NAG 1-437
DAA/LANGLEY
IN-63
76299 CR
P.23

Technical Report # TR-87-004

Analysis and Synthesis of Abstract Data Types Through Generalization from Examples

Christian Wild, Ph.D.[*]
Associate Professor
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23508-8508

Abstract

The discovery of general patterns of behavior from a set of input/output examples can be a useful technique in the automated analysis and synthesis of software systems. These generalized descriptions of the behavior form a set of assertions which can be used for validation, program synthesis, program testing and run-time monitoring. Describing the behavior is characterized as a learning process in which general patterns can be easily characterized. The learning algorithm must choose a transform function and define a subset of the transform space which is related to equivalence classes of behavior in the original domain. An algorithm for analyzing the behavior of abstract data types is presented and several examples are given. The use of the analysis for purposes of program synthesis is also discussed.

Categories and Subject Descriptors: I.2.6 [Artificial Intelligence]: Learning; D.2.1 [Software Engineering]: Requirements/Specifications; F.3.1 [Theory of Computing]: Specifying and Verifying and Reasoning about Programs

General Terms: software analysis and synthesis

Additional Key Words and Phrases: Analysis of program specifications, run-time monitoring, theorem learning, transform spaces, automatic programming.

[*] Member IEEE/CS, ACM and AAAI

(NASA-CR-180976) ANALYSIS AND SYNTHESIS OF ABSTRACT DATA TYPES THROUGH GENERALIZATION FROM EXAMPLES (Old Dominion Univ.) 23 p
Avail: NTIS HC A02/MF A01 CSCL 09B
N87-27447
Unclas
H1/63 0076299

1. Introduction

The development of reliable and cost effective software systems is one of the biggest challenges facing the computer science community today. Many people feel that current practice, which is highly labor intensive, is inadequate to the task and that new approaches to software development utilizing increased automation will be required.[1,2] It is our feeling that machine learning can play a significant role in automated analysis and synthesis of software. The research described in this paper is motivated by the belief that people use examples of behavior to analyze and understand complex systems. The approach to learning from examples described in this paper is part of an overall project to increase the reliability of software systems through automation (section 2).

Generalization from a set of examples is viewed as a transformation problem in which one attempts to find an appropriate transformation from the space of input data objects into the transform space such that subsets in the transform space characterize general patterns of behavior (section 3). One algorithm for finding an appropriate transform function and classifying the transform space is also described (section 4). Some results applied to the data type queue are given in section 5. Section 6 discusses some unresolved issues and gives the relation of this work to other work in this area.

2. Generation of the Input/Output Examples

The learning algorithm described in this paper is an integral part of a system to analyze the behavior of abstract data types. Figure 1 shows the major subsystems involved in this system. The specification of an abstract data type is given algebraically.[3] Input/Output examples are generated by symbolically executing the

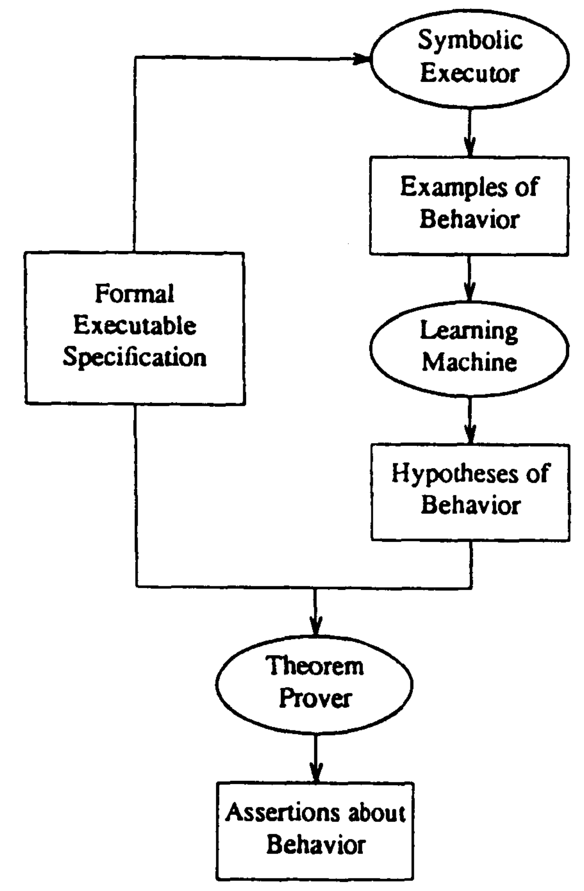

Figure 1 - *Major Subsystems for Automated Analysis*

specifications. The symbolic execution is based on deductive theorem proving techniques.[4] The set of examples is generalized using machine learning techniques described in this paper. These generalizations are proved consistent with the specifications by considering the generalizations to be theorems and the specifications to be a set of axioms. These theorems typically require inductive proofs. Thus, the machine learning subsystem can be considered as a means of guiding the selection of theorems which may be useful in analyzing the behavior of the abstract data types.

