Report Number: WUCS-92-04

1992

Seeking Concurrency in Rule-based Programming

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

This paper describes a formal approach for developing concurrent rule-based programs. Specification of 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. To the best of our knowledge, this is the first time formal derivation is employed in the context of rule-based programming.

Washington

WASHINGTON • UNIVERSITY • IN • ST • LOUIS

School of Engineering & Applied Science

Seeking Concurrency in Rule-based Programming

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

WUCS-92-4

January 1992

This paper appeared in *Proceedings of the 14th International Conference on Software Engineering*, May 1992, pp. 225-234.

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

Seeking Concurrency in Rule-based Programming

Gruia-Catalin Roman[†] *Rose F. Gamble*[‡] *William E. Ball*[‡]

Department of Computer Science
Washington University
St. Louis, Missouri, USA

Abstract

This paper describes a formal approach for developing concurrent rule-based programs. 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. To the best of our knowledge, this is the first time formal derivation is employed in the context of rule-based programming.

1 Introduction

Program derivation refers to a systematic formal process of constructing correct programs from their specifications, typically through some form of stepwise refinement. Chandy and Misra's work on UNITY [4] advocates an approach in which a formal specification of the problem is gradually refined to the point where the specification is restrictive enough to suggest a 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. The approach is targeted toward rule-based programs that terminate, but may be easily extended to non-terminating programs.

Rule-based programming is a common programming paradigm in the artificial intelligence community. To date, most efforts toward concurrency in this area start with existing sequential 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 [7,8,9,14]. In contrast, we are concerned with generating programs that exhibit a high potential for parallel execution, i.e., they are less likely to impose sequential dependencies that could undermine parallel implementations. The program notation and proof logic used in this paper are those of Swarm [12], 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.

The remainder of the paper consists of four main parts followed by conclusions. Section 2 introduces Swarm and the 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 [15] relies on conflict resolution for tasking and rule-ordering, and no speed-up would be gained if it were executed on available parallel production system models, such as those proposed by Ishida and Stolfo [8], and Schmolze [14]. Section 5 presents a systematic formal derivation of a highly concurrent version of this program without reliance on traditional conflict resolution.

2. Notation

Swarm [12] belongs to a class of languages and models that use tuple-based communication. Other languages and models in this class are Linda [3], Associons [11], and GAMMA [2]. In this section we review briefly 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 purposes of this paper, the dataspace is partitioned into a *tuple space*, which corresponds to working memory, and a *transaction space*, which corresponds to the knowledge base.

Working memory. The tuple space consists of a set of data tuples (elsewhere called working memory elements). A data tuple assumes the form:

```
class_name(sequence_of_attribute_values).
```

For example, *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

† Supported in part by the National Science Foundation under the Grant CCR-9015677. The government has certain rights in this material.

‡ Supported by the Center for Intelligent Computer Systems at Washington University, whose primary sponsors are McDonnell Douglas Corporation and Southwestern Bell Corporation.

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 (†). An example will be given next.

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 the LHS of the simple transaction and the action list is the RHS. For instance,

$$
\begin{aligned}
&B,I_1,w_1,n_1,I_2,w_2,n_2 :\\
&\quad \text{item}(I_1,w_1,B,n_1) \wedge \text{item}(I_2,w_2,B,n_2) \wedge\\
&\quad (w_1 > w_2) \wedge (n_1 > n_2)\\
&\rightarrow\\
&\quad \text{item}(I_1,w_1,B,n_1)\dagger,\ \text{item}(I_2,w_2,B,n_2)\dagger,\\
&\quad \text{item}(I_1,w_1,B,n_2),\ \text{item}(I_2,w_2,B,n_1)
\end{aligned}
$$

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. The exchange is done by deleting the old instances of the two item descriptions and by inserting new ones. This example illustrates that the query of a transaction is a predicate that may involve testing for the presence or absence of data tuples. A successful query binds the variables listed before the query (implicitly these are existential quantified) to values that can be used to compute the dataspace deletions and insertions. All deletions and insertions must contain only bound 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 is unsuccessful (i.e., it evaluates to false), no explicit deletions or insertions are performed. For convenience, commas may be used inside the query as shorthand for the logical *and* ($\wedge$), the order in which deletions and insertions are listed is immaterial, and when the tuples being deleted are present in the query part, their deletion can be marked by daggers inside the query.

