Report Number: WUCS-91-17

1991-12-01

Formal Derivation of Rule-Based Program

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

This paper describes a formal approach to developing concurrent rule-based programs. Our program derivation strategy starts with a formal specification of the problem. Specification refinement is used to generate an initial version of the program. Program refinement is then applied to produce a highly concurrent and efficient version of the same program. Techniques for the deriving concurrent programs through either specification or program refinement have been described in previous literature. The main contribution of this paper consists of extending the applicability of these techniques to a broad class of rule-based programs. The derivation process is supported by a powerful... Read complete abstract on page 2.

Formal Derivation of Rule-Based Program

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

Complete Abstract:

This paper describes a formal approach to developing concurrent rule-based programs. Our program derivation strategy starts with a formal specification of the problem. Specification refinement is used to generate an initial version of the program. Program refinement is then applied to produce a highly concurrent and efficient version of the same program. Techniques for the deriving concurrent programs through either specification or program refinement have been described in previous literature. The main contribution of this paper consists of extending the applicability of these techniques to a broad class of rule-based programs. The derivation process is supported by a powerful proof logic, a logic that recently has been extended to cover rule-based programs. The presentation centers around a rigorous and systematic derivation of a concurrent rule-based solution to a classic problem.

Formal Derivation of Rule-Based Programs

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

WUCS-91-17

December 1991

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

To appear in IEEE Transactions on Software Engineering.

Abstract

This paper describes a formal approach to developing concurrent rule-based programs. Our program derivation strategy starts with a formal specification of the problem. Specification refinement is used to generate an initial version of the program. Program refinement is then applied to produce a highly concurrent and efficient version of the same program. Techniques for deriving concurrent programs through either specification or program refinement have been described in previous literature. The main contribution of this paper consists of extending the applicability of these techniques to a broad class of rule-based programs. The derivation process is supported by a powerful proof logic, a logic that recently has been extended to cover rule-based programs. The presentation centers around a rigorous and systematic derivation of a concurrent rule-based solution to a classic problem.

Acknolwedgements: The first author is supported by the National Science Foundation under the Grant CCR-9015677. The Government has certain rights in this material. The second and third authors are supported by the Center for Intelligent Computer Systems at Washington University, whose primary sponsors are McDonnell Douglas Corporation and Southwestern Bell Corporation.

Keywords: concurrency, formal verification, parallel rule-based programming, program derivation, stepwise refinement.

Correspondence: All communication regarding this paper should be addressed to

Dr. Gruia-Catalin Roman (314) 935-6190
Department of Computer Science •
Washington University •
509 Bryan Hall, Campus Box 1045 •
One Brookings Drive roman@CS.WUSTL.edu
St. Louis, MO 63130-4899 fax: (314) 935-7302

or

Rosanne F. Gamble (314) 935-7528
Department of Computer Science •
Washington University •
509 Bryan Hall, Campus Box 1045 •
One Brookings Drive rfg@CS.WUSTL.edu
St. Louis, MO 63130-4899 fax: (314) 935-7302

Formal Derivation of Rule-Based Programs

Gruia-Catalin Roman
Rosanne F. Gamble
William E. Ball

Department of Computer Science
Washington University
509 Bryan Hall, Campus Box 1045
One Brookings Drive
St. Louis, MO 63130-4899

1 Introduction

Rule-based programming has become one of the dominant computing paradigms within the artificial intelligence community, particularly in the expert systems arena. To a large extent, this popularity is due to the intuitive appeal of specifying computations as sets of rules. Success, however, has generated an increased demand for computing efficiency, particularly in using rule-based programming for application areas involving real-time decision processes. It is generally agreed that parallel processing offers the only hope for achieving significant execution speed-up [10,17]. Throughout this paper we use the term concurrency to refer to the potential for parallelism inherent in the logical structure of a program. We reserve the term parallelism to refer to implementations that employ hardware consisting of multiple processing units. Our research is concerned with promoting concurrency in rule-based programs, i.e., with enhancing the opportunities for parallel matching and firing of rules whether the implementation is parallel or not. To date, most efforts directed toward the parallel processing of expert systems has followed a relatively conservative strategy [11,12]. They start with existing rule-based programs and attempt to identify parallel algorithms for key functions of the run-time system such as the matching and firing of rules. Our strategy both complements and competes with these approaches. The complementary aspect is a direct consequence of the fact that rule-based programs written with an eye towards concurrency are less likely to impose sequential dependencies that could undermine parallel implementations. The competitive aspect stems from the fact that the very nature of rule-based programming may be altered and completely new implementation strategies may become feasible when one injects the concern for concurrency into the program derivation process.

Program derivation refers to a systematic formal process of constructing correct programs from their specifications, typically through some form of stepwise refinement. Thanks to work by Dijkstra [6], Gries [9], and many others, this approach has reached high degrees of sophistication and formality in the arena of sequential programming. The introduction of similar techniques to the area of concurrent programming is a relatively recent development. Chandy and Misra's work on UNITY [4] advocates an approach in which a formal specification of the problem is gradually refined up to the point when the specification is restrictive enough as to suggest a trivial translation into a concurrent program. An alternate approach is offered by work on action systems. Back and Sere [1] start with an initial (mostly sequential) program and refine it into an efficient concurrent one. In this paper we show that a combination of specification and program refinement may be applied to deriving efficient concurrent rule-based programs. We employ specification refinement to generate an initial rule-based program that is later refined into a program that is highly concurrent and efficient. The approach is targeted to rule-based programs that terminate.

The program notation and proof logic used in this paper are those of Swarm [15], a concurrency model in which all the entities that make up the program state have a tuple-like representation and state transitions, called transactions, are described using a rule-like notation. Swarm is uniquely appropriate for this task. Its proof logic [5] is a direct extension of the UNITY proof logic. Consequently, specification refinement techniques used in UNITY are also applicable to Swarm. Moreover, the Swarm notation is very close to common rule-based languages, such as OPS5 [7]. This facilitates the integration of techniques used in concurrent programming into the rule-based programming arena.

The remainder of the paper consists of five main parts followed by a discussion and conclusions. Section 2 introduces Swarm and programming notation used throughout the paper. Section 3 summarizes the proof logic for Swarm. The use of assertions to specify rule-based programs is illustrated in Section 4 on a typical artificial intelligence textbook problem, grocery bagging. The published programming solution [19] relies on conflict resolution for tasking and rule-ordering, and no speed-up would be gained if executed in parallel on available parallel productions system models, such as those proposed by Ishida and Stolfo [11], and Schmolze [17]. Section 5 presents a systematic formal derivation of a highly concurrent version of the this program without reliance on traditional conflict resolution. Section 6 gives the results of executing the derived program in the parallel production system PARS [17], which is also described in this section.

2 Notation

Swarm [15] belongs to a class of languages and models that use tuple-based communication. Other languages and models in this class are Linda [3], Associons [14], and GAMMA [2]. In this section we present a brief overview of the Swarm notation and its relation to traditional rule-based programming notation.

The dataspace. In Swarm, the entire computation state is captured by a set of tuple-like entities called the dataspace. For the purpose of this paper, the dataspace is partitioned into a tuple space, which corresponds working memory, and a transaction space, which corresponds to the knowledge base.

Working memory. The tuple space consists of a set of data tuples, which correspond to working memory elements. A data tuple assumes the form:

class_name(sequence_of_attribute_values)

e.g., *item(I,w,B,n)* may be a tuple representing a grocery item uniquely identified by *I*, of weight *w*, and packed in bag *B* after *(n-1)* other items;

Data tuples may be queried, deleted, and inserted. To query for the existence of a data tuple in the dataspace one simply treats tuple descriptions as predicates over the dataspace. Insertions are specified by fully instantiated tuples and deletions are specified by tagging a fully instantiated tuple with a dagger (†). Some examples of queries and actions relating to data tuples one can specify in Swarm follow:

- $[\forall$ I,w,B,n : item(I,w,B,n) :: B = 0 $\wedge$ n = 0]^1– (only as a query) checks if all items are unbagged, i.e., the bag identifier is zero;
- item(I,w,0,0)† – (as an action) deletes this item from the dataspace.
- item(I,w,0,0) – (as an action) inserts this item into the dataspace.

Production memory. The transaction space consists of a set of transactions. A simple transaction is analogous to a rule found in rule-based programming languages, and is defined in terms of a query followed by an action list consisting of deletions and insertions. The query is considered as the LHS of the simple transaction and the action list is the RHS. For instance,

$$
\begin{array}{l}
\mathrm{B}, \mathrm{I}_1, \mathrm{w}_1, \mathrm{n}_1, \mathrm{I}_2, \mathrm{w}_2, \mathrm{n}_2 : \\
\quad \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_1) \wedge \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_2) \wedge (\mathrm{w}_1 > \mathrm{w}_2) \wedge (\mathrm{n}_1 > \mathrm{n}_2) \\
\quad \longrightarrow \\
\quad \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_1)\dagger, \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_2)\dagger, \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_2), \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_1)
\end{array}
$$

[1] This three part notation is 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.

states that two groceries items that have been packed in the same bag with the heavier one on top of the lighter one are subject to a position exchange by deleting the old instances of the two item descriptions and by inserting new ones. The query is an arbitrary predicate that may involve testing for the presence or absence of data tuples. A successful query binds the variables listed before the query (existentially quantified by implication) to values used to compute the dataspace deletions and insertions. All deletions and assertions must contain bounded variables. Deletions always precede insertions, and it is acceptable for a transaction to attempt to delete an instantiated tuple that does not exist in working memory. Such deletions have no effect and do not represent semantic errors. If the query evaluates to false, no explicit deletions or insertions are performed.

Commas may be used inside the query as shorthand for the logical *and* ($\wedge$) and the order in which deletions and insertions are listed is immaterial. Actually, when the tuples being deleted are present in the query part, their deletion can be marked by daggers inside the query (only as a shorthand notation). Using these conventions the simple transaction above becomes

$$
\begin{array}{l}
\mathrm{B}, \mathrm{I}_1, \mathrm{w}_1, \mathrm{n}_1, \mathrm{I}_2, \mathrm{w}_2, \mathrm{n}_2 : \\
\quad \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_1)\dagger,\ \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_2)\dagger,\ (\mathrm{w}_1 > \mathrm{w}_2),\ (\mathrm{n}_1 > \mathrm{n}_2) \\
\quad \longrightarrow \\
\quad \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_2),\ \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_1)
\end{array}
$$

Naming transaction instances and classes. In Swarm the simple transaction above can be parameterized with respect to B and can be given a name:

$$
\begin{array}{l}
\mathrm{Swap}(\mathrm{B}) \equiv \\
\quad \mathrm{I}_1, \mathrm{w}_1, \mathrm{n}_1, \mathrm{I}_2, \mathrm{w}_2, \mathrm{n}_2 : \\
\qquad \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_1)\dagger,\ \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_2)\dagger,\ (\mathrm{w}_1 > \mathrm{w}_2),\ (\mathrm{n}_1 > \mathrm{n}_2) \\
\qquad \longrightarrow \\
\qquad \mathrm{item}(\mathrm{I}_1, \mathrm{w}_1, \mathrm{B}, \mathrm{n}_2),\ \mathrm{item}(\mathrm{I}_2, \mathrm{w}_2, \mathrm{B}, \mathrm{n}_1)
\end{array}
$$

This becomes a transaction type or transaction class definition, whose parameters (if any) are attribute values. Actually, every rule used in a Swarm program must be an instance of one of a finite set of transaction classes associated with the program. The transaction space contains transaction instances (henceforth simply called transactions) that can be executed by the program at a particular point in the computation. For example, a transaction of class *Swap*, such as *Swap(1)*, represents a specific transaction that may be included in the transaction space at initialization or during execution. Since transaction names in the transaction space are superficially indistinguishable from data tuples in the tuple space, Swarm allows queries and actions to refer to both data tuples and transactions, except that the action list may not include deletions of transactions. The technical reasons for this restriction will become apparent later in the paper. A query that tests for the presence or absence of a transaction in the transaction space is successful if there exists a desired instance of the transaction class in the transaction space. This test does not evaluate the transaction instance. The distinction between a transaction type definition and a transaction in Swarm represents another point of departure from current rule-based languages.

