction can execute simultaneously. Also, we can replicate the entire program at each processor, giving the impression of multiple workers packing multiple bags with different size items.

6 Conclusion

We conclude that the **SC3** conflict resolution rule can be emulated when used in rule-based programs as it is used in Bagger. The best way to emulate the use of the **SC3** rule is to group those production rules whose interaction with each other rely on this conflict resolution rule, and make each production rule in the group mutually exclusive of the other rules in its group.

We have shown one way, specific to the Swarm language, that, if necessary, tasks can be ordered similar to the use of the control variable. If this methodology cannot be adopted in the general languages in which rule-based programs are expressed, at this time we believe that the control variable should remain in working memory.

By studying the characteristics of a rule-based program and the problem it is to solve, we can determine where potential parallelism lies. Then by taking into account its initial description, we can design a correct program to exploit increased parallelism.

We believe that applying verification techniques to rule-based programs will allow these programs to be used in real world application where guarantees in behavior are necessary. We are encouraged

by the results achieved by utilizing formal methods to show correctness in the behavior of a rule-based program. By formally specifying Bagger, we were able to determine the most important properties that were to be maintained. As seen from our example, using the shared-dataspace language, Swarm, has aided us in transforming the original Bagger to one which does not rely on conflict resolution, and proving the equivalence of the new program to the old.

7 Discussion and Future Work

There is still much work to be done in analyzing the role of conflict resolution in rule-based programs. Further research in this area will include examining ways in which programs relying on different conflict resolution rules or strategies can be transformed. We want to develop a general methodology for such transformations to produce equivalent programs which do not rely on conflict resolution and exhibit correct behavior.

There are obvious differences between OPS5-like languages and Swarm. We believe that we can exploit these differences to our advantage because of the flexibility of Swarm and its associated proof theory. The main difference is that the execution cycle is unlike any production system language currently available. The different rule execution cycle does not make it necessary to match all rules to determine which ones have instantiations, since only one rule is chosen at a time to match and execute. Instead all rules in the form of transactions must be independent of one another. The increase in independence, while maintaining correct behavior, will make the mapping of the program to parallel architectures easier.

Executing rule-based programs in a parallel production system model creates additional correctness criteria to guarantee certain results. One of the main problems in parallel production systems, which has received a lot of attention, is the maintenance of serializability [6, 12, 13, 9]. Rules may interfere with each other by changing data upon which other rules initially relied. The interleaving of actions of rules fired in parallel may produce solutions that cannot be obtained in a serial execution of the rules. This problem was mentioned in section 5.2, in which additional constraints may be needed to make sure two interfering transactions do not execute simultaneously. How this problem will eventually be addressed within our research is the subject of more study.

A problem that is due also to parallel execution of a rule-based program and is related to serializability, is the problem of multi-rule execution ordering. When multiple rules are fired asynchronously from different sources, restrictions must be placed on the order in which changes are made to working memory. Also, the order in which rules match working memory may be restricted. These restrictions are necessary for achieving a correct solution. We hope, that by developing a methodology for designing rule-based programs for parallel execution, these problems will be dealt with earlier in the design of the program.

Acknowledgements

Thanks go to Dr. W.E. Ball, director of CICS, for his support and encouragement of this project, Dr. G.-C. Roman, for directing me toward Swarm as a vehicle for investigation, Dr. J.G. Schmolze for discussing the problems of the current parallel production system benchmark programs, and T. Gamble for comments on earlier drafts of this paper.

References

[1] CHANDY, K., AND MISRA, J. *Parallel Program Design: A Foundation.* Addison Wesley, Reading, MA, 1988.

[2] CUNNINGHAM, H., AND ROMAN, G.-C. Toward formal verification of rule-based systems: A shared dataspace perspective. Tech. Rep. WUCS-89-28, Washington University, June 1989.