The benefits of these theorems are: feedback to the systems analyst for validating the specification, generation of test cases,[5] generation of assertions for run-time monitoring,[4] guidance in program synthesis (see section 6)

Each abstract data type is analyzed individually and is referred to as the **Type Of Interest (TOI)** during its analysis. We assume that the syntax of the TOI is known before analysis. The queue data type, whose specification is given in Figure 2, will be used for purposes of illustration.

```
Type Queue(Integer)

SYNTAX

        Newq                   -> Queue
        Addq(Queue,Integer)    -> Queue
        Deleteq(Queue)         -> Queue
        Frontq(Queue)          -> Integer ∪ {error}
        Isemptyq(Queue)        -> Boolean

SEMANTICS
For all q : Queue; i : integer Let

1)      Isemptyq(Newq)         = True
2)      Isemptyq(Addq(q,i))    = False
3)      Deleteq(Newq)          = Newq
4)      Deleteq(Addq(q,i))     = If Isemptyq(q) then Newq
                                   else Addq(Deleteq(q),i)
5)      Frontq(Newq)           = error
6)      Frontq(Addq(q,i))      = If Isemptyq(q) then i
                                   else Frontq(q)

End Queue
```

Figure 2 - Specification of Queues

The set of functions defining a data type is divided into two subsets. The **generator functions** are those functions of the specification which return a data object of the TOI *(Newq, Addq, Deleteq* in Figure 2). The remaining functions will be called the **behavior functions** because of the important role they play in defining the behavior of the data type[3] *(Isemptyq and Frontq* in Figure 2). When constructing input examples, the TOI data objects and non-TOI data objects are generated differently. Each TOI data object is described in terms of the sequence of generator functions which created that object. Each non-TOI data object is represented by a variable which results from a symbolic execution of the specification. These variables are subscripted, with the subscript denoting the order in which the data object was introduced.

Figure 3 shows six data objects of type queue.

Var I_1, I_2: integer

1) Newq
2) Addq(Newq,I_1)
3) Addq(Addq(Newq,I_1),I_2)
4) Deleteq(Addq(Addq(Newq,I_1),I_2))
5) Addq(Newq,I_2)
6) Deleteq(Addq(Newq,I_1))

Figure 3 - Representative Queue Data Objects

This representation is chosen for the input for two reasons. First, each application of a function is an observable event which constitutes a point at which black box testing, run-time monitoring or record keeping can occur. Second, the string representing this sequence of function applications is a complete abstract representation of the data object. This representation is complete in the sense that it can be used to compute the output value of the behavior functions. This takes advantage of the fact that algebraic specifications are executable.[6]

The smallest subset of the generator functions which can generate every data object of the TOI is called the **constructor function** subset. A data object is in **canonical form** when it is expressed as a sequence of constructor functions only. For the queue data type, *Addq and Newq* are the constructors. In figure 3, example 5 is the canonical form of example 4. A data object and its canonical form are required by the specification to behave identically. The availability of the canonical forms of the input data objects greatly simplifies the learning process.[7]

The behavior of a data type is determined solely by the value returned by the behavior functions.[3] Thus, the output value returned by a behavior function will be referred to as the behavior of the input data object (under the behavior function).

3. The Transformation Problem

Each behavior function of the data type under analysis is considered individually. To simplify this discussion, the behavior function will contain only one input argument of the TOI. Let B represent a behavior function. Let T be the set of all data objects of the TOI and let T' be the subset used as input in the examples (see figure 4). Let P be the image of T under B. Choose $y \in P$ such that $B(x) = y$ for some $x \in T'$. Let S be the inverse image of y under B; that is, $S = \{u \mid B(u) = y\}$ and let S' be the subset of S which is in T'. The set S defines an equivalence class of all data objects of TOI which behave identically under *B*. The goal of the generalization process is to define a characteristic function for S.