Transaction selection and execution. The transaction space of a Swarm program consists of a set of transactions. For rule-based programs expressed in Swarm, fairness requires that each transaction in the transaction space is eventually selected and executed (atomically), according to its class definition. The transaction selection is done prior to the evaluation of its query and is based simply on the fact that the transaction exists in the transaction space. A transaction executes any time it is selected. If the query does not succeed, the transaction is deleted from the transaction space. If its query is successful the deletions and insertions listed in the action list of the transaction are performed. The transaction is deleted implicitly unless it explicitly reinserts itself in the transaction space. Since transactions may be deleted and inserted, the transaction space is dynamic. In general, the transaction space is continuously updated by executing transactions that insert new transactions. Of course, the transaction space may be kept the same by reinserting each transaction after every execution. A Swarm program terminates when there are no more transactions in the transaction space.

Composition of simple transactions. In traditional rule-based languages, such as OPS5, rules cannot be combined to create new rules. In Swarm, however, the ||-operator, borrowed from UNITY, may be used to combine several simple transactions into a single complex transaction. A complex transaction is defined using a class name and parameters in the same manner shown for a simple transaction. For instance, a transaction that swaps out-of-order items in a bag B can be composed with a transaction that counts the number of successful swaps.

```
Swap_and_Count(B) ≡
    I1, w1, n1, I2, w2, n2 :
        item(I1, w1, B, n1)†, item(I2, w2, B, n2)†, (w1 > w2), (n1 > n2)
        ⟶
        item(I1, w1, B, n2), item(I2, w2, B, n1)
||  k :
        swapcount(k)† ⟶ swapcount(k+1)
```

The simple transactions making up a complex transaction are called subtransactions. When an instance of a complex transaction is chosen from the transaction space, all the subtransaction queries are applied together but only those subtransactions whose queries are successful contribute deletions and insertions. The combined deletions of successful subtransactions are performed simultaneously and are followed by the combined assertions of the same subtransactions. If the ||-operator is used without restriction, the resulting transaction may not be equivalent to any serial execution of its component subtransactions since all queries precede all deletions which, in turn, precede all insertions.

There are two problems with the above *Swap_and_Count(B)* transaction. The first problem is that because the queries of the subtransactions are performed simultaneously, it is possible for the second subtransaction query to be successful when the first subtransaction query is not. The result is that if there are no out of order items in the bag *B*, the count is still increased. The second problem is that *Swap_and_Count(B)* executes once and deletes itself. Thus, the program can terminate with items incorrectly ordered in a bag. To solve the problem, the second subtransaction should succeed only if the first subtransaction succeeds, and the transaction should reinsert itself as long as it finds out of order items in bag *B*. This requires strengthening the query on the second subtransaction. Swarm provides a very compact notation for doing this.

Notational convenience with special queries. In Swarm, the correct version of *Swap_and_Count(B)* may be stated very compactly as

```
Swap_and_Count(B) ≡
    I1, w1, n1, I2, w2, n2 :
        item(I1, w1, B, n1)†, item(I2, w2, B, n2)†, (w1 > w2), (n1 > n2)
        ⟶
        item(I1, w1, B, n2), item(I2, w2, B, n1)
||  k :
        OR, swapcount(k)† ⟶ swapcount(k+1), Swap_and_Count(B)
```

The special predicate, **OR**, succeeds by definition, whenever some other query appearing in the same transaction is successful, and this query makes no reference to any special built-in predicates. Such queries are called *regular*, while those that utilize special predicates are called *special* queries. Besides **OR**, other special predicates are **AND**, **NAND**, **NOR**, and **TRUE** with the meaning *all, not all, none,* and *no matter how many* of the regular queries appearing in the same transaction succeed. Throughout this paper, the special queries provide only notational convenience.[2] They are most often used either to force the recreation of the transaction whenever the query fails or to prevent recreation of the transaction when it is no longer useful.

Initialization section. Each program in Swarm must have a section that defines the initial configuration of the dataspace. For instance, one initial configuration of a program using *Swap_and_Count(B)* may involve *M* items and a transaction that reorders the items in bag number 3:

[2] These special queries have other purposes when used with features of Swarm that are not presented in this paper.

$[\ I : 1 \leq I \leq M :: \text{item(I,weight(I),bag(I),position(I)), Swap\_and\_Count(3), swapcount(0)}\ ]$

in which *weight(I)*, *bag(I)*, and *position(I)* are functions that map *I* to a weight value, bag number, and position, respectively. The three-part construct used above is called an object generator. *I* is a dummy variable that is restricted to ranging between 1 and some constant value *M*. For each valid value of *I* the generator contributes two data tuples *item(I,weight(I),bag(I),position(I))* and *swapcount(0)* and a transaction *Swap_and_Count(3)*. Since the net product is a set, object duplication is harmless.

3 Proof Logic Overview

Our program derivation methodology presupposes the ability to specify the operational details and formal properties of the program under development and to formalize the functional requirements imposed by the application. Section 2 gave an overview of the Swarm notation that is used to describe the structure and behavior of rule-based programs resulting from the application of our method. Safety and progress properties of programs are specified and verified using the Swarm proof logic [5] summarized in this section. The same proof logic is used to define an initial program specification. This is accomplished, as shown in Section 4, by constructing a sufficiently complete assertional-style characterization of the class of programs that represent acceptable realizations of the particular application. The Swarm proof logic follows the notational conventions for UNITY [4]. We use Hoare-style assertions of the form $\{p\}\ t\ \{q\}$ where p and q are predicates over the combined tuple space and transaction space (i.e., the dataspace) and t is a transaction. Properties and inference rules are often written without explicit quantification; they are universally quantified over all the values of the free variables occurring in them. The notation $[t]$[3] denotes the predicate "transaction t is in the transaction space", **TRS** denotes the set of all possible transactions (not a specific transaction space), and *INIT* denotes the initial state of the program. The proof rules for the subset of Swarm used in this paper are summarized in Figure 1. The first use of these concepts appears in the next section, where we elaborate the specifications of a sample problem.

4 Formal Specification

In this section we introduce and give a formal specification of the problem used to illustrate our approach to formal derivation of rule-based programs. *Bagger* is a rule-based program described by Winston in [19]. More detail of this particular formulation of the problem is given in Section 6. It expresses the desired way in which grocery items should be packed into bags. The program must pack all unbagged items according to their weight, with the heavier items preceding the lighter ones in each bag. A bag must only be created when it is needed, and its weight must not exceed some predetermined maximum value. The program must terminate when all items have been bagged.

4.1 Representation

The very first decision one must take in building a formal specification is the choice of representation for the entities manipulated by the program. This decision has important implications on the simplicity and conciseness of the formal specification. Moreover, in the absence of data refinement, the representation has a direct bearing also on the degree of concurrency achievable by programs derived from a particular specification. In this paper, however, we are not concerned with heuristics for constructing "good" specifications. We assume that a valid and adequate specification exists and concentrate our attention on a process for deriving concurrent programs that meet that specification. Our choice regarding the representation of items is fine grained because such a representation is less likely to restrict the opportunities for concurrency.

[3]Throughout the paper we simply use t in place of $[t]$ in predicates dealing with the existence of transaction t.

1. $\{p\}\ t\ \{q\}$

   Whenever the precondition p is *true* and t is a transaction in the transaction space, all dataspaces that can result from executing t satisfy postcondition q.

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

   If p is *true* at some point in the computation and q is not, then, after the execution of any single transaction, p remains *true* or q becomes *true*.

3. **stable** $p \ \equiv\ p$ **unless** *false*

   If p becomes *true*, it remains *true* forever.

4. **invariant** $p \equiv (INIT \Rightarrow p) \wedge (\textbf{stable}\ p)$
5. **constant** $p \equiv (\textbf{stable}\ p\ ) \wedge (\textbf{stable}\ \neg p\ )$

   Invariants are properties that are *true* at all points in the computation, written **inv** p and **const** p.

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

   For p **ensures** q to be *true*, there must exist a transaction t in the transaction space such that t will establish q when executed from a state in which $p \wedge \neg q$ holds. The requirement $p \wedge \neg q \Rightarrow [t]$ generalizes the UNITY definition of **ensures** to accommodate Swarm's dynamic creation of transactions. The second part of the definition guarantees q will eventually become *true*. This follows from the fact that a transaction can be removed from the dataspace *only if it executes*; the fairness assumption guarantees that a transaction will eventually be selected and executed.

7. $p \longmapsto q$ (Read p **leads-to** q.)

   Once p becomes *true*, q will eventually become *true*, but p is not guaranteed to remain *true* until q becomes *true*. As in UNITY, the assertion $p \longmapsto q$ is *true* if and only if it can be derived by a finite number of applications of the following inference rules:

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

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

8. **TERM** $\equiv [\forall\ t : t \in \textbf{TRS} :: \neg[t]]$.

   Swarm programs *terminate* when the transaction space is empty.

Figure 1: A subset of the Swarm proof logic.

The universe of grocery items forms a class whose members assume the form *item(I,w,B,n)*, where *item* is the class name and the parentheses hold a sequence of values, one for each attribute of the class. I is the unique identifier of the item, w is the weight of the item, B is the unique identifier of the bag in which the item is packed, and n is the position of the item in its bag. If B is zero, then the item is considered unbagged. Regarding the bags, as already shown, their representation is distributed across that of the items they hold. Furthermore, both bags and items are restricted in weight to a maximum value H. Given this representation, we can turn next to developing a formal specification from which the program will be derived. All variables appearing free are universally quantified.

