Design and Implementation of the Formal Specification Acquisition System SAQ*

Dong Yunmei, Li Kaide, Chen Haiming, Hu Yongqian,
Zhang Ruiling, Tang Ruqing, Wan Zhanyong and Chen Ziming
Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing 100080, China
Email: {dym,lkd,chm,hyq,zrl,trq,wzy,czm}@ox.ios.ac.cn

Abstract

The formal specification acquisition system SAQ is designed to assist users to acquire and validate specifications with support of specification bases. In SAQ, specifications are represented by context-free languages and recursive functions defined on context-free languages, which are called concepts and operations defined on concepts respectively. Acquisition of specifications is completed by human-machine cooperation, based on specification reuse. Concepts are validated by sample recognition and generation, and operations are validated by applying operations on samples of concepts. The design and key implementation techniques of SAQ are introduced, experiments are presented.

Key words: formal specification, acquisition, validation, grammar learning algorithm, recursive function evaluation.

1 Introduction

Software automation is the radical way to improve software productivity. Most of previous researches in this area focus on how to derive executable programs from formal specifications. However, one more fundamental issue of software automation, which is also one of the basic issues of software engineering, is how to obtain formal specifications and make sure that specifications meet actual requirements. This is a rather difficult problem, and researchers began to pay much attention to it in recent years.

Requirements of systems often are hard to be completely and clearly stated at a time. Yet user's requirements about system's functionalities are under continuous changing. We have been developing a methodology for specification acquisition, i.e. the MLIRF method [3], the ultimate goal of which is to study how to assist human users by computer, through human-machine cooperation, to develop precise, complete and consistent formal specifications, which are approved by human users through verification and then used as the starting point of software design and implementation, from human users' vague, incomplete and inconsistent informal statement of needs about target problems, together with, and making full use of, known specification knowledge. A complete solution to it is still years away. This paper presents what we have achieved at this direction.

We realized that to ease the acquisition of complex specifications, it is important to be able to construct specifications based on reuse of known specifications. In this method, we use formal languages and a new kind of recursive functions to represent specifications. Acquisition of specifications is completed by human-machine cooperation, based on specification reuse.

We have designed and implemented an environment SAQ (Specification AcQuisition system) that implements MLIRF method. It assists users to acquire and validate specifications with support of specification bases [1, 2]. The present focus is on verifying the method and its related implementation techniques. Other relevant work like automatic verification tool have not been set about.

Many trivial and nontrivial experiments have been done successfully. These experiments indicate that SAQ is particularly suited for the problem class which has complex syntax structure, such as the automatic conversion problem of programming languages.

There have been some other work on specification acquisition. Most of them are domain dependent, and domain knowledge plays a key role there. For example, WATSON [9] applies to reactive systems, SPACE [8] operates in business data processing sys-

*This research was supported by the National "863" Hi-Tech Programme, the National Natural Science Foundation, and the National "Ninth-Five" Sci-Tech Programme of China.

tems, and [10] describes method for obtaining specifications of communication services. The techniques adopted by existing work include logical inference, object-oriented model, psychological and human performance model, and so on. SAQ is not domain dependent, and provides new representation for specifications and reuse-based learning method for specification acquisition.

Related with SAQ's specification is the algebraic specification, which is composed of signatures and equations. However, recursive functions can be defined by structural induction method in the assistance of machine, and evaluation of recursive functions can be implemented more efficiently then term rewriting for algebraic specifications.

In this paper the design, implementation and application of SAQ will be introduced. Section 2 describes the function and structure of SAQ. In section 3 the key implementation techniques are presented. A nontrivial example is described in section 4 for the illustration of using SAQ. Experiments and future work are sketched in section 5.

2 Design of SAQ

2.1 Specification Representation and Acquisition

We split a specification into two parts: problem space and operations defined on it, which respectively correspond to syntax and semantics of the specification. In SAQ, problem space consists of *concepts*. A concept is a set, i.e. the set of all samples of the concept. We are especially interested in the following case: the elements of a set are expressions to describe samples of a concept, in other word, an element is a sentence. Therefore, concept is a language. Moreover, we just consider context-free languages (CFLs). Tightly connected with concepts are *operations*, which are defined on these concepts. Operations are represented by recursive functions defined on context-free languages (CFRFs) [5]. Therefore the semantics is a description of "how to do" instead of "what to do". But, within a framework of functional language (CFRF), it avoids many implementation details.

Specification is a formal representation of concepts and operations.

Specification acquisition is to acquire grammar definitions of concepts and the effective (executable) definitions of operations. Acquisition of the description of problem space, i.e. the concepts, is realized by inference from a few typical instances of the problem by human-machine interaction and reuse of specification knowledge. In particular, concepts will be obtained by grammar acquisition methods. For this purpose a reuse-based new method for context-free grammar acquisition is presented. Because of the inherent difficulties in the inference of semantics, the semantics of specification is instead offered by user in the assistance of machine. The key point is, complete definition of the semantics is assured by the assistance of machine.