[3] CUNNINGHAM, H., AND ROMAN, G.-C. A UNITY-style programming logic for a shared dataspace language. Tech. Rep. WUCS-89-5, Washington University, March 1989. To appear in *IEEE Transaction on Distributed and Parallel Computing.*

[4] FORGY, C. OPS5 user's manual. Tech. Rep. CMU-CS-81-135, Carnegie-Mellon University, 1981.

[5] GUPTA, A. *Parallelism in Production Systems.* Pitman Publishing, London, England, 1987.

[6] ISHIDA, T., AND STOLFO, S. 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.

[7] MCDERMOTT, J., AND FORGY, C. Production system conflict resolution strategies. In *Pattern-Directed Inference Systems*, D. Waterman and F. Hayes-Roth, Eds. Academic Press, New York, NY, 1978.

[8] MIRANKER, D., KUO, C., AND BROWNE, J. Compiling parallelism among rules. In *IJCAI-89 Workshop on Parallel Algorithms* (August 1989). Unpublished proceedings.

[9] MIRANKER, D., KUO, C., AND BROWNE, J. Parallel transformation for a concurrent rule execution language. Tech. Rep. TR-89-30, University of Texas at Austin, October 1989.

[10] PASIK, A. A methodology for programming production systems and its implementations on parallelism. Tech. Rep. Technical report and Ph.D dissertation, Columbia University, 1988.

[11] ROMAN, G.-C., AND CUNNINGHAM, H. A shared dataspace model of concurrency-language and programming implications. In *Proceedings of the 9th International Conference on Distributed Computing Systems* (Los Alamitos, CA, 1989), IEEE Computer Society Press, pp. 270–9.

[12] SCHMOLZE, J. Guaranteeing serializable results in synchronous parallel production systems. Tech. Rep. 89-5, Tufts University, Medford MA, October 1989.

[13] SCHMOLZE, J., AND GOEL, S. A parallel asynchronous distributed production system. In *Submitted to ICPP-90* (1990). Technical Report in Preparation.

[14] TAMBE, M., KALP, D., GUPT, A., FORGY, C., MILNES, B., AND NEWELL, A. Soar/PSM-E: Investigating match parallelism in a learning production system. In *ACM/SIGPLAN PPEALS* (Sept. 1988), pp. 146–160.

[15] WINSTON, P. *Artificial Intelligence, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Mass., 1984.

A OPS5 Bagger

This section contains the original Bagger program written in OPS5 using the LEX conflict resolution strategy. Weights are given to each size grocery item with a large item weighing 6lbs, a medium 4lbs, and a small 2lbs.