4.2 Proof Obligations

The constraints we want to impose over the representation of grocery items have to do with the desirability of being able to distinguish items by using unique identifiers and the requirement that they do not change weight along the way:

$$\textbf{inv } [\Sigma \text{ w,B,n} : \text{item(I,w,B,n)} :: 1]^4 \leq 1 \tag{S1}$$

$$\textbf{const } [\exists \text{ B,n} :: \text{item(I,w,B,n)}] \tag{S2}$$

The fact that bags have a distributed representation results in the necessity to impose constraints over working memory. In particular, bags are not permitted to exceed a maximum weight capacity H and must have contiguous identifiers:

$$\textbf{inv } \text{WgBag(B)} \leq \text{H} \tag{S3}$$

$$\textbf{inv } (\text{WgBag}(\text{B}_2) > 0) \wedge (\text{B}_2 > \text{B}_1 > 0) \Rightarrow \text{WgBag}(\text{B}_1) > 0 \tag{S4}$$

where

$$\text{WgBag(B)} \equiv [\Sigma \text{ I,w,n} : \text{item(I,w,B,n)} :: \text{w}] \tag{D1}$$

Next, we impose restrictions on the bagging policy of the program. First, once an item is placed in the bag, it cannot be removed and it cannot change positions within the bag. This constraint maps directly to the assertion

$$\textbf{stable } \text{item(I,w,B,n)} \wedge (\text{B} = \text{B}_0 > 0) \wedge (\text{n} = \text{n}_0 > 0) \tag{S5}$$

Second, we require that bagged items have non-zero positions and that no two items occupy the same position in the same bag. A third restriction is that items in the same bag are ordered according to their weights, with heavier items packed before lighter ones. Finally, if one item is in the first position of some bag, then all bags created prior to this bag (as determined by the identification numbers of the bags) cannot hold the item. This guarantees that bags are created as needed. All these requirements are captured by the following invariant:

$$\begin{aligned}
&\textbf{inv } [\forall \text{I}_1, \text{I}_2, \text{w}_1, \text{w}_2, \text{B}_1, \text{B}_2, \text{n}_1, \text{n}_2 : \\
&\qquad \text{item}(\text{I}_1, \text{w}_1, \text{B}_1, \text{n}_1) \wedge \text{item}(\text{I}_2, \text{w}_2, \text{B}_2, \text{n}_2) \wedge (\text{B}_1 > 0) \wedge (\text{B}_2 > 0) :: \\
&\qquad\qquad (\text{n}_1 > 0) \wedge (\text{n}_2 > 0) \\
&\qquad\quad \wedge ((\text{I}_1 = \text{I}_2) \Leftrightarrow (\text{B}_1 = \text{B}_2) \wedge (\text{n}_1 = \text{n}_2)) \\
&\qquad\quad \wedge ((\text{I}_1 \neq \text{I}_2) \wedge (\text{w}_1 > \text{w}_2) \wedge (\text{B}_1 = \text{B}_2) \Rightarrow (\text{n}_1 < \text{n}_2)) \\
&\qquad\quad \wedge ((\text{I}_1 \neq \text{I}_2) \wedge (\text{B}_1 < \text{B}_2) \wedge (\text{n}_2 = 1) \Rightarrow (\text{WgBag}(\text{B}_1) + \text{w}_2) > H)]
\end{aligned} \tag{S6}$$

[4] The three part notation is extended to allow operators in place of quantifiers. In such cases, the third part of the notation defines a list of operands over which the operator is applied. Here we sum the weights of items contained in a particular bag. If the list of operands is empty, the result is zero.

Given these integrity and policy statements, the problem to be solved is stated very simply: *given a finite set of unbagged grocery items with identifiers in the range 1 to N and weights in the range 1 to H, the program terminates with all items packed.* This is captured by the following set of conditions:

$$\text{GINIT} \longmapsto \text{GPOST} \qquad \text{(P1)}$$
$$\text{GINIT} \equiv [\forall\ \text{I,w,B,n} : \text{item(I,w,B,n)} :: (1 \le \text{w} \le \text{H}) \wedge (1 \le \text{I} \le \text{N}) \wedge (\text{B} = 0) \wedge (\text{n} = 0)] \qquad \text{(D2)}$$
$$\text{GPOST} \equiv [\forall\ \text{I,w,B,n} : \text{item(I,w,B,n)} :: \text{B} > 0] \qquad \text{(D3)}$$

where *GINIT*, the initial state of the data, requires that all items start unbagged and have weight less than *H* and *GPOST*, the desired outcome of the computation, requires that all items are bagged.

Of course, *GINIT* must be established initially in the program, and once the desired outcome of the computation is reached the program eventually terminates. Therefore,

$$\text{INIT} \Rightarrow \text{GINIT} \qquad \text{(C1)}$$
$$\text{stable GPOST} \qquad \text{(S7)}$$
$$\text{GPOST} \longmapsto \text{TERM} \qquad \text{(P2)}$$

INIT is the initial state of the program including the input data and *TERM* is the termination condition of all Swarm programs as defined in Figure 1. The property (S7) is implied by the stronger requirement (S5). Termination, however, is an additional requirement independent of all others listed so far.

5 Program Derivation

In this section we develop a concurrent version of *Bagger*. We start with the formal specification given in Section 4 and apply to it, throughout Section 5.1, a series of refinements in a manner reminiscent of the UNITY derivation process. The basic strategy behind the refinement steps is as follows. First, we introduce a way of measuring global progress toward the desired outcome of the computation. Second, we express the global measure in terms of simpler measures dealing with subproblems suggested by a proposed solution strategy. Third, we seek ways of accomplishing progress toward solving the individual subproblems. Finally, we generate a program that can be easily shown to meet the refined specification. The resulting program exhibits a significant degree of concurrency, has a static set of rules, is correct except that it is non-terminating, and makes indiscriminate use of highly complex queries.

Program refinement, detailed in Section 5.2, starts with this program and attempts to generate a new program that offers greater opportunities for efficient parallel execution. First, we attempt to maximize the degree of concurrency achievable by the program. This involves replacing single transactions with groups that can perform the same computational task but possibly in parallel. Second, we address the issue of termination. Third, we apply a series of heuristics to decrease the complexity of the queries employed by the program. Finally, we take advantage of dynamic transaction creation and attempt to continually update the contents of the transaction space to ensure that transactions are present only when they can contribute to reaching the computational goals at hand and not sooner.

5.1 Stepwise Refinement of the Specification

Having given a formal specification for *Bagger*, this section is concerned with refining the specification to the point that an initial Swarm program can be constructed directly from the refined specification. Successive refinements of the main progress property (P1) are performed by gradually factoring in elements of an emerging solution strategy. The discovery of the solution strategy is generally accepted to be a creative step. Verifying that the original specification is satisfied by the refinement is a formal step involving an application of the Swarm proof logic.

5.1.1 Refinement 1: measuring progress

We see refinement as a reversal of the verification process. In other words, given a property that needs to be refined we pose the question how one might prove such a property. For instance, proving a property such as

$$\text{GINIT} \longmapsto \text{GPOST} \tag{P1}$$

usually requires the introduction of a variant function needed to construct an inductive proof. For *Bagger*, the number of unbagged grocery items

$$\text{NrOut} \equiv [\Sigma\ \text{I,w,n} : \text{item(I,w,0,n)} :: 1] \tag{D4}$$

is the most obvious way to measure progress. From its definition and from properties (S5) and (S1), which state that bagged items are stable and that items are unique, we can establish that *NrOut* is non-increasing and well-founded

$$\textbf{stable}\ \text{NrOut} < \text{k} \tag{S8}$$
$$\textbf{inv}\ \text{NrOut} \geq 0 \tag{S9}$$

The requirement (P1) becomes a consequent of the three conditions

$$\text{GINIT} \Rightarrow \text{NrOut} \leq \text{N} \tag{S10}$$
$$\text{NrOut} = \text{k} \wedge \text{k} > 0 \longmapsto \text{NrOut} < \text{k} \tag{P3}$$
$$\text{NrOut} = 0 \Rightarrow \text{GPOST} \tag{S11}$$

Property (S10) follows from the definitions of *NrOut* and *GINIT*. Property (S11) may be proven using (S1), which requires an item to be either bagged or unbagged, but not both. Property (P3) replaces the requirement (P1) in the derivation.

5.1.2 Refinement 2: introducing local measures

If one is interested in generating a sequential program, the variant function *NrOut* is a good choice. Items could be bagged one at a time. However, when seeking concurrency one generally needs to discover areas of localized progress that together contribute toward achieving global progress. In the case of *Bagger*, we observe that *NrOut* can be replaced by an equivalent measure consisting of a vector whose components are the number of unbagged items corresponding to each conceivable weight *w*. By introducing the function

$$\text{NrWg(w)} \equiv [\Sigma\ \text{I,n} : \text{item(I,w,0,n)} :: 1] \tag{D5}$$

we can replace the condition (P3) by

$$\text{NrWg(w)} = \text{k} \wedge \text{k} > 0 \longmapsto \text{NrWg(w)} < \text{k} \tag{P4}$$

with *w* being universally quantified, by convention. Properties similar to (S8) and (S9) also hold for *NrWg(w)*

$$\textbf{stable}\ \text{NrWg(w)} < \text{k} \tag{S12}$$
$$\textbf{inv}\ \text{NrWg(w)} \geq 0 \tag{S13}$$

and

$$\text{NrOut} = \text{k} \Leftrightarrow [\Sigma\ \text{w} :: \text{NrWg(w)}] = \text{k} \tag{S14}$$

is a direct consequence of the definitions (D4) and (D5). Since, according to (P4), items in every weight category are eventually bagged, (P3) clearly holds. Therefore, (P4) replaces (P3) in the derivation.

5.1.3 Refinement 3: shaping local progress

Our next task is to refine the relatively abstract property (P4) so as to make explicit the way in which packing is carried out. Two plausible solution strategies are: (i) pack the heaviest items before considering unpacked items in the lower weight categories, and (ii) pack in each bag the heaviest item the bag can hold. Both solutions are consistent with the problem specification but the latter strategy provides more opportunities for exploiting concurrency by packing items of differing weights at the same time. Under this strategy, *NrWg(w)* is expected to decrease whenever *w* is the largest weight among all unbagged items or *w* is the weight of the largest item that fits in a particular bag *B*.

We define two predicates, *MaxWg(w)* and *MaxFitWg(w)*, to capture the two cases

$$\text{MaxWg(w)} \equiv (\text{w} = [\max \text{I, w'} : \text{item(I,w',0,0)} :: \text{w'}])^{5} \tag{D6}$$
$$\text{MaxFitWg(w)} \equiv [\exists\ \text{B} :: \text{w} = [\max \text{I',w'} : \text{item(I',w',0,0)} \wedge \text{Fit(B,w')} :: \text{w'}]\ ] \tag{D7}$$

where

$$\text{Fit(B,w)} \equiv (\text{w} \leq \text{H - WgBag(B)}) \wedge [\exists\ \text{I',w',n'} : \text{item(I',w',B,n')} :: \text{B} > 0] \tag{D8}$$

Using transitivity, we substitute for (P4) the properties

$$\text{NrWg(w)} = \text{k} \wedge \text{k} > 0 \longmapsto \text{NrWg(w)} = \text{k} \wedge (\text{MaxFitWg(w)} \vee \text{MaxWg(w)}) \tag{P5}$$
$$\text{NrWg(w)} = \text{k} \wedge (\text{MaxFitWg(w)} \vee \text{MaxWg(w)}) \longmapsto \text{NrWg(w)} < \text{k} \tag{P6}$$

Since property (P5) (for a particular value of *w*) is provable given that (P6) holds for all values of *w*, we need to consider further only the property (P6).