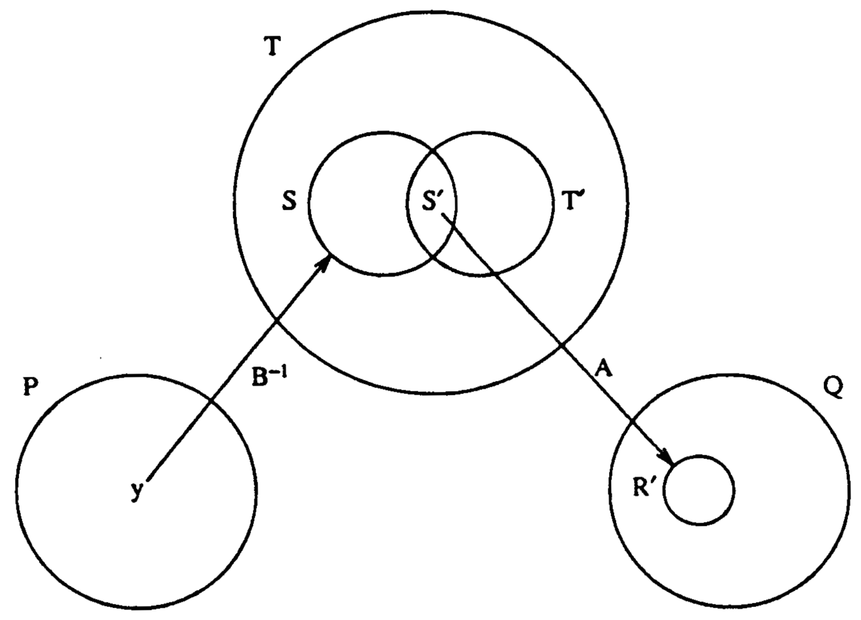

Figure 4 - *Input, Output and Transform Spaces*

Define a transform function, A, on T which can be used to derive a characteristic function for S. Let Q be the transform space (i.e., image of T under A). Let $R' = \{u \mid A(v) = u\ ,\ \text{where } v \in S'\}$. Thus R′ represents the set of values in the transform space which are in the image of examples which have the same behavior (under B). Let D′ be the inverse image of R′; that is, $D' = \{\ u \mid A(u) \in R'\}$. Then, the predicate, $A(u) \in R'$, can be used as a characteristic function (call it C) for D′.

Consider the ways in which D′ is related to S. If D′ = S then

$$\text{FOR\_ALL } X\ (X \in T \Rightarrow (A(X) \in R' \text{ iff } B(X) = y))$$

is the proposed assertion about the behavior of the data type. In this case, C is a necessary and sufficient condition to explain behavior *y*. If $D' \subset S$ then C is only a sufficient condition for $B(X) = y$. If $D' \supset S$ then C is only a necessary condition for $B(X) = y$†. Such weakened conditions can be useful in testing and monitoring.[8] If none of the above relationships hold, then the transform function, A, is not useful in characterizing behavior *y*.

The above approach may be generalized in two ways. The first generalization involves the image set in the transform space (R′). In some cases $D' \subset S$ because R′ has been generated from $S' \subset S$; but, in many cases, a natural extension of R′ (call it R) will have as its inverse image a set D which is equal to S. For example, if the range of A is the set of natural numbers and $R' = \{1..M\}$ where M is the maximum value of the set $\{u \mid A(v) = u \text{ , where } v \in T'\}$, then R′ can be generalized to $\{1..\infty\}$. The characteristic function, $A(u) \in R$, for D should produce a more general description of the behavior of the data type.

The second generalization attempts to handle behavior functions with an infinite image space, P. In this case, it is the characteristic function which is generalized. An example will be shown in section 5.

Given a particular behavior function, B, and a particular behavior value, y, the generalization process involves the following tasks:

- Find a transform function, A. These functions are derived by examining proper-

† $\subset$ and $\supset$ denote proper subset and proper superset respectively.

ties of the observable events (sequence of generator function applications) of input TOI data object. Adaptations of the trace functions defined by Tony Hoare in this book *Communicating Sequential Processes*[9] is one source of transform functions.

- Generate set R′ in the transform space, generalizing to R if appropriate. Define a characteristic function for D′ from R′.
- Determine the relationship between the subsets D′ and S. Since neither D′ nor S is known, the relationship is determined using $D'' \equiv D' \cap T'$ in place of D′ and S′ in place of S.

4. The Algorithm

This section describes one of several algorithms which were investigated. This algorithm starts with a pre-defined set of potentially useful transformation functions which essentially count the number of events (of various kinds) in the TOI input objects. It examines one example and at most one counterexample at a time as it attempts to converge on a characteristic function which is both necessary and sufficient over the sample set (i.e. such that $D' \doteq S$). It fails when it exhausts the pre-defined set of transform functions. Attempts are made to discover a necessary and sufficient characteristic function for every distinct output value. The algorithm then attempts to generalize each set R′.

Let ALL_TRANSFORMS be the pre-defined set of transform functions, CURRENT_SET be a current set of transform functions, A be a transform function, C and C′ be characteristic functions, B be a behavior function, E be an input value of the TOI and y be its corresponding output value. Let E′ be a counterexample input value of TOI and let x be a variable ranging over the set of input values of the TOI.

```
For every distinct output value, y, in the sample set:
    Choose E such that B(E) = y.
    Set CURRENT_SET to ALL_TRANSFORMS.

    Set C to Find-a-Sufficient-Characteristic-Function for E.

    REPEAT UNTIL done
        IF there-exists a counterexample (E′) in the sample set such that B(E′) = y but
        C(E′) is false.
            Set C′ to Find-a-Sufficient-Characteristic-Function for E′.
            IF C′ is true for E then set C to C′
            ELSE set C to C OR C′.
        ELSE done


Where Find-a-Sufficient-Characteristic-Function for x is
    REPEAT UNTIL return
        Choose A from CURRENT_SET.
        Set R′ to the singleton set consisting of A(x).

        IF there does not exist a counterexample (E′) in the sample set such that
        A(E′) ∈ R′ is true, but B(E′) ≠ y
        THEN return A(x) ∈ R′.

        Remove from CURRENT_SET all transform functions such that A(E) =
        A(E′).
        IF CURRENT_SET is empty, then return failure.
```