Specifications: 1) large bottles are bagged first, large items are bagged next, then medium, then small, 2) frozen items are bagged before non-frozen items (only for medium and small items), 3) a bag can hold a maximum of 15lbs. of groceries.

```
(literalize grocery item container size frozen)
(literalize bag weight contains)
(literalize unbagged item)
(literalize step is)


(p property_list
        (inventory)
-->
        (remove 1)
        (make grocery
                ^item bread
                ^container plastic-bag
                ^size medium
                ^frozen no)
        (make grocery
                ^item glop
                ^container jar
                ^size small
                ^frozen no)
        (make grocery
                ^item granola
                ^container cardboard_box
                ^size large
                ^frozen no)
        (make grocery
                ^item ice_cream
                ^container cardboard_carton
                ^size medium
                ^frozen yes)
        (make grocery
                ^item paper_towels
                ^container plastic_bag
                ^size large
                ^frozen no)
```

```
        (make grocery
                ^item Pepsi
                ^container bottle
                ^size large
                ^frozen no)
        (make grocery
                ^item popsicles
                ^container cardboard_carton
                ^size small
                ^frozen yes)
        (make grocery
                ^item potato_chips
                ^container plastic_bag
                ^size medium
                ^frozen no))

(p l1
        (list1)
-->
        (remove 1)
        (make step ^is bag-large-bottles)
        (make unbagged ^item bread)
        (make unbagged ^item glop)
        (make unbagged ^item ice_cream)
        (make unbagged ^item paper_towels)
        (make unbagged ^item Pepsi)
        (make unbagged ^item popsicles)
        (make unbagged ^item granola)
        (make unbagged ^item potato-chips)
        (make current ^number 1))

(make list1)
(make inventory)


(p blb1
        (step ^is bag-large-bottles)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size large ^container bottle)
        (bag ^weight { < 9 <y> })
-->
        (write Putting bottle of <x> in bag (crlf))
        (remove 2)
        (modify 4 (substr 4 1 inf) <x>
```

```
                ^weight (compute <y> + 6) ))

(p blb2
        (step ^is bag-large-bottles)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size large ^container bottle)
-->
        (write Starting a fresh bag)
        (make bag ^weight 0))


(p blb3
        (step ^is bag-large-bottles)
-->
        (modify 1 ^is bag-large-items))

(p bli1
        (step ^is bag-large-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size large)
        (bag ^weight { < 9 <y> })
-->
        (write Putting large item <x> in bag (crlf))
        (remove 2)
        (modify 4 (substr 4 1 inf) <x> ^weight (compute <y> + 6)))

(p bli2
        (step ^is bag-large-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size large)
-->
        (make bag ^weight 0))


(p bli3
        (step ^is bag-large-items)
-->
        (modify 1 ^is bag-medium-items))

(p bmi1
        (step ^is bag-medium-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size medium ^frozen yes)
        (bag ^weight { < 11 <y> })
```

```
-->
        (remove 2)
        (modify 4 (substr 4 1 inf) <x> ^weight (compute <y> + 4)))

(p bmi2
        (step ^is bag-medium-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size medium)
        (bag ^size medium ^weight { < 11 <y> })
-->
        (write Putting medium item <x> in the bag (crlf))
        (remove 2)
        (modify 4 (substr 4 1 inf) <x> ^weight (compute <y> + 4)))

(p bmi3
        (step ^is bag-medium-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size medium)
-->
        (make bag ^weight 0))

(p bmi4
        (step ^is bag-medium-items)
-->
        (modify 1 ^is bag-small-items))

(p bsi1
        (step ^is bag-small-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size small ^frozen yes)
        (bag ^weight { < 13 <y> })
-->
        (remove 2)
        (modify 4 (substr 4 1 inf) <x> ^weight (compute <y> + 2)))



(p bsi2
        (step ^is bag-small-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size small)
        (bag ^weight { < 13 <y> } )
-->
        (remove 2)
```

```
        (modify 4 (substr 4 1 inf) <x> ^weight (compute <y> + 2)))

(p bsi3
        (step ^is bag-small-items)
        (unbagged ^item <x>)
        (grocery ^item <x> ^size small)
-->
        (make bag ^weight 0))

(p s1
        (step ^is bag-small-items)
        -(unbagged ^item <x>)
        (grocery ^item <x> )
-->
        (write (crlf) Grand summary of your purchases (crlf))
        (modify 1 ^is summarize))

(p s2
        (step ^is summarize)
        (bag)
-->
        (write Bag: (substr 2 4 inf) (crlf))
        (remove 2))

(p s3
        (step ^is summarize)
-->
        (remove 1)
        (halt))
```

B Direct Translation of Bagger from OPS5 to Swarm

In this the Bagger program is directly translated (rule to transaction) from OPS5 to Swarm. **The rules do not reflect the correct logic due to the absence of a conflict set and conflict resolution.** Also, the programs are presented as such for clarity, and for easy comparison. There are more efficient ways to represent the programs in Swarm. These will be presented in a later papers.

OPS5 working memory allows multiple working memory elements that are identical. Because the tuple space in Swarm is a *set* of tuples, each tuple must be unique. This causes some translation changes. Though there are ways to directly translate, here we have simply assumed that each item is unique. Therefore, you can must have unique item descriptions in the grocery tuples, and only one unbagged item of that type.