The structure of the LHS of property (P6) suggests its rewriting into two distinct progress properties; one corresponding to *MaxFitWg(w)* being *true* whether or not *MaxWg(w)* is *true*, and the other corresponding to the case in which *MaxWg(w)* is *true* but *MaxFitWg(w)* is not.

$$\text{NrWg(w)} = \text{k} \wedge \text{MaxFitWg(w)} \longmapsto \text{NrWg(w)} < \text{k} \tag{P7}$$
$$\text{NrWg(w)} = \text{k} \wedge \neg\text{MaxFitWg(w)} \wedge \text{MaxWg(w)} \longmapsto \text{NrWg(w)} < \text{k} \tag{P8}$$

In the first case, there are bags that need items of weight *w* and as a result some of them get packed. In the second case, there are no bags that can hold items of weight *w* and some new bag must be created for this purpose. To accomplish this latter goal further refinement of (P8) must be considered.

For *(¬MaxFitWg(w) ∧ MaxWg(w))* to be *true*, *w* must be the heaviest weight among unpacked items and cannot fit in any existing bag, i.e.,

$$\text{NoFit(w)} \equiv [\forall\ \text{B} : [\exists\ \text{I,w,n} :: \text{item(I,w,B,n)}] \wedge \text{B} > 0 :: \text{H} < \text{WgBag(B)} + \text{w}] \tag{D9}$$

When this state occurs, a new bag must be created. By creating bags in this manner, (S6) in particular the portion that describes that bags are created as needed, is maintained. But, according to the invariant (S4) bags must be created in contiguous order as defined by the predicate *NextBag(B)* below.

$$\text{NextBag(B)} \equiv (\text{B} = [\max \text{I,w, B',n} : \text{item(I,w,B',n)} :: \text{B'}] + 1) \tag{D10}$$

Using these two definitions we reformulate (P8) as

$$\begin{aligned}&\text{NrWg(w)} = \text{k} \wedge \text{NoFit(w)} \wedge \text{MaxWg(w)} \wedge \text{NextBag(B)}\\ &\longmapsto\\ &\text{NrWg(w)} < \text{k} \wedge \text{NextBag(B + 1)}\end{aligned} \tag{P9}$$

[5]Here we use the operator max, for maximum value. Its meaning here is take the maximum value of every w returned from an unbagged item." If there are no such items, the result is zero.

5.1.4 Refinement 4: generating an initial program

By now the specification has acquired sufficient detail to consider transforming it into a concrete program. Towards this aim, we must find transactions that realize the progress conditions (P7) and (P9) while satisfying the safety conditions that are also part of the specification. We postulate that the following two **ensures** properties hold in order to prove (P7) and (P9).

$$\mathrm{NrWg(w) = k \land MaxFitWg(w)}\ \textbf{ensures}\ \mathrm{NrWg(w) < k}$$

$$\begin{array}{l}\mathrm{NrWg(w) = k \land NoFit(w) \land MaxWg(w) \land NextBag(B)}\\ \textbf{ensures}\\ \mathrm{NrWg(w) < k \land NextBag(B + 1)}\end{array}$$

First we concentrate on (P7). By definition of **ensures**, any program that satisfies (P7) must include, for each weight *w*, a transaction, call it *Bag(w)*, which satisfies

$$\{\mathrm{NrWg(w) = k \land MaxFitWg(w)}\}\ \mathrm{Bag(w)}\ \{\mathrm{NrWg(w) < k}\} \qquad \text{(C2)}$$

One possible design choice is to have *Bag(w)* select some unbagged item *I* of weight *w* and some bag *B* that can hold items of weight at most *w* and pack the item *I* in the next available position in *B*. The following transaction definition captures this idea:

```
Bag(w) ≡
    I,B,n :
        MaxFitWg(w), BestFit(B,w), NextPos(B,n), item(I,w,0,0)†
        ⟶
        item(I,w,B,n)
||  :   TRUE ⟶ Bag(w)
```

BestFit(B,w), similar to *MaxFitWg(w)*, determines the weight of the item that could be packed next in *B* and *NextPos(B,n)* determines the next available packing position in bag *B*

$$\mathrm{BestFit(B,w) \equiv w = [max\ I',w' : item(I',w',0,0) \land Fit(B,w') :: w']} \qquad \text{(D11)}$$
$$\mathrm{NextPos(B,n) \equiv (n = [max\ I',w',n' : item(I',w',B,n') :: n'] + 1)} \qquad \text{(D12)}$$

Finally, by noting that

$$\mathrm{BestFit(B,w) \Rightarrow MaxFitWg(w)}$$

the definition for *Bag(w)* becomes

```
Bag(w) ≡
    I,B,n :
        BestFit(B,w), NextPos(B,n), item(I,w,0,0)†
        ⟶
        item(I,w,B,n)
||  :   TRUE ⟶ Bag(w)
```

The first subtransaction of *Bag(w)* does the packing while the second guarantees that, once created, the transaction continues to exist in the dataspace indefinitely. We handle the creation by requiring the existence of *Bag(w)* transactions at the start of the program, i.e.,

$$\text{INIT} \Rightarrow [\forall\, w : (1 \le w \le H) :: \text{Bag}(w)] \tag{C3}$$

At this point we are under the obligation to show that *Bag(w)* satisfies (C2), the **unless** properties implicit in (P7) and (P9), and the safety conditions (S1) through (S14). Fortunately, these obligations can be easily proven by considering only the transactions *Bag(w)* in isolation from the remainder of the program, still to be derived. For the sake of brevity the actual proofs are omitted. Here, as elsewhere in the paper, we left out the proof details for the sake of focusing on the derivation process rather than its low level mechanics.

Starting from (P9) and following the same line of reasoning we discover the need for transactions of type *Make_Bag* that satisfy the property

$$\begin{array}{l}\{\text{NrWg}(w) \wedge \text{NoFit}(w) \wedge \text{MaxWg}(w) \wedge \text{NextBag}(B)\} \\ \qquad \text{Make\_Bag}(B,w) \\ \{\text{NrWg}(w) < k \wedge \text{NextBag}(B+1)\}\end{array} \tag{C4}$$

One possible definition of *Make_Bag(B,w)* is

```
Make_Bag(B,w) ≡
    I :
        MaxWg(w), NoFit(w), NextBag(B), item(I,w,0,0)† ⟶ item(I,w,B,1)
||  :   TRUE ⟶ Make_Bag(B,w)
```

The requirement

$$\text{INIT} \Rightarrow [\forall\, w,B : (1 \le w \le H) \wedge (1 \le B \le N) :: \text{Make\_Bag}(B,w)] \tag{C5}$$

is added to guarantee the inclusion of these transactions in the initial dataspace configuration.

The resulting program consists of *H* transactions of type *Bag* and *H*N* transactions of type *Make_Bag*. The set of transactions is finite and constant. This first version of *Bagger* differs from a corresponding UNITY program only with respect to the fact that transactions are nondeterministic while conditional assignment statements are deterministic. This version is correct with respect to the specifications from Section 4 except that we ignored the termination requirement – we will return to it in a later section.

5.2 Stepwise Refinement of the Program

The program generated in Section 5.1 is the result of a series of specification refinements motivated by logical arguments that did not take into account the costs associated with executing individual transactions and the amount of concurrency ultimately achievable under the adopted solution strategy. Our experience to date strongly suggests that these concerns are more readily addressed through a program refinement process whose goals are to maximize concurrency and to increase efficiency. Successive program refinements alter the program while preserving its correctness with respect to the specification. For example, concurrency is enhanced by increasing the number of transactions that can perform useful work. Individual transactions are replaced by groups of transactions that carry out the same computational task possibly in parallel. Efficiency is improved by eliminating queries that examine large portions of the dataspace and by ensuring that transactions are present in the dataspace only when needed.

5.2.1 Refinement 1: splitting transactions

Our first refinement is concerned with ensuring that later optimizations are applied to a program that exhibits the maximum possible potential for concurrent execution, under the constraints of the solution strategy that we adopted in the previous section. The refinement involves only the transaction space which, for the time being, continues to be static, i.e., it contains all the transactions the program will ever need. The idea is to increase the number of transactions in the transaction space through a technique called *splitting*. Splitting takes advantage of the nondeterminism present in query satisfaction by replacing a single transaction with several transactions whose queries are satisfied by disjoint instantiations of the original query.

The simplest form of splitting entails the replacement of a variable bound in the query by a constant, which appears as a new parameter of the corresponding transaction class. The replaced variable must range over a finite set. Thus, the transaction space contains a new transaction for each possible instantiation of the variable. The technique is similar to constrained copying of rules used in some parallel implementations of rule-based programs [13] to improve run-time performance. Its use in program derivation shares the same general goal but in a very distinct context. The resulting program automatically satisfies the specification. The assertions that are part of the specification make no explicit references to the transaction space. Any program property preserved by the original transaction is also preserved by the transactions generated by splitting, since they perform only state transitions that were possible originally. Also, fairness is not affected because the number of new transactions is restricted to being finite.

In the absence of a particular implementation model, we define the concurrency $C_{T,s}$ exhibited by a transaction class T in a program state s to be the largest number of transactions of type T that can execute in parallel (i.e., as if they were connected by the ||-operator) without violating the program specification. $C_{T,s}$ is maximal with respect to the set of all conceivable instantiations of the respective transaction queries in the state s. For instance, the concurrency exhibited by transactions of type *Bag* is determined by the number of distinct weights that can be packed at one time

$$C_{Bag,s} = [\Sigma \text{ w} : 1 \leq \text{w} \leq \text{H} :: \min([\Sigma \text{ I} : \text{item(I,w,0,0)} :: 1], [\Sigma \text{ B} : \text{BestFit(B,w)} :: 1], 1)]$$

Each *Bag(w)* packs one item whenever there is at least one unbagged item of weight w and at least one bag in which to pack the item. For distinct values of w, all instances of *Bag(w)* can execute in parallel.