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

$$
\begin{aligned}
&\text{Swap}(B) \equiv\\
&\quad I_1,w_1,n_1,I_2,w_2,n_2 :\\
&\qquad \text{item}(I_1,w_1,B,n_1)\dagger,\ \text{item}(I_2,w_2,B,n_2)\dagger,\\
&\qquad (w_1 > w_2),\ (n_1 > n_2)\\
&\quad \rightarrow\\
&\qquad \text{item}(I_1,w_1,B,n_2),\ \text{item}(I_2,w_2,B,n_1)
\end{aligned}
$$

This becomes a transaction type or transaction class definition, whose parameters (if any) are attribute values. The transaction space contains transaction instances (henceforth simply called transactions) that can be executed by the program at a particular point in the computation. 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 for technical reasons the action list may not include deletions of transactions.

Transaction selection and execution. The transaction space of a Swarm program consists of a set of transactions. Fairness requires that each transaction in the transaction space is eventually selected and executed (atomically). 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 in its action list are performed. The transaction is deleted implicitly unless its reinsertion into the transaction space is one of its actions.

Composition of simple transactions. In Swarm, the ||-operator, borrowed from UNITY, may be used to combine several simple transactions into a single complex transaction. For instance, a transaction that swaps out-of-order items in a bag B can be composed with a transaction that simultaneously counts the number of swaps and reinserts the transaction:

$$
\begin{aligned}
&\text{Swap\_and\_Count}(B) \equiv\\
&\quad I_1,w_1,n_1,I_2,w_2,n_2 :\\
&\qquad \text{item}(I_1,w_1,B,n_1)\dagger,\ \text{item}(I_2,w_2,B,n_2)\dagger,\\
&\qquad (w_1 > w_2),\ (n_1 > n_2)\\
&\quad \rightarrow\\
&\qquad \text{item}(I_1,w_1,B,n_2),\ \text{item}(I_2,w_2,B,n_1)\\
&\|\ k:\\
&\qquad \textbf{OR},\ \text{swapcount}(k)\dagger\\
&\quad \rightarrow\\
&\qquad \text{swapcount}(k+1),\ \text{Swap\_and\_Count}(B)
\end{aligned}
$$

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. All deletions associated with successful subtransactions are performed simultaneously, followed by the combined insertions of the same subtransactions.

The special predicate **OR** succeeds, 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 respective meanings *all*, *not all*, *none*, and *no matter how many* of the regular queries appearing in the same transaction succeed.

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*:

[I : 1 ≤ I ≤ M :: item(I,weight(I),bag(I),position(I)),
Swap_and_Count(3), swapcount(0)]

Here, *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]. The three-part notation [op quantified_variables : range :: expression] used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for *op*, e.g., *true* when *op* is $\forall$, $-\infty$ when *op* is *max*, zero when *op* is Σ. 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 and t is a transaction. When properties and inference rules are written without explicit quantification, they are universally quantified over all the values of the free variables occurring in them. The proof rules for the subset of Swarm used in this paper are summarized in Figure 1. The notation $[t]$[1] denotes that the "transaction t is in the transaction space," **TRS** denotes the set of all possible transactions, and *INIT* denotes the initial state of the program. 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 [15] that expresses the desired way in which grocery items should be packed into bags. The universe of grocery items forms a class whose members assume the form *item(I,w,B,n)*, where *item* is the class name, *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. Furthermore, both bags and items are restricted in weight to a maximum value *H*. Given this representation, we develop a formal specification from which the *Bagger* program will be derived.

(a) Items are distinguishable by using unique identifiers and they do not change weight along the way:

$$\underline{\text{inv}}\ [\Sigma\ w,B,n : \text{item}(I,w,B,n) :: 1] \le 1 \qquad \text{(S1)}$$

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

(b) Bags are not permitted to exceed a maximum weight capacity *H* and must have contiguous identifiers:

$$\underline{\text{inv}}\ \text{WgBag}(B) \le H \qquad \text{(S3)}$$

$$\underline{\text{inv}}\ (\text{WgBag}(B_2) > 0) \wedge (B_2 > B_1 > 0) \Rightarrow \text{WgBag}(B_1) > 0 \qquad \text{(S4)}$$

where

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

(c) Once an item is placed in the bag, it cannot be removed nor change positions within the bag.