5. Results

In this section results of applying the generalization algorithm to the queue data type will be presented. A set of transform functions will be chosen somewhat arbitrarily at this point but section 6 discusses this choice in a little more detail. Let #D (#A) be the number of times that Deleteq (Addq) appears in the input. Let #A′ be #A for the canonical form of the input. Let ALL_TRANSFORMS = {#D,#A,#A′}, B = Isemptyq, y = True. Let the set of input examples be those in figure 3. The corresponding output values are all False except for example 1 and 6. Let E be example 1 and A be #D, then $R' = \{0\}$ and $C \equiv \#D(x) \in \{0\}$. C is not sufficient because of the counterexample E' = example 2. Based on this counterexample, CURRENT_SET = {#A,#A′}. Now chose A to be #A, then $R' = \{0\}$ and $C \equiv \#A(x) \in \{0\}$. This is a sufficient characterization of the behavior *True*. However, it is not necessary. Example 6 is a counterexample. A sufficient characterization which explains example 6 is $C' = \#A'(x) \in \{0\}$. This is also necessary and can replace C. Thus the algorithm converges on the following assertions:

if $\#A(x) \in \{0\}$ then $B(x) = True$ (assertion 1)
$\#A'(x) \in \{0\}$ iff $B(x) = True$ (assertion 2)

The first assertion is only sufficient while the second is both necessary and sufficient.

The analysis of *Isemptyq* is unfinished since we have not characterized the behavior value *False*. Choosing E = example 2 and A = #A′ leads to the sufficient characterization $C \equiv \#A' \in \{1\}$. However this is not necessary as indicated by counterexample, example 3. Example 3 has a sufficient characterization $C' \equiv \#A'(x) \in \{2\}$ which is also not necessary. Combining C and C′ (by ORing the conditions) yields the characterization $\#A'(x) \in \{1..2\}$‡. Since 2 is the maximum value returned by any of the examples for transform function #A′, $R' = \{1..2\}$ can be generalized to $\{1..\infty\}$. The following is characteristic function for the subset S (which are all the queues which are not empty):

$\#A'(x) \in \{1..\infty\}$ (assertion 3)

Learning the Behavior of Frontq. Generalizing the behavior of *Frontq* is complicated because there are a potentially infinite number of integers (and hence behaviors)

‡ Union is the OR operator for sets and intersection is the AND operator

returned by this function. In this case, the system must learn a characteristic function schema to describe the behavior. First define a new transformation function as follows:

#D′(x) = number of *Deleteq* function application whose
input argument had #A′ > 0

That is, #D′ only counts the number of *Deleteq*s which actually deleted something from the queue. Let I_1 be the first integer added to the queue after it was created, I_2 be the second, I_n be the n-th integer added. The algorithm generates:

#D′(x) = 0 iff Frontq(x) = I_1 (assertion 4)
#D′(x) = 1 iff Frontq(x) = I_2 (assertion 5)

This can be generalized to

#D′(x) = n-1 iff Frontq(x) = I_n (assertion 6)

Multi-dimensional transform spaces. The analysis of *Frontq* can be used to illustrate specialization in multi-dimensioned transform spaces. This would be necessary if, for instance, #D′ was not in the repertoire of transform functions. First, however, we need to extend the set of transform functions defined previously. Event counting functions can be applied to subsequences of the input, where the subsequences are generated by dividing the input around some **key event.** For instance, the value returned by *Frontq* is traceable to a particular *Addq* event (the key event) during which the returned value was originally inserted. Identifying a key event allows the analysis of the data object both before and after that event. For the queue data type, the key event can be uniquely identified by the variable inserted during the key event. Transform functions which are applied to the subsequence of the input which occurred before the key event are denoted by a subscript of pre/I_n, where I_n is the item added at the key event. For the subsequence which occurred after the key event, the subscript $post/I_n$ is used. To illustrate on example 4 (figure 3), $\#A'_{pre/I_2} = 1$ and $\#D_{post/I_1} = 1$.