In order to increase the concurrency exhibited by the program we must be able to pack more than one item of weight w at a time. Looking at the definition of $C_{Bag,s}$ it is clear that, for a given weight w, the largest number of items we can hope to pack at one time is

$$\min([\Sigma \text{ I} : \text{item(I,w,0,0)} :: 1], [\Sigma \text{ B} : \text{BestFit(B,w)} :: 1])$$

no matter how large we make the third parameter of the *min* function, i.e., the number of available packing transactions. Maximal concurrency can be accomplished equally well by splitting *Bag(w)* either bag-wise as in

```
Bag_In(B,w) ≡
    I,n :
        BestFit(B,w), NextPos(B,n), item(I,w,0,0)†
        ⟶
        item(I,w,B,n)
||  :   TRUE ⟶ Bag_In(B,w)
```

or item-wise as in

```
Bag_Item(I,w) ≡
    B,n :
        BestFit(B,w), NextPos(B,n), item(I,w,0,0)†
        ⟶
        item(I,w,B,n)
||  :   TRUE ⟶ Bag_Item(I,w)
```

Choosing among these two alternatives must involve criteria other than maximizing concurrency. In the case of this application, one can reasonably expect to see many more items waiting to be packed than bags available for packing, especially since bags are created as needed. Therefore, many items of equal weight are likely to compete for few bags if item-wise splitting is done. Whereas, with bag-wise splitting, few bags are free to choose from a large selection of equal weight items. Also, our choice of data representation allows easy access to information about particular items, while information about bags must be computed by considering all the items present in each bag. For these reasons we opt in favor of bag-wise splitting. Hence, for each possible value of w (1 through H) and of B (1 through N), we create an instance of *Bag_In(B,w)* to replace *Bag(w)*.

Although splitting can only improve concurrency, unnecessary splitting should be avoided since it may complicate later program optimization steps that try to ensure that transactions are present only when needed. For instance, our current version of *Bagger* already includes $H*N$ transactions of type *Make_Bag*, yet bags must be created one at a time. In this case, splitting *Make_Bag* cannot improve concurrency.

5.2.2 Refinement 2: addressing termination

The goal of this step is to address the termination requirement (P2) by eliminating transactions whenever they are no longer needed and by showing that eventually all transactions become unnecessary. This portion of the derivation process is the first to take advantage of the dynamic nature of the Swarm transaction space, i.e., transactions may be deleted as a by-product of being executed. To guarantee termination we want to eliminate transactions that can no longer be executed. Up to now all transactions were created initially and existed forever. They had the form:

$$\mathrm{T} \equiv \mathrm{q} \longrightarrow \alpha \parallel \mathbf{TRUE} \longrightarrow \mathrm{T}$$

where q is the query part of the first subtransaction of T, and α is the action list of that subtransaction. The special query **TRUE** causes the transaction to be reasserted unconditionally. Our intent is to find some property r for which we can prove **stable** r and $r \Rightarrow \neg q$. This allows us to safely redefine T as

$$\mathrm{T} \equiv \mathrm{q} \longrightarrow \alpha \parallel \neg\mathrm{r} \longrightarrow \mathrm{T}$$

because the stability of r guarantees that q will never again become *true* and, therefore, T will never again be able to modify the dataspace. Of course, since T is created at initialization, T is guaranteed to exist any time q is *true*. The property r need only be strong enough to prove termination. For both transaction classes in *Bagger*, the property r is easily defined as

$$\mathrm{r(w)} \equiv [\forall\, \mathrm{I', w'} : \mathrm{item(I', w', 0, 0)} :: \mathrm{w'} \neq \mathrm{w}]$$

One can show that *r(w)* falsifies the query of the first subtransaction of *Bag_In(B,w)* and *Make_Bag(B,w)* by observing that $[\forall\, I', w' : item(I', w', 0, 0) :: w' \neq w] \Rightarrow \neg[\exists\, I :: item(I, w, 0, 0)]$. The stability of *r(w)* is due to (S2), which states that items have a constant weight and (S5), which states that bagged items are stable. The transaction definitions become

```
Bag_In(B,w) ≡
    I,n :
        BestFit(B,w), NextPos(B,n), item(I,w,0,0)†
        ⟶
        item(I,w,B,n)
||  I :
        item(I,w,0,0) ⟶ Bag_In(B,w)
```

```
Make_Bag(B,w) ≡
    I :
        MaxWg(w), NoFit(w), NextBag(B), item(I,w,0,0)†
        ⟶
        item(I,w,B,1)
||  I :
        item(I,w,0,0) ⟶ Make_Bag(B,w)
```

At this point, in order to conclude that the termination condition (P2) is met, we need to show is that *GPOST* $\Rightarrow$ $r(w)$. This is clearly so since when *GPOST* is established, there are no unbagged items left and *GPOST* is stable.

5.2.3 Refinement 3: reducing query complexity

It is a well-known fact within the expert system community that the pattern matching phase involved in query evaluation consumes a large portion of the total time needed to execute one cycle in a rule-based program [10]. Nevertheless, our program derivation process often used complex queries, whose evaluation is likely to burden even the fastest matching algorithm. The goal of this refinement step is to reduce the complexity of these queries, thus yielding a more efficient program. The basic mechanism we employ throughout this section is to create new, continuously-updated tuples that hold the values otherwise computed by complex queries. New safety properties involving these tuples are added to the specification in order to formalize processing obligations relating to the maintenance of such tuples. The same safety properties are used to prove that transactions, in which complex queries are replaced by references to these tuples, preserve the original specifications.

In Section 5.2.1, we made the decision to let individual bags be in control of the items packed in them, i.e., bag-wise parallelism, instead of letting individual items be in charge of the bag they were to occupy, i.e., item-wise parallelism. Intuitively, it is reasonable to expect that complex queries that change with respect to bags are easier to reduce than complex queries that involve changes to the set of items. This is exactly the case. The satisfaction of the queries *NextPos(B,n), WgBag(B), NextBag(B), Fit(B,w)* and *NoFit(w)* changes only when the state of a bag changes. References to these complex queries can be replaced by references to tuples that maintain the values of the original query. Such reductions will yield tuples that are only accessed by the corresponding bag. Whereas, the satisfaction of the queries *BestFit(B,w)* and *MaxWg(w)* changes when the state of the items changes. In order to maintain these state changes within a tuple, the tuple would have to be shared by all transactions. This representation would cause a serious decline in the available concurrency. A technique we use to reduce the need for such globally accessed data is to create an *approximation*, i.e., a tuple containing a value that is guaranteed to converge to the correct value. For example, the query *MaxWg(w)* in transaction *Make_Bag(B,w)* involves looking at the weight of all unbagged items to determine the largest weight. Using a single tuple to represent this exact weight would require every transaction that modified the set of unbagged items, to access and possibly modify this weight, creating a bottleneck. However, if we create a tuple that approximates this weight under the restriction that this tuples weight is always at most the actual largest weight and eventually converges to the actual weight, then only transactions of type *Make_Bag(B,w)* need to maintain the approximating tuple. Thus, for each of the complex queries involving the overall state of the items, a tuple will be used to approximate the value of the query.

Replacing queries by single tuple searches. To make the reduction more clear, we will first expand *BestFit(B,w)*. This is because the queries *Fit(B,w)* and *WgBag(B)* are hidden inside *BestFit(B,w)* in *Bag_In(B,w)*. We will return to the actual query for *BestFit(B,w)* later. The expanded *Bag_In(B,w)* transaction is

```
Bag_In(B,w) ≡
    I,n :
        w = [max I',w' : item(I',w',0,0) ∧ Fit(B,w') :: w'], NextPos(B,n), item(I,w,0,0)†
        ⟶
        item(I,w,B,n)
||  I : item(I,w,0,0) ⟶ Bag_In(B,w)
```

(Please recall that commas represent an alternate notation for the logical *and*.) We can redefine *Fit(B,w)* using a tuple, *capacity(B,c)*, where *c* represents the spare capacity of the bag, as

$$\text{Fit(B,w)} \equiv [\exists \text{ c : capacity(B,c) :: w} \leq \text{c}] \tag{D13}$$

provided the following invariant holds.

$$\textbf{inv} \text{ capacity(B,c)} \Leftrightarrow \text{c = H - WgBag(B)} \wedge [\exists \text{ I',w',n' : item(I',w',B,n') :: B} > 0] \tag{S15}$$

Although (D13) allows us to simplify considerably the query in *Bag_In(B,w)*, it adds new processing requirements dealing with the creation and updating of the new tuple. The tuple *capacity(B,c)* must be asserted into the tuple space the first time an item is packed in any bag. This is done by *Make_Bag(B,w)*. The tuple must be updated whenever the spare capacity changes. Thus, *Bag_In(B,w)* is responsible for updating *capacity(B,c)*.

Before showing the changes to *Bag_In(B,w)*, let us consider also the query *NextPos(B,n)*. A tuple *next_pos(B,n)* can be directly utilized provided the specifications are strengthened to include the following invariant.

$$\textbf{inv} \text{ next\_pos(B,n)} \Leftrightarrow \text{NextPos(B,n)} \tag{S16}$$

When bag *B* is created by filling its first position, *Make_Bag(B,w)* creates *next_pos(B,2)* and whenever an unbagged item of weight *w* is placed in bag *B*, *Bag_In(B,w)* replaces *next_pos(B,n)* with *next_pos(B,n+1)*. We now show the changes to *Bag_In(B,w)* caused by introducing *capacity(B,c)* and *next_pos(B,n)*. The changes to *Make_Bag(B,w)* are presented later when reduction of its queries is performed

```
Bag_In(B,w) ≡
    I,n,c :
        w = [max I',w' : item(I',w',0,0) ∧ w' ≤ c :: w'], capacity(B,c)†,
        next_pos(B,n)†, item(I,w,0,0)†
        ⟶
        item(I,w,B,n), capacity(B,c-w), next_pos(B,n+1)
||  I : item(I,w,0,0) ⟶ Bag_In(B,w)
```

We turn our attention to *Make_Bag(B,w)* to reduce the queries *NoFit(w)* and *NextBag(B)*. The reduction of the query *MaxWg(w)* will be discussed later. The only reduction that can be performed on *NoFit(w)* is to redefine the predicate using *capacity(B,c)*, but all existing bags must still be checked.

$$\text{NoFit(w)} \equiv [\forall \text{ B,c : capacity(B,c) :: c} < \text{w}] \tag{D14}$$

To reduce *NextBag(B)*, the tuple *next_bag(B)* can be introduced to keep track of the next bag to be created.

$$\textbf{inv} \text{ next\_bag(B)} \Leftrightarrow \text{NextBag(B)} \tag{S17}$$

Because initially the next bag to be created is the first bag, we require that $INIT \Rightarrow$ *next_bag*(1). Whenever the first item of weight *w* is placed in a bag *B*, *next_bag(B+1)* must be asserted by *Make_Bag(B,w)*. The resulting definition of *Make_Bag(B,w)*, including the changes from the reduction of *Bag_In(B,w)* is as follows.

```
Make_Bag(B,w) ≡
    I :
        MaxWg(w), [∀ B',c' : capacity(B',c') :: c' < w], next_bag(B)†, item(I,w,0,0)†
        ⟶
        item(I,w,B,1), next_bag(B+1), capacity(B,H-w), next_pos(B,2)
||  I : item(I,w,0,0) ⟶ Make_Bag(B,w)
```