Program Bagger (items,container,size,maxwgt,step:

items $\equiv$ (*enumeration of all item names*);
container $\equiv$ (*enumeration of all container names*);
size $\equiv$ (small, medium, large);
maxwgt $\in \mathbb{N}$;
step $\equiv$ (bag-large-bottles, bag-large-items, bag-medium-items,
bag-small-items))

tuple types

[I,C,S,F,W,A,N,B :
I $\in$ items, C $\in$ container, S $\in$ size, B $\in$ step,
$1 \leq$ W $\leq$ maxwgt, A $\subseteq$ items, F $\in$ boolean, N $\in \mathbb{N}$:
grocery(I,C,S,F);
bag(N,W,A);
unbagged(I);
current(N);
step(B)
]

transaction types

Begin large bottle transactions

[blb1 $\equiv$
I,F,N,W,A:
step(bag-large-bottles),
grocery(I,large,bottle,F), unbagged(I)†,
bag(N,W,A)†, W $\leq$ 9
$\longrightarrow$
bag(N, W+6, A $\cup$ I)
‖

I: unbagged(I) $\longrightarrow$ blb1;

blb2 $\equiv$
I,F,N:
step(bag-large-bottles),
grocery(I,large,bottle,F), unbagged(I)
current(N)†,
$\longrightarrow$
bag(N+1, 0, ϕ), current(N+1)
||
I: unbagged(I) $\longrightarrow$ blb2;

blb3 $\equiv$
step(bag-large-bottles)† $\longrightarrow$ step(bag-large-items)
||
I: unbagged(I) $\longrightarrow$ blb3;

End large bottle transactions.
Begin large item transactions.

bli1 $\equiv$
I,C,F,N,W,A:
step(bag-large-items),
grocery(I,large,C,F), unbagged(I)†,
bag(N,W,A)†, W $\leq$ 9
$\longrightarrow$
bag(N, W+6, A $\cup$ I)
||
I: unbagged(I) $\longrightarrow$ bli1;

bli2 $\equiv$
I,C,F,N:
step(bag-large-items),
grocery(I,large,C,F), unbagged(I)
current(N)†,
$\longrightarrow$
bag(N+1, 0, ϕ),current(N+1)
||
I: unbagged(I) $\longrightarrow$ bli2;

bli3 $\equiv$
step(bag-large-items)† $\longrightarrow$ step(bag-medium-items)
||
I: unbagged(I) $\longrightarrow$ bli3;

End large item transactions.
Begin medium item transactions.

bmi1 ≡
I,C,N,W,A:
step(bag-medium-items),
grocery(I,medium,C,true), unbagged(I)†,
bag(N,W,A)†, W ≤ 11
⟶
bag(N, W+4, A ∪ I)
‖
I: unbagged(I) ⟶ bmi1;

bmi2 ≡
I,C,N,W,A:
step(bag-medium-items),
grocery(I,medium,C,false),unbagged(I)†,
bag(N,W,A)†, W ≤ 11
⟶
bag(N, W+4, A ∪ I)
‖
I: unbagged(I) ⟶ bmi2;

bmi3 ≡
I,C,F,N:
step(bag-medium-items),
grocery(I,medium,C,F), unbagged(I),
current(N)†
⟶
bag(N+1, 0, ϕ), current(N+1)
‖
I: unbagged(I) ⟶ bmi3;

bmi4 ≡
step(bag-medium-items)† ⟶ step(bag-small-items)
‖
I: unbagged(I) ⟶ bmi4;

End medium item transactions.
Begin small item transactions.

bsi1 ≡
I,C,N,W,A:

step(bag-small-items),
grocery(I,small,C,true), unbagged(I)†,
bag(N,W,A)†, W $\leq$ 13
$\longrightarrow$
bag(N, W+2, A $\cup$ I)

‖

I: unbagged(I) $\longrightarrow$ bsi1;

bsi2 $\equiv$

I,C,N,W,A:

step(bag-small-items),
grocery(I,small,C,false), unbagged(I)†,
bag(N,W,A)†, W $\leq$ 13
$\longrightarrow$
bag(N, W+2, A $\cup$ I)

‖

I: unbagged(I) $\longrightarrow$ bsi2;

bsi3 $\equiv$

I,C,F,N:

step(bag-small-items),
grocery(I,small,C,F), unbagged(I),
current(N)†
$\longrightarrow$
bag(N+1, 0, ϕ), current(N+1)

‖

I: unbagged(I) $\longrightarrow$ bsi3;

End small item transactions.