$$\underline{\text{stable}}\ \text{item}(I,w,B,n) \wedge B > 0 \wedge B = B_0 \wedge n > 0 \wedge n = n_0 \qquad \text{(S5)}$$

(d) Bagged items must have non-zero positions and no two items occupy the same position in the same bag. Items in the same bag are ordered according to their weights, with heavier items packed before lighter ones. 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, guaranteeing that bags are created as needed.

$$\begin{aligned}\underline{\text{inv}}\ [&\forall\ I_1,I_2,w_1,w_2,B_1,B_2,n_1,n_2 \qquad \text{(S6)}\\ &: \text{item}(I_1,w_1,B_1,n_1) \wedge \text{item}(I_2,w_2,B_2,n_2) \wedge (B_1 > 0) \wedge (B_2 > 0)\\ &:: (n_1 > 0) \wedge (n_2 > 0)\\ &\wedge ((I_1 = I_2) \Leftrightarrow (B_1 = B_2) \wedge (n_1 = n_2))\\ &\wedge ((I_1 \neq I_2) \wedge (w_1 > w_2) \wedge (B_1 = B_2) \Rightarrow (n_1 < n_2))\\ &\wedge ((I_1 \neq I_2) \wedge (B_1 < B_2) \wedge (n_2 = 1) \Rightarrow (\text{WgBag}(B_1) + w_2) > H)]\end{aligned}$$

[1] Elsewhere in the paper we simply use t in place of [t] in predicates dealing with the existence of transaction t.

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

The "Hoare triple" means that, 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. $$\frac{(\forall\ t : t \in \mathbf{TRS} :: \{p \wedge \neg q\}\ t\ \{p \vee q\})}{p\ \mathbf{unless}\ q}$$

If p is *true* at some point in the computation and q is not, then executing any single transaction either maintains p or establishes q.

3. $\underline{\text{stable}}\ p \equiv p\ \mathbf{unless}\ \mathit{false}$

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

4. $\underline{\text{inv}}\ p \equiv (\mathit{INIT} \Rightarrow p) \wedge (\underline{\text{stable}}\ p)$

The property p is *true* at all points in the computation, i.e., invariant.

5. $\underline{\text{const}}\ p \equiv (\underline{\text{stable}}\ p) \wedge (\underline{\text{stable}}\ \neg p)$

The property p is either *true* or *false* throughout the computation, i.e., constant.

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

If $p \wedge \neg q$ is *true*, there exists a transaction t in the transaction space that will establish q when it is executed. The fairness assumption guarantees that t will eventually be selected. Its execution establishes q.

7. $p \mapsto q$

This, read p **leads-to** q, means that once p becomes *true*, q will eventually become *true*, but p is not guaranteed to remain *true* until q becomes *true*. The assertion $p \mapsto q$ is *true* if and only if it can be derived by a finite number of applications of the following inference rules:

$$\frac{p\ \mathbf{ensures}\ q}{p \mapsto q}$$

$$\frac{p \mapsto q \wedge q \mapsto s}{p \mapsto s}$$

$$\frac{(\forall\ m : m \in W :: p(m) \mapsto q)}{(\exists\ m : m \in W :: p(m)) \mapsto q}$$ for any set W

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

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

Figure 1: A subset of the Swarm proof logic.

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:

$$\text{GINIT} \mapsto \text{GPOST} \qquad \text{(P1)}$$

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

$$\text{GPOST} \equiv [\forall\ \text{I,w,B,n} : \text{item(I,w,B,n)} :: \text{B} > 0] \qquad \text{(D3)}$$

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 must eventually terminate. Therefore

$$\text{INIT} \Rightarrow \text{GINIT} \qquad \text{(C1)}$$

$$\underline{\text{stable}}\ \text{GPOST} \qquad \text{(S7)}$$

$$\text{GPOST} \mapsto \text{TERM} \qquad \text{(P2)}$$

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 outline the development of a concurrent version of *Bagger*. The full formal derivation is available in [13]. We start with the formal specification given in Section 4 and apply a series of refinements. The basic strategy behind the specification refinement steps of Section 5.1 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.