Approximating queries by single tuple searches. We desire (as earlier) to replace by single tuples complex queries whose values change due to changes in the set of items. For example, ideally we want to include in the specification

$$\textbf{inv}\ \text{best\_fit(B,w)} \Leftrightarrow \text{BestFit(B,w)}$$

where *BestFit(B,w)* can be rewritten as

$$\text{BestFit(B,w)} \equiv [\exists\ c : \text{capacity(B,c)} :: w = [\max I', w' : \text{item(I',w',0,0)} \land w' \leq c :: w']]$$

Because bagging elsewhere may use up all of the items of weight *w*, this tuple would have to be shared by all *Bag_In* transactions, which must check its value with only one allowed access at a time. This type of shared tuple would severely limit the available concurrency. In an attempt to distribute access to shared information we create a tuple that approximates the global state for each local operation and eventually converges to that state, while obeying the specifications. Applying this technique, our solution is to make the tuple *best_fit(B,w)* approximate and gradually converge to *BestFit(B,w')*. We define convergence as

$$\textbf{inv}\ 1 \geq [\Sigma\ w : \text{best\_fit(B,w)} :: 1] \geq 0 \qquad \text{(S18)}$$
$$\textbf{inv}\ \text{best\_fit(B,w)} \land \text{BestFit(B,w')} \land \text{capacity(B,c)} \Rightarrow c \geq w \geq w' \qquad \text{(S19)}$$
$$\text{best\_fit(B,w)} \land \text{BestFit(B,w')} \land w > w' \longmapsto \text{best\_fit(B,w-1)} \qquad \text{(P10)}$$

and detect convergence by

$$\textbf{inv}\ \text{best\_fit(B,w)} \land \text{item(I,w,0,0)} \Rightarrow \text{BestFit(B,w)} \qquad \text{(S20)}$$

To ensure convergence we add to *Bag_In(B,w)* a subtransaction of the form

$$\begin{array}{ll} \| : & [\forall\ I', w' : \text{item(I',w',0,0)} :: w' \neq w],\ \text{best\_fit(B,w)}\dagger,\ w > 0 \\ & \longrightarrow \\ & \text{best\_fit(B,w-1)} \end{array}$$

The next step is to substitute the LHS of (S20) for *BestFit(B,w)* in *Bag_In(B,w)*. This poses a problem because we can no longer use the **ensures** (as in 5.1.4) to prove (P7) the original leads-to property on which *Bag_In(B,w)* was based. The reason is that we used the set of *Bag_In* transactions to prove (P7) and now we are altering their meaning. We need to prove (P7) differently by examining the global effect of the local convergence of *best_fit(B,w)*. Such a proof splits (P7) into two **leads-to** properties on which transitivity can be applied:

$$\begin{array}{l} \text{NrWg(w)} = k \land \text{MaxFitWg(w)} \\ \longmapsto \\ \text{NrWg(w)} = k \land \text{MaxFitWg(w)} \land \text{NearFit(w)} \end{array} \qquad \text{(P11)}$$

and

$$\begin{array}{l} \text{NrWg(w)} = k \land \text{MaxFitWg(w)} \land \text{NearFit(w)} \\ \longmapsto \\ \text{NrWg(w)} < k \end{array} \qquad \text{(P12)}$$

where

$$\text{NearFit(w)} \equiv [\exists\ B :: \text{best\_fit(B,w)}] \qquad \text{(D15)}$$

We can see that the new subtransaction helps to prove (P11) through induction, and the first subtransaction of *Bag_In(B,w)* with *BestFit(B,w)* properly replaced helps to prove (P12). *Bag_In(B,w)* becomes

```
Bag_In(B,w) ≡
    I,n,c :
        best_fit(B,w)†, capacity(B,c)†, next_pos(B,n)†, item(I,w,0,0)†
        ⟶
        item(I,w,B,n), capacity(B,c-w), next_pos(B,n+1), best_fit(B,min(w,c-w))
||  :   [∀ I',w' : item(I',w',0,0) :: w' ≠ w], best_fit(B,w)†, w > 0
        ⟶
        best_fit(B,w-1)
||  I : item(I,w,0,0) ⟶ Bag_In(B,w)
```

The assertion of *best_fit(B,min(w,c-w))* is necessary to maintain the invariant (S19) by keeping $w \leq c$ for *capacity(B,c)*.

The same process is applied to *Make_Bag(B,w)* to reduce the complex query *MaxWg(w)*. The result includes the introduction of an approximating tuple, *max_wg(w)*. Convergence is defined as

$$\textbf{inv}\ 1 = [\Sigma\ \text{w} : \text{max\_wg(w)} :: 1] \tag{S21}$$
$$\textbf{inv}\ \text{max\_wg(w)} \wedge \text{MaxWg(w')} \Rightarrow \text{w} \geq \text{w'} \tag{S22}$$
$$\text{max\_wg(w)} \wedge \text{MaxWg(w')} \wedge \text{w} > \text{w'} \longmapsto \text{max\_wg(w-1)} \tag{P13}$$

Convergence is detected by

$$\textbf{inv}\ \text{max\_wg(w)} \wedge \text{item(I,w,0,0)} \Rightarrow \text{MaxWg(w)} \tag{S23}$$

A subtransaction is added to *Make_Bag(B,w)* to guarantee convergence and *MaxWg(w)* is replaced in the first subtransaction to detect convergence, as was done with *best_fit(B,w)*. The tuple *max_wg(H)* must be present initially in the dataspace.

```
Make_Bag(B,w) ≡
    I :
        max_wg(w), [∀ B',c' : capacity(B',c') :: c' < w], next_bag(B)†, item(I,w,0,0)†
        ⟶
        item(I,w,B,1), next_bag(B+1), capacity(B,H-w), next_pos(B,2), best_fit(B,min(w,H-w))
||  :   [∀ I',w' : item(I',w',0,0) :: w' ≠ w], max_wg(w)†, w > 0 ⟶ max_wg(w-1)
||  I : item(I,w,0,0) ⟶ Make_Bag(B,w)
```

At this point the initialization requirements become

$$\begin{aligned} \text{INIT} \Rightarrow\ & [\forall\ \text{w,B} : (1 \leq \text{w} \leq \text{H}) \wedge (1 \leq \text{B} \leq \text{N}) :: \text{Make\_Bag(B,w)}] \\ & \wedge [\forall\ \text{w,B} : (1 \leq \text{w} \leq \text{H}) \wedge (1 \leq \text{B} \leq \text{N}) :: \text{Bag\_In(B,w)}] \\ & \wedge \text{max\_wg(H)} \wedge \text{next\_bag(1)} \end{aligned} \tag{C6}$$

5.2.4 Refinement 4: maintaining only necessary transactions

Our final goal is to restrict the number of transactions present in the transaction space at any one time in order to reduce the time and space complexity. To do so we take advantage of Swarm's ability to dynamically create new transactions. Ideally, we want a transaction to exist only in those states in which it can perform some useful work, i.e., alter the current state. This is not always possible. In some cases, transactions must perform unavoidable waiting. In other cases, a state change may render some transactions useless but the elimination of the transaction cannot take place until it is selected for execution. This latter case will not be observed in this example.

Given a transaction T, we analyze its queries and seek to discover a predicate P that provides a reasonable characterization for the set of states in which T can make a useful contribution. In addition, we want to select P in such a way that (1) any transaction that establishes P can also create T without much added complexity; and (2) the only transaction that invalidates P is T itself. Upon finding such a P, we attempt to alter the program in order to achieve **inv** $P \Leftrightarrow T$.

In the case of *Make_Bag(B,w)* it is clear that no useful work can be performed unless the next empty bag is B and the largest weight among all items is approximated by w, (i.e., $P \equiv max\_wg(w) \wedge next\_bag(B)$). Based on this observation, it is reasonable to attempt to modify the program so as to enforce

$$\textbf{inv}\ (\text{max\_wg(w)} \wedge \text{next\_bag(B)}) \Leftrightarrow \text{Make\_Bag(B,w)} \qquad \text{(S24)}$$

Since the LHS of (S24) is affected only by transactions of type *Make_Bag*, enforcing this invariant involves only a redefinition of *Make_Bag(B,w)*. We approach the task by considering the subtransactions of *Make_Bag(B,w)* and their impact on the LHS of (S24). The first two subtransactions are mutually exclusive and replace *next_bag(B)* and *max_wg(w)* by *next_bag(B+1)* and *max_wg(w-1)*, respectively. To preserve the invariant, we must create corresponding instances of *Make_Bag*, i.e., *Make_Bag(B+1,w)* and *Make_Bag(B,w-1)*, and we must ensure that the third subtransaction is prevented from recreating *Make_Bag(B,w)* whenever either of the other subtransactions succeeds. The final result is

```
Make_Bag(B,w) ≡
    I :
        max_wg(w), [∀ B',c' : capacity(B',c') :: c' < w], next_bag(B)†, item(I,w,0,0)†
        ⟶
        item(I,w,B,1), next_bag(B+1), Make_Bag(B+1,w),
        capacity(B,H-w), next_pos(B,2), best_fit(B,min(w,H-w))
||  :
        [∀ I',w' : item(I',w',0,0) :: w' ≠ w], max_wg(w)†, w > 0
        ⟶
        max_wg(w-1), Make_Bag(B,w-1)
||  I,B',c' :
        item(I,w,0,0), capacity(B',c'), w ≤ c' ⟶ Make_Bag(B,w)
```

Having established (S24) and by noticing that the tuples *next_bag(B)* and *max_wg(w)* are not referenced anywhere else in the program, the transaction above can be further simplified leading to the following transaction.

```
Make_Bag(B,w) ≡
    I :
        [∀ B',c' : capacity(B',c') :: c' < w], item(I,w,0,0)†
        ⟶
        item(I,w,B,1), Make_Bag(B+1,w),
        capacity(B,H-w), best_fit(B,min(w,H-w)), next_pos(B,2)
||  :
        [∀ I',w' : item(I',w',0,0) :: w' ≠ w], w > 0 ⟶ Make_Bag(B,w-1)
||  I,B',c' :
        item(I,w,0,0), capacity(B',c'), w ≤ c' ⟶ Make_Bag(B,w)
```

None of these transformations have any impact on the other properties of the program.

For *Bag_In(B,w)*, one way to accomplish the same task is to require

$$\textbf{inv}\ \text{best\_fit(B,w)} \Leftrightarrow \text{Bag\_In(B,w)} \qquad \text{(S25)}$$

(i.e., $P \equiv best\_fit(B, w)$). This time the changes are not limited to a single definition because *Make_Bag(B,w)* can create tuples of type *best_fit*. To satisfy (S25), *Make_Bag* must create a transaction *Bag_In(B,w,)* whenever it creates a tuple *best_fit(B,w)*. (See final version of the program in Figure 2.)