]

End of transaction types

initialization

The following tuples represent a sample grocery list

grocery(glop,jar,small,false);
grocery(popsicles,cardboard-box,small,true);
grocery(bread,plastic-bag,medium,false);
grocery(ice-cream,cardboard-box,medium,true);
grocery(potato-chips,plastic-bag,medium,false);
grocery(granola,cardboard-box,large,false);
grocery(paper-towels,plastic-bag,large,false);
grocery(pepsi,bottle,large,false);

unbagged(glop);
unbagged(popsicles);
unbagged(bread);
unbagged(ice-cream);
unbagged(potato-chips);
unbagged(granola);
unbagged(paper-towels);
unbagged(pepsi);

current(1);

step(bag-large-bottles);

bag(1,0,ϕ);

blb1; blb2; blb3;

bli1; bli2; bli3;

bmi1; bmi2; bmi3;

bsi1; bsi2; bsi3; bsi4.

End program initialization.

C Corrected Directed Translation of Bagger

This section includes the Swarm program of Bagger using numbers to distinguish among rules in a task. These number will dictate exactly when the rule can fire, simulating how the specificity (or special case) conflict resolution rule is used in the OPS5 Bagger problem to control rule execution order.

The **NAND** construct in Swarm reads as follows: *If one the local subtransactions is not satisfied, i.e., cannot fire, then attempt to match the following subtransaction.* The **NAND** is considered a global subtransaction query because it can examine the firing of the other subtransactions, which issue local queries. But it is only global within its transaction, not within the entire program. Therefore, the **NAND** is satisfied if one of the local subtransactions is not satisfied. There are also, **AND, OR,** and **NOR** global subtransaction queries.

Program Bagger (items,container,size,maxwgt,step:

items ≡ (*enumeration of all item names*);
container ≡ (*enumeration of all container names*);
size ≡ (small, medium, large);
maxitems ∈ $\mathbb{N}$;
step ≡ (bag-large-bottles, bag-large-items, bag-medium-items,
bag-small-items))

tuple types

[I,C,S,B,W,A,F,N :
I ∈ items, C ∈ container, S ∈ size, B ∈ step,
1 ≤ W ≤ maxwgt, A ⊆ items, boolean(F), N ∈ $\mathbb{N}$:
grocery(I,C,S,F);
bag(N,W,A);
unbagged(I);
current(N);
step(B)
]

transaction types

Begin large bottle transactions.

[blb1 ≡
I,F,N,W,A :
step(bag-large-bottles,1),
grocery(I,large,bottle,F), unbagged(I)†,
bag(N,W,A)†, W ≤ 9
⟶
bag(N, W+6, A ∪ I)

‖
I: unbagged(I) ⟶ blb1;
‖
NAND,
step(bag-large-bottles,1)† ⟶ step(bag-large-items,2)

blb2 ≡
I,F,N:
step(bag-large-bottles,2)†,
grocery(I,large,bottle,F), unbagged(I),
current(N)†,
⟶
bag(N+1, 0, ϕ), step(bag-large-bottles,1), current(N+1)
‖
I: unbagged(I) ⟶ blb2;
‖
NAND,
step(bag-large-bottles,2)† ⟶ step(bag-large-items,3)

blb3 ≡
step(bag-large-bottles,3)† ⟶ step(bag-large-items,1)
‖
I: unbagged(I) ⟶ blb3;