5.1 Specification Refinement

Having given a formal specification for the *Bagger* problem, 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 view refinement as a reversal of the verification process. In other words, given a property that needs to be refined we consider how one might prove such a property. For instance, proving a progress property such as

$$\text{GINIT} \mapsto \text{GPOST} \qquad \text{(P1)}$$

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

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

is the most obvious way to measure progress. From its definition and from properties (S1), (S2) and (S5) stating respectively that items are unique, that no new items (different from the initial set) can be inserted and that bagged items are stable, we can establish that *NrOut* is non-increasing and well-founded. The progress property (P1) is provable if we require *NrOut* to eventually decrease.

5.1.2 Refinement 2: Introducing local measures

The variant function *NrOut* is a good choice for a sequential program, since 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 the *Bagger* problem 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 weight *w*:

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

All proof obligations relating to *NrOut* are easily reformulated in terms of *NrWg*.

5.1.3 Refinement 3: Shaping local progress

Our next task is 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 weight of the largest unbagged item that fits in a particular bag *B* or *w* is the largest weight among all unbagged items. For a given *w*, these cases reduce to two disjoint situations: either there are bags that need items of weight *w* and as a result some of them get packed, or there are no bags that can hold items of weight *w* and some new bag must be created for this purpose.

At this point the progress property (P1) has been reshaped several times yielding two progress properties. First,

$$\text{NrWg(w)} = \text{k} \wedge \text{MaxFitWg(w)} \;\mapsto\; \text{NrWg(w)} < \text{k} \qquad \text{(P3)}$$

where *MaxFitWg(w)* is true if *w* is the maximum weight of an unbagged item that can fit in an existing bag, as determined by *Fit(B,w)*.

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

$$\text{Fit(B,w)} \equiv \qquad \text{(D6)}$$
$$(\text{w} \le \text{H} - \text{WgBag(B)})$$
$$\wedge\, [\exists\, \text{I',w',n'} : \text{item(I',w',B,n')} :: \text{B} > 0]$$

The second progress property is

$$\text{NrWg(w)} = \text{k} \wedge \text{NoFit(w)} \wedge \text{MaxWg(w)} \wedge \text{NextBag(B)}$$
$$\mapsto$$
$$\text{NrWg(w)} < \text{k} \wedge \text{NextBag(B} + 1) \qquad \text{(P4)}$$

where *NoFit(w)* is true when weight *w* cannot fit in any existing bag. If this weight is maximum among all unbagged items then *MaxWg(w)* is true. *NextBag(B)* holds when *B* is the next bag in the sequence to be created.

$$\text{MaxWg(w)} \equiv (\text{w} = [\max\, \text{I, w'} : \text{item(I,w',0,0)} :: \text{w'}]) \qquad \text{(D8)}$$

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

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

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 can strengthen the progress conditions (P3) and (P4) by replacing the **leads-to** relations with **ensures** relations and we can try to discover transactions that preserve all the safety conditions accumulated so far and help prove (P3) and (P4). For (P3), one possible design choice is to have a transaction *Bag(w)* that selects 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.

```
Bag(w) ≡                                                    (T1)
  I,B,n :
      BestFit(B,w), NextPos(B,n), item(I,w,0,0)†
    →
      item(I,w,B,n)
[] :  TRUE → Bag(w)
```

where *BestFit(B,w)*, which implies *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.

```
BestFit(B,w) ≡                                              (D11)
  (w = [max I',w' : item(I',w',0,0) ∧ Fit(B,w') :: w'])

NextPos(B,n) ≡                                              (D12)
  (n = [max I',w',n' : item(I',w',B,n') :: n'] + 1)
```

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.,

$$INIT \Rightarrow [\forall\ w : 1 \le w \le H :: Bag(w)] \qquad (C2)$$

For (P4) we introduce the transaction class

```
Make_Bag(B,w) ≡                                             (T2)
  I :
      MaxWg(w), NoFit(w), NextBag(B), item(I,w,0,0)†
    →
      item(I,w,B,1)
[] :  TRUE → Make_Bag(B,w)
```

and the requirement

$$INIT \Rightarrow \qquad (C3)$$
$$[\forall\ w,B : (1 \le w \le H) \wedge (1 \le B \le N) :: Make\_Bag(B,w)]$$

The resulting program consists of H transactions of type *Bag* and $H*N$ transactions of type *Make_Bag* with transaction definitions (T1) and (T2). 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 UNITY's conditional assignment statements are deterministic. This version is correct with respect to the specifications from Section 4 (for brevity, proofs are omitted), except that we ignored the termination requirement—we will return to it in a later section.