In order to illustrate learning using multi-dimensional transform spaces, consider the analysis of the those input queues which return the value I_2 under the behavior function *Frontq*. Let ALL_TRANSFORMS = $\{\#D, \#D, \#A', \#A'_{pre/I_k}, \#D_{post/I_k}, \#A_{post/I_2}\}$, B = *Frontq*, y = I_2. All single dimensional transform spaces using this set of transform functions fail to generate a necessary and sufficient characteristic function. Therefore, it is necessary to explore multi-dimensional transform spaces. These multi-dimensional spaces are formed by conjuncting two or more characteristic functions. In terms of set operations, this is equivalent to checking membership in a set of n-tuples of the relevant transform functions and the corresponding transform values as shown below. Depending on the order in which the transform functions from the set ALL_TRANSFORMS are paired together to form a two dimensional transform space, the following three sufficient characteristic functions can be generated§:

Assertion 7)	$Frontq(Q) = I_2$ if $(\#A', \#A_{post/I_2})$	$\in \{(1,0),(2,1),(3,2),...\}$
Assertion 8)	$Frontq(Q) = I_2$ if $(\#D_{post/I_2}, \#A'_{pre/I_2})$	$\in \{(0,0),(1,1)\}$
Assertion 9)	$Frontq(Q) = I_2$ if $(\#A, \#A')$	$\in \{(2,1),(3,2),(4,3),...\}$

There are many simplifications and generalizations which are possible. For instance, the following simplifications are possible*:

Assertion 7′)	$Frontq(Q) = I_2$ iff $(\#A' - \#A_{post/I_2} = 1)$
Assertion 8′)	$Frontq(Q) = I_2$ iff $(\#D_{post/I_2} - \#A'_{pre/I_2} = 0)$
Assertion 9′)	$Frontq(Q) = I_2$ iff $(\#A - \#A' = 1)$

§ These are generated from a larger set of examples than those shown in figure 3.

* These simplifications and generalizations are not currently programmed into the system but are based on straightforward arithmetic and algebraic knowledge.

By examining further behavior results (for I3, I4, etc.) and by further generalizations, the following hypothesis could be produced:

Assertion 7″) Frontq(Q) = I_n iff ($\#A' - \#A_{post/I_n} = 1$)
Assertion 8″) Frontq(Q) = I_n iff ($\#D_{post/I_n} - \#A'_{pre/I_n} = 0$)
Assertion 9″) Frontq(Q) = I_n iff ($\#A - \#A' = n-1$)

Hypothesis H10″ is another way of saying that the integer I_n is at the front of the queue if exactly every item previously in front of it has been deleted so that there is precisely one more item in the queue than was added after I_n. That one more item is I_n. Hypothesis H11″ states that all the items in the queue when I_n was added have subsequently been deleted. Hypothesis H12″ states that if n-1 items have been deleted from the queue, then I_n must be at the front. These different assertions are represented pictorially in figure 5.

6. Discussion

Generating Transform Functions. The success of the learning process depends on several factors, the most critical of which is the choice of useful transform functions. A good transform function will key in on essential features of the input while ignoring the irrelevant details. Many of the transform functions are derived rather naturally from the input, such as counting the number of occurrences of certain kinds of events. This class of transform functions ignores the non-TOI arguments in the input as well as the order of function invocation. Identifying a key event allows the analysis of those subsequences of events before and after the key event.

Another class of transform functions are those which are concerned with the order in which events occur. For example, what was the last event, or the last key event