End large bottle transactions.
Begin large item transactions.

bli1 ≡
I,C,F,N,W,A:
step(bag-large-items,1),
grocery(I,large,C,F), unbagged(I)†,
bag(N,W,A)†, W ≤ 9
⟶
bag(N, W+6, A ∪ I)
‖
I: unbagged(I) ⟶ bli1;
‖
NAND,
step(bag-large-items,1)† ⟶ step(bag-large-items,2)

bli2 ≡
I,C,F,N:
step(bag-large-items,2)†,
grocery(I,large,C,F), unbagged(I),

current(N)†
⟶
bag(N+1, 0, ϕ),current(N+1)

‖

I: unbagged(I) ⟶ bli2;

‖

NAND,
step(bag-large-items,2)† ⟶ step(bag-large-items,3)

bli3 ≡

step(bag-large-items,3)† ⟶ step(bag-medium-items,1)

‖

I: unbagged(I) ⟶ bli3;

End large item transactions.
Begin medium item transactions.

bmi1 ≡

I,C,N,W,A:
step(bag-medium-items,1),
grocery(I,medium,C,true), unbagged(I)†,
bag(N,W,A)†, W ≤ 11
⟶
bag(N, W+4, A ∪ I)

‖

I: unbagged(I) ⟶ bmi1;

‖

NAND,
step(bag-medium-items,1)† ⟶ step(bag-medium-items,2)

bmi2 ≡

I,C,N,W,A:
step(bag-medium-items,2),
grocery(I,medium,C,false), unbagged(I)†,
bag(N,W,A)†, W ≤ 11
⟶
bag(N, W+4, A ∪ I)

‖

I: unbagged(I) ⟶ bmi2;

‖

NAND,
step(bag-medium-items,2)† ⟶ step(bag-medium-items,3)

bmi3 ≡

I,C,F,N:
step(bag-medium-items,3)†,
grocery(I,medium,C,F), unbagged(I),
current(N)†
⟶
bag(N+1, 0, ϕ), current(N+1), step(bag-medium-items,1)
‖
I: unbagged(I) ⟶ bmi3;
‖
NAND,
step(bag-medium-items,3)† ⟶ step(bag-medium-items,4)

bmi4 ≡
step(bag-medium-items,4)† ⟶ step(bag-small-items,1)
‖
I: unbagged(I) ⟶ bmi4;

End medium item transactions.
Begin small item transactions.

bsi1 ≡
I,W,C,A,N:
step(bag-small-items,1),
grocery(I,small,C,true), unbagged(I)†,
bag(N,W,A)†, W ≤ 13
⟶
bag(N, W+2, A ∪ I)
‖
I: unbagged(I) ⟶ bsi1;
‖
NAND,
step(bag-small-items,1)† ⟶ step(bag-small-items,2)

bsi2 ≡
I,C,N,W,A:
step(bag-small-items,2),
grocery(I,small,C,false), unbagged(I)†,

bag(N,W,A)†, W ≤ 13
⟶
bag(N, W+2, A ∪ I)
‖
I: unbagged(I) ⟶ bsi2;
‖

NAND,
step(bag-small-items,2)† ⟶ step(bag-small-items,3)

bsi3 ≡
I,C,F,N:
step(bag-small-items,3)†,
grocery(I,small,C,F), unbagged(I),
current(N)†
⟶
bag(N+1, 0, ϕ), current(N+1), step(bag-small-items,1)
‖
I: unbagged(I) ⟶ bsi3;

End small item transactions.
]
End of transaction types

initialization

The following tuples represent a sample grocery list

grocery(glop,jar,small,false);
grocery(popsicles,cardboard-box,small,true);
grocery(bread,plastic-bag,medium,false);
grocery(ice-cream,cardboard-box,medium,true);
grocery(potato-chips,plastic-bag,medium,false);
grocery(granola,cardboard-box,large,false);
grocery(paper-towels,plastic-bag,large,false);
grocery(pepsi,bottle,large,false);

unbagged(glop);
unbagged(popsicles);
unbagged(bread);
unbagged(ice-cream);
unbagged(potato-chips);
unbagged(granola);
unbagged(paper-towels);
unbagged(pepsi);

current(1);

step(bag-large-bottles);

bag(1,0,ϕ);

blb1; blb2; blb3;

bli1; bli2; bli3;

bmi1; bmi2; bmi3; bmi4;

bsi1; bsi2; bsi3.