The transaction type definition for *Bag_In(B,w)* becomes

```
Bag_In(B,w) ≡
    I,n,c :
        best_fit(B,w)†, capacity(B,c)†, next_pos(B,n)†, item(I,w,0,0)†
        ⟶
        item(I,w,B,n), capacity(B,c-w), next_pos(B,n+1),
        best_fit(B,min(w,c-w)), Bag_In(B,min(w,c-w))
||  :
        [∀ I',w' : item(I',w',0,0) :: w' ≠ w], best_fit(B,w)†, w > 0
        ⟶
        best_fit(B,w-1), Bag_In(B,w-1)
```

The subtransaction tasked with recreating the earlier version of *Bag_In(B,w)* disappears all together because it is always the case that one of the other subtransactions succeeds.

Finally, having established the invariant relation between *best_fit(B,w)* and *Bag_In(B,w)*, we can eliminate the former throughout the program. Moreover, if we allow *Bag_In(B,w)* to carry two extra parameters, the invariant,

$$\text{inv capacity(B,c)} \wedge c > 0 \wedge \text{next\_pos(B,n)} \Leftrightarrow \text{Bag\_In(B,w,c,n)} \wedge w > 0 \qquad \text{(S26)}$$

can also be established resulting in the simpler transaction definition below (with some related changes in *Make_Bag(B,w)*).

```
Bag_In(B,w,c,n) ≡
    I :
        item(I,w,0,0)† ⟶ item(I,w,B,n), Bag_In(B,min(w,c-w),c-w,n+1)
||  :
        [∀ I',w' : item(I',w',0,0) :: w' ≠ w], w > 0 ⟶ Bag_In(B,w-1,c,n)
```

The final Swarm program is given in its entirety in Figure 2. The special query NOR can be used in the subtransaction of each transaction definition in place of *[∀ I',w' : item(I',w',0,0) :: w' ≠ w]* to further reduce query complexity.

Finally, we must address termination of the program to show that the derivation process does not violate (P2), i.e., $GPOST \longmapsto TERM$. Therefore, it must be shown that

$$\begin{array}{l}[\forall \text{ I,w,B,n: item(I,w,B,n)} :: B > 0] \\ \longmapsto \\ \neg[\exists \text{ B,w} :: \text{Make\_Bag(B,w)}] \wedge \neg[\exists \text{ B,w,c,n} :: \text{Bag\_In(B,w,c,n)}]\end{array} \qquad \text{(P16)}$$

By the bagging policy (S6), the above progress condition can be restated as

$$\begin{array}{l}\neg[\exists \text{ I,w} :: \text{item(I,w,0,0)}] \\ \longmapsto \\ \neg[\exists \text{ B,w} :: \text{Make\_Bag(B,w)}] \wedge \neg[\exists \text{ B,w,c,n} :: \text{Bag\_In(B,w,c,n)}]\end{array} \qquad \text{(P17)}$$

```
Program Bagger (H, N, weight : natural(H), natural(N), weight[1..H] of natural)

tuple types
  [I,w,B,n : natural(I), natural(w), 0 < w ≤ H, natural(B), natural(n) :: item(I,w,B,n)]

transaction types
  [B,c,n,w : natural(B), natural(c), 0 ≤ c ≤ H, natural(n), natural(w), 0 < w ≤ H ::
     Bag_In(B,w,c,n) ≡
        I :
           item(I,w,0,0)† ⟶ item(I,w,B,n), Bag_In(B,min(w,c-w),c-w,n+1)
     ||   :
           NOR, w > 0 ⟶ Bag_In(B,w-1,c,n)

     Make_Bag(B,w) ≡
        I :
           item(I,w,0,0)†, [∀ B',w',c',n' : Bag_In(B',w',c',n') :: c' < w]
           ⟶
           item(I,w,B,1), Make_Bag(B+1,w), Bag_In(B,min(w,H-w),H-w,2)
     ||   :
           NOR, w > 0 ⟶ Make_Bag(B,w-1)
     ||   I,B',w',c',n' :
           item(I,w,0,0), Bag_In(B',w',c',n'), w ≤ c' ⟶ Make_Bag(B,w)
  ]

initialization
  [I : 0 < I ≤ N :: item(I, weight(I),0,0), Make_Bag(1,H)]
```

Figure 2: Final concurrent version of *Bagger*.

Due to the use of approximating tuples to reduce query complexity and the replacement of the transaction class name for tuples, the proof of (P17) results from applying the transitivity of **leads-to** to the following two progress conditions.

$$\neg[\exists\ \mathrm{I,w} :: \mathrm{item(I,w,0,0)}] \\ \longmapsto \\ [\forall\ \mathrm{B} :: \mathrm{Make\_Bag(B,0)}] \wedge [\forall\ \mathrm{B,c,n} :: \mathrm{Bag\_In(B,0,c,n)}] \qquad \text{(P18)}$$

and

$$[\forall\ \mathrm{B} :: \mathrm{Make\_Bag(B,0)}] \wedge [\forall\ \mathrm{B,c,n} :: \mathrm{Bag\_In(B,0,c,n)}] \\ \longmapsto \\ \neg[\exists\ \mathrm{B,w} :: \mathrm{Make\_Bag(B,w)}] \wedge \neg[\exists\ \mathrm{B,w,c,n} :: \mathrm{Bag\_In(B,w,c,n)}] \qquad \text{(P19)}$$

The proof of (P18) is a consequence of using a variant function defined over the parameter w in each transaction definition and showing the value of w decreases to zero. Informally, we can show that due to the initialization of all items to a weight greater than zero by (D2), items having a constant weight in (S2), the stability of bagged times in (S5), and the **NOR** clause in each transaction definition (Figure 2), w decreases incrementally to zero when $\neg[\exists\ I,w :: item(I,w,0,0)]$. The proof of (P19) follows directly from the transaction definitions, such that in order for the transaction to be reasserted, either there must exist an unbagged item or the weight parameter of the transaction must be greater than zero.

The final version of the program is compact and highly concurrent. The strategy used to develop the program is formal in the sense that every refinement can be shown to be correct–even though, for the sake of brevity, many of the proof details were omitted. The strategy is economical, i.e., most proofs involve only small parts of the program or the specification. To a large extent, this is due to the use of a UNITY-like proof system but also, due to the way in which we structured the overall derivation process. This same careful structuring of the process, we believe, makes it feasible to use our derivation strategy on larger problems.

6 Implementing the Program

In this section, we explain how a Swarm program can be translated to a traditional rule-based programming language, OPS5, and discuss the results of executing the translated program on a parallel rule-based system. The performance of the parallel execution will be compared against that of the original sequential program from [19].

The program derived in Section 4, henceforth called the concurrent *Bagger*, has the property that the queries of subtransaction making up each individual transaction are disjoint. This enables us to map each subtransaction in a transaction class definition to a distinct OPS5 rule and to use a WME to indicate the existence of each transaction instance in the dataspace. The resulting OPS5 program can be easily executed in PARS without violating the Swarm semantics. The synchronous version of PARS (for Parallel Asynchronous Rule-based System) [17] is a running system that ensures serializability attempts to maximize the number of rule instances executed in each cycle. A parallel program is serializable if the parallel execution of a program is equivalent to some serial execution. There are two types of rule interference that can violate serializability, *disabling* and *clashing* [16]. Disabling occurs when two rules are instantiated and one rule's actions delete (add) a WME that the other rule positively (negatively) matches against.[6] If a cycle of disabling exists among instantiated rules, in order to execute sequentially the cycle must be broken by disallowing one instantiation to execute. Clashing occurs if either one rule instance adds (deletes) a WME that another deletes (adds) and one instance disables the other or the interleaving of the add and delete actions of the two rule instances can give results that could not be achieved serially. In PARS, rules that clash are not allowed to execute in parallel.

[6]A WME is *positively* matched against if the match is successful because the WME exists. A WME is *negatively* matched against if the match is successful because no such WME exists.

We used the synchronous PARS implementation that allows every instantiation possible to execute during each cycle provided the serializability constraints are not violated.

In Section 6.1 we explain and illustrate the translation process. In Section 6.2 we review the sequential version of *Bagger* found in [19], henceforth called the original *Bagger*. We summarize the results of the parallel execution of the concurrent *Bagger* in Section 6.3.

6.1 Returning to Traditional Rule-Based Languages

In this section we explain how the class of Swarm programs previously described is translated into OPS5. In the derived program three main constructs are used that are not directly available in OPS5: (1) transactions, (2) the ||-operator with subtransactions and (3) a dynamic knowledge base. We will concentrate on how these can be represented in OPS5. Other concepts such as the use of $\forall$, special queries, global constants and data tuples have direct correspondents. As an example of the translation, Figure 3 gives the OPS5 rules corresponding to the transaction class *Bag_In*.

- **Transactions:** It is necessary to define a WME for each transaction instance present in the dataspace. In the concurrent *Bagger*, two types of working memory elements are used for this purpose.

  ```
  (Bag-In ^bag <B> ^wgt <w> ^cap <c> ^pos <n>)
  (Make-Bag ^bag <B> ^wgt <w>)
  ```

  The translation must ensure that whenever a transaction is present in the transaction space, the corresponding WME is in working memory.

- **The ||-Operator with subtransactions:** Because in Swarm a single transaction is chosen non-deterministically for execution, any enabled transaction that is chosen can successfully execute without violating correctness criteria. In the class of Swarm programs to which the concurrent *Bagger* belongs, a transaction is enabled by a single transaction, allowing no interference among subtransactions of the same transaction. This restricted enabling allows multiple subtransactions of a transaction to be separated into distinct rules, each controlled by the WME representing the parent transaction. Therefore, the conflict set in the rule-based program can contain at most one rule per transaction (i.e., one subtransaction per transaction can be in the conflict set during any cycle.) Then any sequential execution ordering is correct because that same ordering can be executed in Swarm. In OPS5, we do not have to be concerned about interference between subtransactions of different transactions because only a single rule is executed per cycle. Because PARS executes all possible instantiations in each cycle, this interference is a concern, but it is handled by the serializability constraints of the system.

- **Dynamic Knowledge Base:** The subtransactions, now individual rules, are responsible for explicit addition and deletion of the WME representing the parent transaction, as was performed in the Swarm program. This simulates the presence and absence of transactions in the transaction space. In Swarm, transactions are deleted implicitly as a by-product of their execution. To maintain correctness, the WME in the LHS of the rule that represents the transaction class must be treated as the transaction was treated by the corresponding subtransaction. Thus, it must be explicitly deleted after each firing of the rule, i.e., the LHS of the rule is satisfied and the actions of the RHS are performed.

We rely on the absence of matching rules for termination of the rule-based program in OPS5. Since the rule translation does not violate any correctness conditions, we know that eventually all items will be bagged, and that all WMEs of type `Make_Bag` and `Bag_In` will have a `wgt` attribute value of zero. Since we no longer need to show that all transactions are deleted from the transaction space, we must show that when these WMEs have a zero value for the `wgt` attribute, there is no possible match of the LHS of all rules. Looking at the definition of each rule and using the fact that items always have a weight greater than zero, we see that no rules can match, thereby causing termination of the OPS5 program.