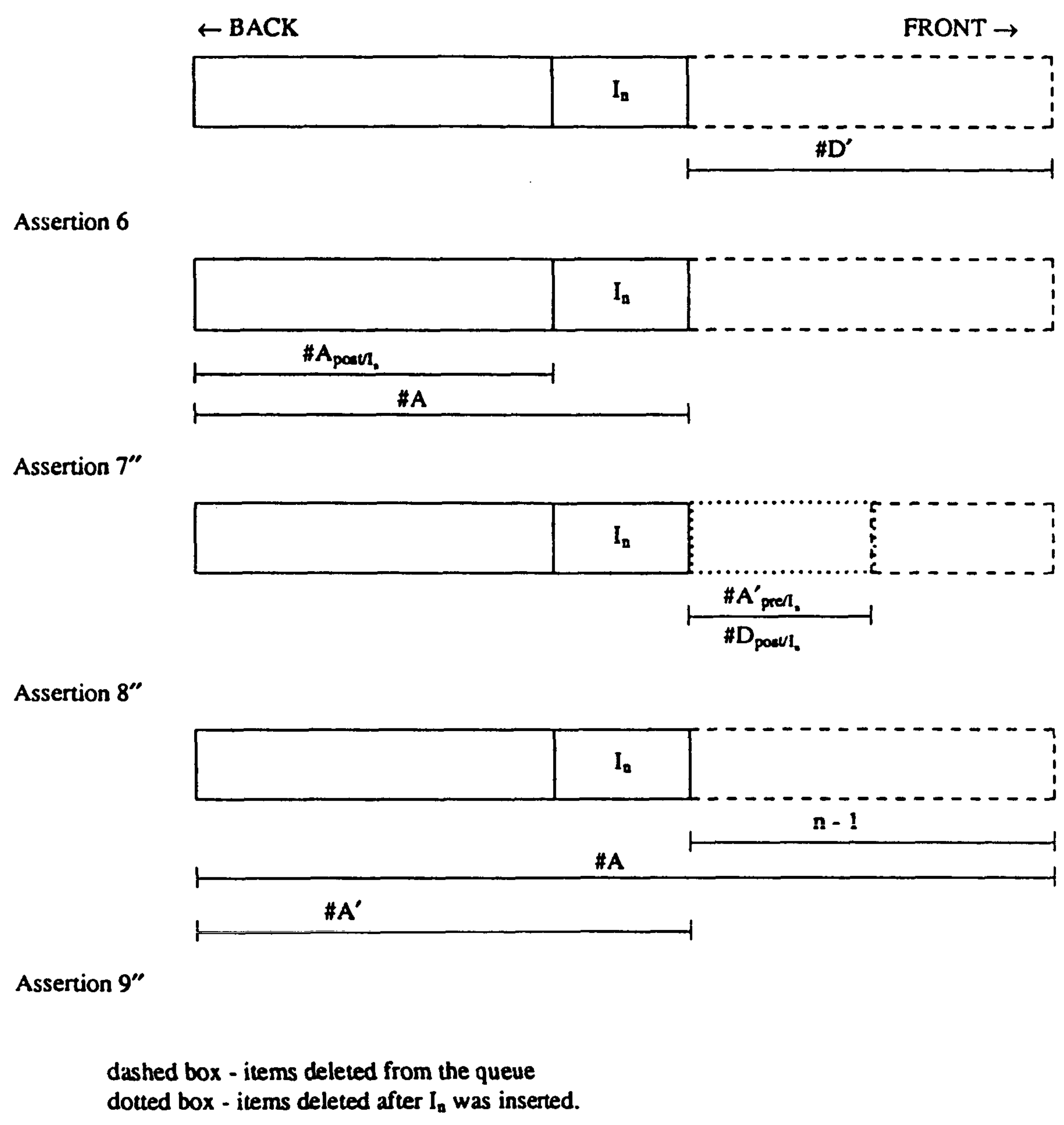

Figure 5 - *Conditions Under Which* I_n is at Front of Queue

containing a particular generator function? As another example, the transform function could return the sequence of generator function applications, ignoring the non-TOI arguments. This class of functions would be useful in analyzing multi-dimensional data structures such as trees and graphs. As might be expected, generalization in this transform space would be more difficult.

New transform functions can be formed from existing ones by conjunction. Such a combination forms a higher dimensional transform space which is more specialized than the original transform spaces. In the algorithm implemented, these higher order transform spaces are explored only after all the lower order transform spaces have been eliminated.

Generalizing in the Transform Space. The kind of generalizations possible in the transform space depends upon the nature of the transform space itself; however, many of the generalization rules defined in the literature[10, 11] can be applied. As described previously in the case of *Isemptyq*, generalization by internal disjunction[12] can combine several sufficient conditions into a necessary and sufficient one. Disjunction is sufficiency preserving†. For generalizing in multi-dimensional transform space, techniques for discovering invariant relationships in a set of quantitative data, such as available in BACON[13] may be useful.

Comparison to other Learning Techniques. Learning can be viewed as the formation of general concepts through the examination of specific examples which illustrate that concept. For program analysis, the concepts to be learned are the behavioral relationships between the input and output values (under some behavior function). Cohen and Sammut[14] point out that concepts are normally thought of as recognition devices, whereas programs are considered generators of output from given input. However

† Because ((A => B) & (C => B)) => ((A or C) => B)

associated with every program is its specification which is a predicate which recognizes whether the output is within the concept illustrated by that program‡. Thus programs are devices for filling in the blanks (the output values) which satisfy the concept. For program synthesis, concept recognition must be both necessary and sufficient (as in Cohen and Sammut). Kodratoff and Ganascia,[11] however, require only that the concept recognition function be sufficient. For program analysis, concept recognition functions can be either necessary or sufficient or both.[8]

Many learning systems start with descriptions of the examples which are complete and maximally specific. In many cases, the descriptions are represented as a conjunction of predicates. These descriptions can be generalized in several ways:

1) Dropping condition rule - one (or more) of the conjuncts are dropped from the description.[12]

2) Class generalization - if a classification taxonomy exists for the domain under analysis, then parts of the description representing more specific objects can be replaced by one of the classes of which that object is a member.[15]

3) Folding§ - parts of a description can be replaced by a concept already learned.[14, 16]

4) Disjunction of concepts - a disjunction is always true in at least as many cases as the constituent concepts.

For the analysis of abstract data types, the transform functions and values can be used to generate a description. However, because the number of transform functions can be

‡ PROLOG programs are always written in this predicate form.

§ Folding is a term used in program transformations whereby the body of a program is replaced by a call to the program. This is the opposite of macro expansion.

infinite, the description may not be complete (at least not finite and complete)*. For this reason, the dropping condition and folding approaches to generalization are not used, since they traditionally require complete descriptions. Due to a lack of a concept taxonomy, this method is also ruled out. Both internal and external disjunction[10] are used to generalize the assertion as described previously, but this is done only after an initial concept has been formed.