End program initialization.

D Swarm Bagger: Sequential Version

This program is a smaller version of the original Bagger problem. The step to bag large bottles is eliminated, along with the step to bag frozen items before items of the same size. The reasons for the elimination are clarity and conciseness. Also, the grocery tuple is both an identifier and marks the place of an unbagged item, eliminating the unbagged tuple. Because of the uniqueness assumption as in the previous programs, it is unnecessary in this program to have both an unbagged tuple and a grocery tuple. Therefore, the grocery tuples are now deleted from the tuple space as those items are bagged, and when there are no longer an grocery tuples left, the program terminates. In this section, we present a different approach to the problem of conflict resolution and task ordering, which is easier to understand with less detail.

In this program, we have eliminated the step tuple. In its place, we use the ability of Swarm to input predefined transactions into the transaction space during program execution. All the rules corresponding to one task are now subtransactions of a single transaction. That transaction represents a context. Now all task rules in a context will be matched simultaneously, and if satisfied, will execute simultaneously. In Bagger, these subtransactions are not written to execute simultaneously. Therefore, they must be made mutually exclusive, since the specificity conflict resolution rule is not used. This causes a "forall" statement to be used. The reader should not be alarmed by this statement, since it is implicitly used in the OPS5 version.

Program Bagger (items,maxwgt,size:

items $\equiv$ {*enumeration of all item types*}
maxwgt $\in \mathbb{N}$;
size $\equiv$ {small, medium, large}

tuple types

[I,S,N,W,A:
I $\in$ items, S $\in$ size, N $\in \mathbb{N}$, A $\subseteq$ items, $1 \leq W \leq$ maxwgt:
grocery(I,S);
bag(N,W,A);
current(N)
]

transaction types

[LG $\equiv$
I,N,W,A:
grocery(I,large)†,
bag(N,W,A)†, W ≤ 9
$\longrightarrow$
bag(N, W+6, A $\cup$ I), LG
‖
I,N:

grocery(I,large),
[$\forall$ M,W,A : bag(M,W,A); W > 9],
current(N)†
$\longrightarrow$
bag(N+1,0,ϕ), current(N+1), LG

||

[$\forall$ I :: $\neg$grocery(I,large)] $\longrightarrow$ MD;

MD $\equiv$

I,N,W,A:

grocery(I,medium)†,
bag(N,W,A)†, W $\leq$ 11
$\longrightarrow$
bag(N, W+4, A $\cup$ I), MD

||

I,N:

grocery(I,medium),
[$\forall$ M,W,A: bag(M,W,A): W > 11],
current(N)†
$\longrightarrow$
bag(N+1,0,ϕ), current(N+1), MD

||

[$\forall$ I :: $\neg$grocery(I,medium)] $\longrightarrow$ SM;

SM $\equiv$

I,N,W,A:

grocery(I,small)†,
bag(N,W,A)†, W $\leq$ 13
$\longrightarrow$
bag(N, W+2, A $\cup$ I), SM

||

I,N:

grocery(I,small),
[$\forall$ M,W,A: bag(M,W,A): W > 13],
current(N)†
$\longrightarrow$
bag(N+1,0,ϕ), current(N+1), SM

] *end transaction types*

initialization

grocery(glop,small);
grocery(popsicles,small);

grocery(bread,medium);
grocery(ice-cream,medium);
grocery(potato-chips,medium);
grocery(granola,large);
grocery(paper-towels,large);
grocery(pepsi,large);

current(1);

bag(1,ϕ,0);

LG;

E Swarm Bagger: Pipelining Version

This section contains the first version of Bagger in Swarm with no influence from OPS5. This version would be considered a parallel version because it makes use of the available pipelining in the program.