Swarm Transaction

$Bag\_In(B,w,c,n) \equiv$

    I :
        item(I,w,0,0)† $\longrightarrow$ item(I,w,B,n), Bag_In(B,min(w,c-w),c-w,n+1)
    || :
        NOR, $w > 0 \longrightarrow$ Bag_In(B,w-1,c,n)

OPS5 Rules

Rule `Bs1` *corresponds to the first subtransaction of Bag_In(B,w,c,n).*

```
(p Bs1
  {(Bag-In ^bag <B> ^wgt <w> ^cap <c> ^pos <n>) <bag-item>}
  {(item ^wgt <w> ^bag 0 ^pos 0) <item-change>}
  -->
  (modify <bag-item> ^wgt (compute <w> :min <c> - <w>)
     ^cap (compute <c> - <w>)
     ^pos (compute <n> + 1))
  (modify <item-change> ^bag <B> ^pos <n>)
)
```

Rule `Bs2` *the second subtransaction for Bag_In(B,w,c,n)..*

```
(p Bs2
  {(Bag-In ^bag <B> ^wgt <w> ^wgt {<w> > 0}) <bag-item>}
  -(item ^wgt <w> ^bag 0 ^pos 0)
  -->
  (modify <bag-item> ^wgt (compute <w> - 1))
)
```

Figure 3: The Translation of *Bag_In(B,w,c,n)* to OPS5

6.2 Original Bagger Implementation

Earlier we made the point that one of the reasons significant performance improvements have not been achieved by executing multiple rules simultaneously is because researchers in parallel rule-based systems use programs written for sequential execution as a testbed. It is our hypothesis that when these programs are derived without sequential controls or biases that the amount of available concurrency increases, translating to more parallelism during execution.

The original *Bagger* [19] takes advantage of the available conflict resolution strategy, especially relying on specificity. Essentially, it groups unbagged items according to weight, and packs all of those with the heaviest weight first (one at a time), creating bags as needed, before packing any items of the next heaviest weight. This process relies on sequential tasking and context switching. The program as described is inherently sequential. Because of the specificity conflict resolution rule used, it appears as though multiple rules can execute. But when conflict resolution is made explicit in the rules, as it must be for implementation on a parallel rule-based system [11], only a single rule can execute in any given state. Thus, *the results given for parallel execution would be the same as those for sequential execution.* The program has been proven to correctly obey the same general specifications as the derived program but with this explicit tasking information given [8]. It is to this program that we compare our results of the parallel execution of the concurrent *Bagger* to validate our original hypothesis.

6.3 Results for Implementation

Once in OPS5, the actual implementation into PARS was straightforward. It is important to remember that: (1) the original *Bagger* (Section 6.2) could exhibit no parallelism if executed in PARS and (2) the amount of available concurrency in the concurrent *Bagger* (Section 5) depends on the number of available bags, which are created only as needed. In this section, we show the results of executing the original (OPS5) *Bagger* and the concurrent (Swarm) *Bagger* on two data sets called *run-1* and *run-2*. The data in *run-1* permits only limited parallelism because (a) the average number of items per bag is low, and (b) more than one item of the current largest weight can fit in a bag restricting new bags from being created early in the computation. The data from *run-2* does not suffer from these restriction and takes greater advantage of the available concurrency. Details of the sample data from each run is presented in Table 1.

example	total number of items	maximum bag weight	distribution of item weights	total weight of items	minimum number of bags needed
run-1	10	7	1 @ wgt=5 2 @ wgt=4 3 @ wgt=3 2 @ wgt=2 2 @ wgt=1	28	4
run-2	34	9	7 @ wgt=5 27 @ wgt=1	62	7

Table 1: Example Details

The results of executing both programs, the original and the concurrent, are given in Table 2. A cycle in both executions is the match-select-act phases typical in a rule-based program. In PARS, all non-interfering instantiations that enter the conflict set are executed, thus the term *parallel cycle*. In the case of the original *Bagger*, only one possible instantiation can be executed per cycle given any program state. The average number of parallel rules per cycle is the total number of rule firings divided by the number of execution cycles. The maximum number is the greatest number of rules that were executed in parallel in any one cycle of the execution.

example	system	total number of cycles	total number of parallel cycles	average number of parallel rules executed per cycle	maximum number of parallel rules executed per cycle
run-1	OPS5	20			
	PARS		16	1.3	2
run-2	OPS5	46			
	PARS		20	3.3	8

Table 2: Implementation Details

It is clear from looking at Table 2 that the concurrent Bagger executes in fewer cycles. For *run-1* the total number of cycles decreases only 20%. This is because after the first item is placed in the first bag, the remaining largest unbagged item also fits in that bag, so a new bag cannot be created on the next cycle. This occurs three times during the packing of only four bags. In contrast, *run-2* gets a 57% decrease in the number of cycles. Improvements can be made to the amount of parallelism exhibited in implementation by knowing a priori the largest item weight in order to avoid wasted sequential cycles early in the computation in which *Make-Bag* (from the derived program) must decrement to that weight. If there was no counting necessary to reach the initial largest item, the number of cycles in *run-1* would decrease to 14, a decrease of 30% and the number of cycles in *run-2* would further decrease to 16, a 65% decrease.

As stated earlier the maximum amount of concurrency available in the concurrent *Bagger* in a particular state is the number of non-full bags plus one, for the possible creation of a new bag. In this program a transaction either packs an item, decrements a counter, or waits (only for class *Make-Bag*). An execution that maximizes concurrency would have every bag packing an item and one new bag created on every cycle. For example, after 4 cycles of exploiting the maximal concurrency, bag#1 has 4 items, bag#2 has 3 items, bag#3 has 2 items and bag#4 has 1 item. The average number of rules that executed simultaneously is 2.5. In general, an upper bound on the average over all cycles of the maximum available concurrency at each cycle is $\frac{n+1}{2}$, where n is the minimal number of bags needed for packing all the items without regard to their individual weights. In *run-1* this bound is 2.5 as given in the example above and for *run-2* this bound is 5. The overhead consisting of the other possible operations of the transactions limits the actual amount of parallelism that can be realized. Returning to Table 2, the average number of rules executed in parallel in *run-1* comes within 52% of the upper bound despite its poor data, while *run-2* comes within 66%. For *run-2*, if *Make_Bag* had a priori knowledge of the initial largest item weight, the average number of rules that executed simultaneously would increase to 4, coming within 80% of the bound.

7 Discussion

When compared with current rule-based programming languages, Swarm makes available a richer repertoire of programming constructs . However, the derivation process using the Swarm computational model can be used for deriving any rule-based program, provided that the features of the target language are considered in the derivation process. For instance, if we are interested in OPS5 then we may want to make sure that in the final program subtransactions appearing in the same transaction have disjoint queries, as is the case with the concurrent *Bagger*. If the program does not fit into this schema, its translation to OPS5 may still be possible, but no longer a mechanical transformation.

In general, it is necessary to define an appropriate schema (i.e., restricted Swarm structure) which ensures that the resulting Swarm program has a direct translation into the target language, and to bias the derivation process toward this schema. As a matter of fact, a similar situation is commonly encountered in the derivation of concurrent programs where the derivation process starts with a very general problem specification that is gradually refined toward a specific target architecture (e.g., shared memory or message passing). In rule-based program derivation we simply have a different target, not a desired architecture, but an existing rule-based language.

8 Conclusions

The theme of this paper is the formal derivation of concurrent rule-based programs from their specifications. Our program derivation strategy applies, adapts, and extends techniques already well established in concurrent programming to the domain of rule-based programming. Our aim is to apply formal techniques in a manner which frees the programmer from considering unnecessary details. The emphasis is on clean formal thinking in a practical setting. Our program derivation strategy is divided into two major tasks. The first task relies on specification refinement. Techniques similar to those employed in the derivation of UNITY programs are used to produce a correct rule-based program having a static knowledge base, i.e., a fixed set of rules. The approach has direct applicability to the generation of programs targeted to currently popular rule-based programming languages, such as OPS5. The second task involves program refinement and is specific to the development of concurrent rule-based programs. It relies heavily on the availability of a computational model, such as Swarm, that has the ability to dynamically restructure the knowledge base. Here, the concern with achieving high degrees of concurrency and with reducing query complexity guides the program transformation. To complete the example, we explained how a Swarm program could be translated to OPS5 specifically, given some restrictions, while maintaining the correctness criteria. The execution of derived program on a parallel rule-based system showed improvement over the execution of the original program developed initially for sequential execution.

Acknowledgement: Special thanks go to Dr. James Schmolze of Tufts University for executing the concurrent *Bagger* on PARS.

9 References

[1] R. J. R. Back and K. Sere, "Stepwise Refinement of Parallel Algorithms," *Science of Computer Programming*, 13, pp. 133-180 (1990).

[2] J. P. Banâtre and D. Le Métayer, "The GAMMA model and its discipline of programming," *Science of Computer Programming*, 15, pp. 55-77 (1990).

[3] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, Vol. 32, No 4, pp. 444-458 (1989).

[4] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison- Wesley, New York (1988).

[5] H. C. Cunningham and G.-C. Roman, "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 365-376 (1990).

[6] E. D. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).

[7] C. L. Forgy, "OPS5 User's Manual," Technical Report CMU-CS-81-13, Carnegie-Mellon University (1981).

[8] R. F. Gamble, G.-C. Roman, and W. E. Ball, "Formal Verification of Rule-Based Programs," *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 329-334 (1991).

[9] D. Gries, *The Science of Programming*, Springer-Verlag, New York, NY (1987).

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

[11] T. Ishida and S. J. Stolfo, "Towards the Parallel Execution of Rules in Production System Programs," *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 568-575, (1985).

[12] D. P. Miranker, C. M. Kuo, and J. C. Browne, "Parallel Compilation of Rule-based Programs," *Proceedings of 1990 International Conference on Parallel Processing*, pp. 247-251, (1990).

[13] A. Pasik and S. J. Stolfo, "Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules," Technical Report, Columbia University (1987).

[14] M. Rem, "Associons: A Program Notation with Tuples Instead of Variables," *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 3, pp 251-262 (1981).

[15] G.-C. Roman and H. C. Cunningham, "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering*, Vol. 16, No. 12,. pp 1361-1373 (1990).

[16] J. G. Schmolze, "Guaranteeing Serializable Results in Synchronous Parallel Production Systems,"*Journal of Parallel and Distributed Computing*, Vol. 13, No. 4, pp 348-365 (1991).

[17] J. G. Schmolze and S. Goel, "A Parallel Asynchronous Distributed Production System," *Pro-. ceeding of the 8th National Conference on Artificial Intelligence*, pp. 65-71 (1990)

[18] T. Sellis, C.-C. Lin and L. Raschid, "Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms," Technical report UMIACS-TR-87-68, University of Maryland, College Park (1987).

[19] P. H. Winston, *Artificial Intelligence, 2nd Edition*, Addison-Wesley Publishing Company, Reading, MA (1984).