The formation of this initial concept uses a method of generalization different from the four methods mentioned above. The choice of one transform function in effect generalizes the concept to include all the input examples which are in the inverse mapping of the transform value of the input example under consideration. An attempt is made to discover if this concept is too general by looking for a counterexample to the sufficient characteristic function. The active searching for an example which fails to satisfy the concept has been called expectation-based filtering.[17] If a counterexample is found then it is used to guide the selection of a new transform function. This helps the algorithm either to find a suitable transform function or to fail more quickly than might be possible using a uninformed search. If none of the transforms function singly are sufficient, then the algorithm adds conditions to form multi-dimensional transform spaces. Thus the formation of a concept is a combination of a selecting conditions rule and an adding conditions rule. The Concept Learning System (CLS) used in ID3 also starts with a generalized hypothesis and specializes it

* An infinite number of transform functions can result is there are an infinite number of key events - each generating its own transform function. An infinite number of key events can result from data types which have a non-TOI input argument which can take on an unbounded number of values. For queues, the number of integers which can be entered into the queue (and thus used to define key events) is potentially infinite.

by adding conditions.[18]

Considering each value returned by a behavior function as a concept to be learned, the analysis of data types is a multi-concept learning problem. The individual concepts are learned separately and then merged together. Sometimes, as in learning the two concepts for *Isemptyq*, the concepts are merged disjunctively (see program synthesized for *Isemptyq* later on in this paper). However, in the case that there are an infinite number of concepts (as there are for *Frontq*), the generalization takes the form of a concept schema†.

To close this section, it should be pointed out for this problem the training examples are noise free and that the inductive theorem prover is the ultimate arbiter of the correctness of any learned concepts.

Comparison to Program Synthesis by Examples. Although the orientation of this research is analysis instead of synthesis, analysis is an important component in generating efficient implementations. The two assertions learned about the behavior function *Isemptyq* could be easily converted into a program. Also, the assertion learned about *Frontq* can provide insight into its implementation. So it may be useful to compare this work to that on program synthesis by example.

There is an extensive body of literature in the synthesis of LISP programs by example.[19] However the general problem area is different. Most work in the literature deals with pre-defined data structures (such as those given in LISP) and tries to synthesize functions which manipulate those data structures. The generator and behavior

† As programs, a concept schema is represented as a fixed code segment and a dynamically sized data structure.

functions of the underlying data type are used as primitives in the generalization process. By contrast, the programming of abstract data types requires the synthesis of its generator and behavior functions.

To explore this difference some more, let's examine how the assertions learned by our approach can be used to synthesize programs. In order to use the learned assertions about *Isemptyq* as a program, the transform function, #A′, must be computable. Synthesizing the implementation of #A′ is easy (in this case) by examining the input examples for the effect that each generator function has on the value returned by #A′. Let N_Addq be a count of the number of occurrences of *Addq* in the canonical representation of a queue, then

```
Newq: sets N_Addq to 0 (initialization)
Addq: increments N_Addq by one.
Deleteq: if N_Addq is not 0 decrements N_Addq by one
```

Recognizing that assertions 1 and 3 form a partition of the transform space of #A′, it would be easy to generate the following program for *Isemptyq*:

```
IF N_Addq = 0 THEN True
ELSE False
```

The interesting result is that the code for implementing *Isemtpyq* is distributed among four functions. This supports the statement made earlier that the behavior of an abstract data type is determined solely by the value returned by the behavior functions. *The only code required in the generator functions is that required to implement the behavior functions.* This distributing of code for the behaviors among the generators is the major way in which the implementation for abstract data types differs from the implementation for functions which manipulate, rather than define, data structures.

There are systems for synthesizing implementations for abstract data types reported in the literature[20,21] but these systems manipulate the specification directly. Examples are not used in the synthesis process.

7. Conclusions

This paper presents an application of learning from examples to the analysis and synthesis of abstract data types. The generalization process searches for a mapping from the input space into a transformation space in which a characteristic function can be generated. This characteristic function defines a subset of the input space which can be equal to, more specific than or more general than the subset of the input space which belongs to the concept being learned. For abstract data types the concept to be learned is defined in relationship to the equivalence class of input objects which exhibit the same behavior.

An algorithm for searching through a pre-defined set of transform functions is given and compared to related literature in machine learning. Results from an analysis of the queue data type are presented and demonstrate the possible application of this method to program synthesis. Some unique characteristics of synthesis for abstract data types are discussed.

Acknowledgements

This research was supported under NASA grant NAG 1-439. I would like to thank Dave Eckhardt, the grant technical monitor, for suggesting the application of Artificial Intelligence methods to the production of reliable software systems.