An extra field has been added to the bag tuple. This field represents the type of items that can be packed in the bag. It is a reservation field in a sense. For example, if the size field in the bag matches to "medium", then only medium sized items can be packed into the bag.

Program Bagger (items,maxwgt,size:

items $\equiv$ {*enumeration of all item types*};
maxwgt: integer natural;
size $\equiv$ {small, medium, large})

tuple types

[I,S,N,W,A:
I $\in$ items, S $\in$ size, N $\in \mathbb{N}$, A $\subseteq$ items, W $\leq$ maxwgt:
grocery(I,S);
bag(N,W,A,S);
current(N)
]

transaction types

[LG $\equiv$
I,N,W,A:
grocery(I,large)†,
bag(N,W,A,large)†, W $\leq$ 9
$\longrightarrow$
bag(N, W+6, A $\cup$ I, large)
‖
I,M,W,A,N:
grocery(I,large),
[$\forall$ M,W,A: bag(M,W,A,large): W > 9],
current(N)†
$\longrightarrow$
bag(N+1,0,ϕ,large), current(N+1)
‖
I: grocery(I,large) $\longrightarrow$ LG;

MD $\equiv$
I,N,W,A:
grocery(I,medium)†,
bag(N,W,A,medium)†, W $\leq$ 11

$\longrightarrow$
bag(N, W+4, A $\cup$ I, medium)

$\|$

I,N:
[$\forall$ N',W',A' :: $\neg$bag(N',W',A',large)],
[$\forall$ I' :: $\neg$grocery(I',large)],
grocery(I,medium),
[$\forall$ M,W,A: bag(M,W,A,medium): W > 11],
current(N)$\dagger$
$\longrightarrow$
bag(N+1,0,ϕ,medium),current(N+1)

$\|$

N,W,A:
bag(N,W,A,large)$\dagger$, 9 < W
$\longrightarrow$
bag(N,W,A,medium), MD

$\|$

N,W,A:
[$\forall$ I :: $\neg$grocery(I,large)], bag(N,W,A,large)$\dagger$
$\longrightarrow$
bag(N,W,A,medium), MD

$\|$

I: grocery(I,medium) $\longrightarrow$ MD;

SM $\equiv$

I,N,W,A:
grocery(I,small)$\dagger$,
bag(N,W,A,small)$\dagger$, W $\leq$ 13
$\longrightarrow$
bag(N, W+2, A $\cup$ I, small)

$\|$

I,N:
[$\forall$ N',W',A' :: $\neg$bag(N',W',A',medium)],
[$\forall$ I' :: $\neg$grocery(I',medium)],
[$\forall$ N',W',A' :: $\neg$bag(N',W',A',large)],
[$\forall$ I' :: $\neg$grocery(I',large)],
grocery(I,small),
[$\forall$ M,W,A: bag(M,W,A,small): W > 13],
current(N)$\dagger$
$\longrightarrow$
bag(N+1,0,ϕ,small),current(N+1)

$\|$

N,W,A:
bag(N,W,A,medium)$\dagger$, 11 < W

$\longrightarrow$
bag(N,W,A,small)
||
N,W,A:
[$\forall$ I :: $\neg$grocery(I,medium)], bag(N,W,A,medium)†
$\longrightarrow$
bag(N,W,A,small)
||
I: grocery(I,small) $\longrightarrow$ SM;
] *end transaction types*

initialization

grocery(glop,small);
grocery(popsicles,small);
grocery(bread,medium);
grocery(ice-cream,medium);
grocery(potato-chips,medium);
grocery(granola,large);
grocery(paper-towels,large);
grocery(pepsi,large);

current(1);

bag(1,0,ϕ,large);

LG; MD; SM.