5.2 Program Refinement

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 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 by the query with a constant, which appears as a new parameter of the corresponding transaction class. The replaced variable must range over a finite set and the transaction space must contain 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 [10] 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; finallu, fairness is not affected because the number of new transactions is finite.

In order to increase the concurrency exhibited by the *Bagger* program we must be able to pack more than one item of weight w at a time. Maximal concurrency can be accomplished by splitting *Bag(w)* across bags:

```
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)
```

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)*. Now multiple bags can pack items having the same weight w. Splitting *Make_Bag* cannot improve concurrency because of

the sequential creation of bags. Therefore, we leave it unchanged.

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 (i.e., can no longer be executed), 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. Up to now all transactions were created initially and existed forever. Since each transaction introduced so far is active only when items of some weight w are present, we can replace **TRUE** by *item(I,w,0,0)* and note that $\neg$*item(I,w,0,0)* is stable for a given w. The transaction definitions become

```
Bag_In(B,w) ≡                                              (T1.1)
  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) ≡                                            (T2.1)
  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)
```

To prove the termination condition (P2), we can simply observe that

$$GPOST \Rightarrow [\forall I,w :: \neg item(I,w,0,0)]$$

5.2.3 Refinement 3: Reducing query complexity

It is well-known within the expert system community that the pattern matching phase of query evaluation consumes a large portion of the total time needed to execute one cycle in a rule-based program [7]. Our program currently uses 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 split the *Bag_In* transaction across bags, allowing individual bags to be in control of the items packed in them. 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. For instance, 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. On the other hand, the values returned by the queries *BestFit(B,w)* and *MaxWg(w)* may change each time the state of the items changes. We therefore use tuples to *approximate* the respective values of these queries. Maintaining the actual values would be unnecessarily difficult, requiring each transaction to re-calculate the values each time the state of the items changes.

Replacing queries by single tuples. To clarify the reduction process, we will first expand *BestFit(B,w)*, make an initial set of simplifications, and later return to the reduction. 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)
```

We can redefine *Bag_In(B,w)* using a tuple *capacity(B,c)*, where c represents the spare capacity of the bag, and a tuple *next_pos(B,n)* that maintains the next position n available in bag B. The tuple *capacity(B,c)* must be inserted 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)*. When bag B is created by filling its first position, *Make_Bag(B,w)* also 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. To reduce *NextBag(B)*, the tuple *next_bag(B)* can be introduced to keep track of the next bag to be created. Because initially the next bag to be created is the first bag, we require that $INIT \Rightarrow$ *next_bag(1)*. Whenever *Make_Bag(B,w)* places the first item of weight w in a bag B, it must also insert *next_bag(B+1)*. The resulting definition of *Make_Bag(B,w)*,

including the changes from introducing tuples required by *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 tuples. We desire to replace the computation of the "max" function in *Bag_In(B,w)* by introducing a tuple called *best_fit(B,w)* that tracks changes in the set of items. (Note that this function computes the maximum weight w among unbagged items that can still fit in bag B.) This is not easily done because bagging elsewhere may use up all the items with weight w. Updating *best_fit(B,w)* in *Bag_In* is too complicated. Our solution is to make w in *best_fit(B,w)* approximate and gradually converge to the maximum w returned by the function. The definition of convergence restricts the tuple space to at most one *best_fit(B,w)* tuple per bag and w to be no greater than the capacity of the bag and at least the actual maximum weight value. The weight values of the tuple and the query it approximates must eventually converge to the same value. Convergence is detected when an unbagged item exists with the weight value of the tuple *best_fit(B,w)*. To ensure convergence we add to *Bag_In(B,w)* a subtransaction of the form

```
||:
    [∀ I :: ¬item(I,w,0,0)], best_fit(B,w)†, w > 0
  →
    best_fit(B,w-1)
```

Making the appropriate substitutions in the *Bag_In* transaction, however, poses a problem because we can no longer use an **ensures** property to prove (P3), 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 (P3) and now we are altering their meaning. We need to prove (P3) differently by examining the global effect of the local convergence of *best_fit(B,w)*. Such a proof splits (P3) into two **leads-to**properties to which transitivity can be applied:

$$\mathrm{NrWg}(w) = k \wedge \mathrm{MaxFitWg}(w) \qquad \text{(P5)}$$
$$\mapsto$$
$$\mathrm{NrWg}(w) = k \wedge \mathrm{MaxFitWg}(w) \wedge \mathrm{NearFit}(w)$$

and

$$\mathrm{NrWg}(w) = k \wedge \mathrm{MaxFitWg}(w) \wedge \mathrm{NearFit}(w) \qquad \text{(P6)}$$
$$\mapsto$$
$$\mathrm{NrWg}(w) < k$$

where

$$\mathrm{NearFit}(w) \equiv [\exists\, B :: \mathrm{best\_fit}(B,w)] \qquad \text{(D13)}$$

The new subtransaction is used to prove (P5) through induction, and the first subtransaction of *Bag_In(B,w)* (below) is used to prove (P6).