References

1. Balzer, Robert, Cheatham, Thomas, and Green, Cordell, "Software Technology in the 1990's: Using a New Paradigm," *Computer*, pp. 39-45, Dec. 1983.
2. Scherlis, William and Scott, Dana, "First Steps Towards Inferential Programming," *Information Processing 83*, vol. 9, no. ISBN 0-444-86729-5, pp. 199-212, Elsevier Science Publishers, NY, Sept. 1983.
3. Guttag, J. and Horning, J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pp. 27-52, 1978.
4. Wild, Christian, Eckhardt, Dave, Pang, Ava, and Sundararajan, Saraswathi, Analysis of Executable Specifications for Testing and Monitoring Abstract Data Types, Department of Computer Science, Old Dominion University, Norfolk, VA, March 1986.
5. Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M.C., "Application of PROLOG to Test Sets Generation From Algebraic Specifications," *Formal Methods and Software Development*, vol. 2, no. ISBN 3-540-15199-0, pp. 261-275, Springer-Verlag, Berlin, March 1985.
6. Goguen, Joseph, "Some Design Principles and Theory for OBJ-0, A Language to Express and Execute Algebraic Specifications of Programs," *Mathematical Studies of Information Processing*, vol. 75, no. ISBN 3-540-09541-1, pp. 425-473, Spring-Verlag, NY, 1978.
7. Wild, Christian, "Learning the Behavior of Software Systems from Executable Specifications," *Proceedings of OAST Technical Symposium on Computer Science and Data Systems*, no. ODU-TR-CS-86-011, November, 1986.
8. Wild, Christian, "Automating Software Fault Tolerance," *Journal of Spacecraft and Rockets*, vol. 24, no. 1, pp. 86-89, January/February 1987.
9. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall International, Englewood Cliffs, NJ, 1985.
10. Michalski, Ryszard, "A Theory and Methodology of Inductive Learning," in *Machine Learning: An Artificial Intelligence Approach*, ed. Ryszard Michalski, Jamie Carbonell and Tom Mitchell, vol. 1, pp. 83-134, Morgan Kaufmann, Los Altos, CA, 1983.
11. Kodratoff, Yves and Ganascia, Jean-Gabriel, "Improving the Generalization Step in Learning," in *Machine Learning*, ed. Ryszard Michalski, Jaime Carbonell and Tom Mitchell, vol. II, pp. 215-244, 1986.
12. Dietterich, Thomas and Michalski, Ryszard, "A Comparative Review of Selected Methods for Learning from Examples," in *Machine Learning: An Artificial Intelligence Approach*, vol. 1, pp. 41-82, Morgan Kaufmann, Los Altos, CA, 1983.
13. Langley, Pat, Zytkow, Jan, Simon, Herbert, and Bradshaw, Gary, "The Search for Regularity: Four Aspects of Scientific Discovery," in *Machine Learning: An Artificial Intelligence Approach*, ed. Ryszard Michalski, Jaime Carbonell and Tom Mitchell, vol. 2, pp. 425-469, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

14. Cohen, Brian and Sammut, Claude, "Program Synthesis Through Concept Learning," in *Automatic Program Construction Techniques*, ed. Alan Biermann, Gerard Guiho and Yves Kadratoff, pp. 463-482, Macmillan Publishing, New York, 1984.

15. Mitchell, Tom, Utgoff, Paul, and Banerji, Ranan, "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," in *Machine Learning: An Artificial Intelligence Approach*, ed. Ryszard Michalski, Jaime Carbonell and Tom Mitchell, vol. 1, pp. 163-190, Morgan Kaufmann, Los Altos, CA, 1983.

16. Sammut, Claude and Banerji, Ranan, "Learning Concepts by Asking Questions," in *Machine Learning: An Artificial Intelligence Approach*, ed. Ryszard Michalski, Jaime Carbonell and Tom Mitchell, vol. 2, pp. 167-191, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

17. Lenat, D.B., Hayes-Roth, F., and Klahr, P., "Cognitive Economy in Artificial Intelligence Systems," *IJCAI*, vol. 6, pp. 531-539, 1979.

18. Cohen, Paul and Fiegenbaum, Edward, "Data-driven Rule-space Operators," in *The Handbook of Artificial Intelligence*, vol. 3, pp. 401-410, William Kaufmann, Inc., 1982.

19. Smith, Douglas, "The Synthesis of LISP Programs from Examples: A Survey," in *Automatic Program Construction Techniques*, ed. Alan Biermann, Gerard Guiho and Yves Kadratoff, pp. 307-324, Macmillan Publishing, New York, 1984.

20. Bartels, U., Olthoff, W., and Raulefs, P., "APE: An expert system for automatic programming from abstract specification of data types and algorithms," *IJCAI-81*, pp. 1037-1043, 1981.

21. Moitra, Abha, "Direct Implementation of Algebraic Specification of Abstract Data Types," *IEEE Trans. on Soft. Eng.*, vol. SE-8, no. 1, pp. 12-20, Jan. 1982.