Specifications are validated by testing. Concepts are validated by sample recognition and generation, and operations are validated by applying operations on samples of concepts.

2.2 Function of SAQ

SAQ assists users to acquire and validate specifications by human-machine cooperation, with support of specification bases. SAQ provides the following facilities:

- Hypertext online help. Users can learn general or specific information of the system, or operating instructions at any time.
- Menu driven. Users can work with the system by selecting one of the operations available at that time.
- Specification base management. Creation: There should be core material in a base. So a base creation tool is necessary to import the minimal contents into bases. Management: Base management. The key problem is rapid access to large-scale specification base. Use: Users can browse or retrieve data in specification bases at any time. The operations on specification bases can be invoked either by users, or by other subsystems.
- Concept acquisition. Essentially it is the inductive learning of CFL.
- Operation acquisition. Essentially it is the definition and evaluation of CFRF.
- Specification validation. Generate samples from concepts, and check if a sample belongs to a concept; apply operation on samples.

2.3 Structure of SAQ

The structure of SAQ is illustrated in Figure 1.

SAQ consists of the following components.

1. Human-machine interface. All human-machine interactive operations and management of the system are realized through this interface.

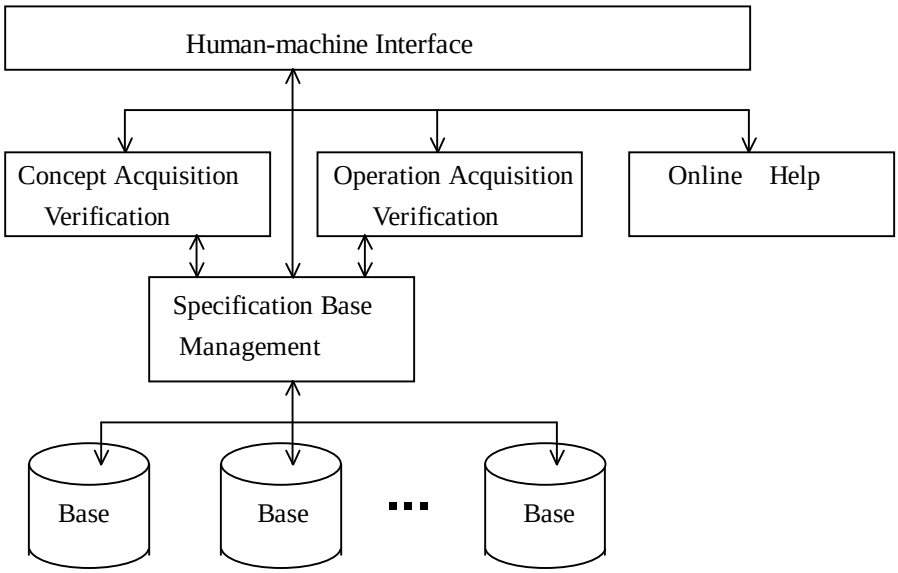

Figure 1: Structure of SAQ

2. Concept acquisition and verification system. It implements grammatical inference of concepts and sample generation and recognition.

3. Operation acquisition and verification system. It implements definition and application of operations.

4. Specification base management system. It carries out management, reuse, browse and retrieval of data in specification bases, and includes base creation tools.

5. Hypertext online help system. It provides an online help manual that can be browsed in hypertext manner.

3 Implementation

In this section the key implementation techniques of SAQ are introduced.

3.1 Notation Convention

Vocabulary $V = V_N \cup V_T$, where V_N is a set of nonterminals, its elements will be denoted by capital letters (may have subscripts) as $X, Y, Z, X_0, Y_0, Z_0, \ldots$; V_T is a set of terminals; $V_N \cap V_T = \Lambda$. V_T^* denotes the set of all strings of terminals (including empty string λ), its elements will be denoted by lower case letters $a, b, c, \ldots$; V^* denotes the set of all strings of characters of V, its elements will be denoted by capital letters P, Q, R. The grammar of X is denoted by $G(X)$.

3.2 Reuse-Based Interactive Learning of Concepts represented as CFL

Concept learning or acquisition is a human-machine interaction process, which needs to utilize the available knowledge (specification base reuse), learning by induction, i.e. from particular to general, to generalize the samples of concept. The results must be confirmed.

Gold indicated in his paper [7]: If both positive and negative samples are offered, then the CFL is identifiable in limit; if just the positive samples are offered, then any language class contains infinite many sentences and cannot be identifiable in limit. After then, many authors study the problem of identification of CFL and their subclass from different viewpoint [11]. This topic is the so-called inductive learning of lan-

guage or grammatical inference.