```
Bag_In(B,w) ≡                                          (T1.2)
  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 :: ¬item(I,w,0,0)], best_fit(B,w)†, w > 0
    →
      best_fit(B,w-1)
|| I :  item(I,w,0,0) → Bag_In(B,w)
```

The insertion of *best_fit(B,min(w,c-w))* is necessary to maintain $w \le c$ for *capacity(B,c)*.

The same process can be applied to *Make_Bag(B,w)* to reduce the complex query *MaxWg(w)*. This leads to the introduction of an approximating tuple *max_wg(w)* and a related definition of convergence. 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) ≡                                        (T2.2)
  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 :: ¬item(I,w,0,0)], max_wg(w)†, w > 0
    →
      max_wg(w-1)
|| I :  item(I,w,0,0) → Make_Bag(B,w)
```

Note that *Make_Bag* creates the tuple *best_fit(B,min(w,H-w))*, setting its weight in accordance with the earlier constraints. At this point the initialization requirements become

$$\mathrm{INIT} \Rightarrow \qquad \text{(C4)}$$
$$[\forall\, w,B : (1 \le w \le H) \wedge (1 \le B \le N) :: \mathrm{Make\_Bag}(B,w)]$$
$$\wedge\, [\forall\, w,B : (1 \le w \le H) \wedge (1 \le B \le N) :: \mathrm{Bag\_In}(B,w)]$$
$$\wedge\, \mathrm{max\_wg}(H) \wedge \mathrm{next\_bag}(1)$$

5.2.4 Refinement 4: Eliminating unnecessary 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 does not occur 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 $\underline{inv}\ P \Leftrightarrow T$.

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

$$\underline{inv}\ (max\_wg(w) \wedge next\_bag(B)) \Leftrightarrow Make\_Bag(B,w) \qquad (S8)$$

and after some additional simplifications we obtain

```
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 :: ¬item(I,w,0,0)], w > 0
     →
       Make_Bag(B,w-1)
|| I,B',c' :
       item(I,w,0,0), capacity(B',c'), w ≤ c'
     →
       Make_Bag(B,w)
```

The additional predicates in the third subtransaction query restrict the transaction to recreating itself (in a wait loop) only if the largest unbagged item can still fit into an existing bag. 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

$$\underline{inv}\ best\_fit(B,w) \Leftrightarrow Bag\_In(B,w) \qquad (S9)$$

(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 (S9), *Make_Bag* must create a transaction *Bag_In* whenever it creates a tuple of type *best_fit*. Also, 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 capacity c and the bag position n, the result is 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 :: ¬item(I,w,0,0)], w > 0
     →
       Bag_In(B,w-1,c,n)
```

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

```
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 the *Bagger* program.

The final version of the *Bagger* program is compact, highly concurrent, and executes efficiently on a parallel implementation [13]. 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, we provided only an outline of the derivation steps and omitted all proofs. The strategy is economical, i.e., most proofs involve only small parts of the program or the specification. This is due largely 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 Conclusions

The theme of this paper is 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 [6]. 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. Since we made almost no assumptions about the underlying architecture, we believe the heuristics employed in this task exhibit a high degree of generality and we expect them to be applicable to other emerging parallel rule-based systems.

7 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*, 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*, 1, No. 3, pp. 365-376 (1990).

[6] C. L. Forgy, "OPS5 User's Manual," Technical Report CMU-CS-81-135, Carnegie-Mellon University (1981).

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

[8] 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).

[9] 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)

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

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

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

[13] G.-C. Roman, R. F. Gamble and W. E. Ball, "Formal Derivation of Rule-Based Programs," Technical Report WUCS-91-17, Washington University, St. Louis, MO (1991).

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

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