SAQ cannot take the known learning algorithms, since SAQ requires the algorithm with features:

- There exist many grammars for a given languages, and the intended one has to represent structure within the concept and subconcept, so the meaning of concept (semantics) can be described in a natural way. We do not want a general algorithm, to acquire a grammar whatever to define the language, without meaningful structure; neither a special algorithm just suitable for particular subclass of context-free grammar, such as precedence grammar or regular set.
- It should work efficiently, acquire in finite time, know when to finish acquisition process, and be effective on computer.

For SAQ, user's taking part in learning process does not mean a bother but necessity. User aided decision making will reduce searching space, and raise learning efficiency and quality.

3.2.1 The Overall Process

We proposed a reuse-based interactive learning process for concepts represented as CFL, based on the algorithm described in [4]. It does concept acquisition by interaction with user (called seeker) to aid decision-making, with support of specification bases. In the learning process, final grammar of a concept is acquired by given a finite set of samples of the concept, and some known concepts (i.e. their grammars are known) may be used in the acquisition, which is the reuse of concepts. A group of interrelated concepts will be acquired in one learning process, generally one of which is the main concept, and the others are auxiliary concepts subordinated to the main one. The learning process encompasses five succeeding steps: conjecture, conjecture confirmation, conjecture refinement, empty word process, and redundancy elimination.

(1) Conjecture. The process starts by making conjecture according to the samples of unknown concepts and grammars of known concepts given by the seeker. The central point is to generalize samples into conjectures. All conjectures for each sample will be found out. Details of this step will be described later.

(2) Conjecture confirmation. After all conjectures have been made, next step is to decide whether to accept or reject each conjecture, which needs the seeker's aid. The seeker will reply questions that cannot be decided by the system.

(3) Conjecture refinement. The accepted conjecture set will be refined. Actually it is a re-generalization of conjectures that will construct more refined productions.

(4) Empty word process. Empty word (i.e. empty string) can both be given as a sample of an unknown concept and occur in the production of a known concept, which is not considered in conjecture and refinement steps. Empty word will be processed with the refined production set.

(5) Redundant elimination. Before reach the final results, redundancy elimination is done. For each production in current conjecture set, if it can be derived from other productions in the set, then the seeker will be asked whether this production will be eliminated or not.

In the following we will describe the way of conjecture in detail.

3.2.2 Conjecture Process

The finite set of given samples of an unknown concept X is denoted by $Sample(X)$. A *term conjecture* of a sample s with respect to unknown concept X is a production in the form $X \to P$ ($P \in V^*$). For each nonterminal (i.e. concept name) Y in P, there exists a string in $Sample(Y)$ (if Y is unknown) or a valid sentence of $G(Y)$ (if Y is known), such that when all concept names in P are replaced by the corresponding strings or sentences, then P becomes the sample s. If there is no nonterminal in P, then $X \to P$ is called a *trivial term conjecture*. The set of the right-hand side of all term conjectures of a sample s with respect to concept X is denoted by $Term(X, s)$.

$$Term(X, s) = \{P \mid X \to P \text{ is a term conjecture of } s\}$$

An allowable decomposition $s = abc$ of terminal string s is a decomposition of s into three subwords a, b, c, such that b is not an empty word; a and c are not empty words at the same time; and there is an unknown concept X such that $b \in Sample(X)$, or there is a known concept Y such that b is a valid sentence of $G(Y)$. Denote $ADecomp(s)$ the set of all allowable decompositions of s. An allowable decomposition abc of s is in $ADecomp(s)$ if (1) there is an element $a_1b_1c_1$ in $ADecomp(s)$, such that $a = a_1,\ b = b_1,\ c = c_1$, or (2) there is an element $a_1b_1c_1$ in $ADecomp(s)$, such that a_1 is the head of a, c_1 is the tail of c [1] (b must be a proper subword of b_1, i.e. b occurs in b_1 and $b \neq b_1$).

The Cartesian product of $T_1 = \{Q \mid Q \in V^*\}$ and $T_2 = \{R \mid R \in V^*\}$ is

$$T_1 \times T_2 = \{QR \mid Q \in T_1,\ R \in T_2\}.$$

[1] For three words a, b, c, if $a = bc$, then b is called a head of a, b is called a proper head of a while c is not an empty word; c is called a tail of a, c is called a proper tail of a while b is not an empty word.

The objective of conjecture process is to construct $Term(X, s)$ for all $s \in Sample(X)$. This process is divided into three steps and each s is dealt with by an order of size from short to long.

(1) If a trivial term conjecture is wanted (this case should be confirmed by seeker), then

$$Term(X, s) = \{s\}.$$

(2) If a nontrivial term conjecture is wanted, then find an allowable decomposition a, b, c of s. According to this decomposition, a nontrivial term conjecture aX_bc is constructed, and becomes an element of $Term(X, s)$. If X_b is unknown, then $b \in Sample(X_b)$; otherwise b is a valid sentence of $G(X_b)$.

There might be several allowable decompositions of s. It is unlikely all the term conjectures formed from them are desirable. In particular, if there are two different allowable decompositions $s = abc$ and $s = a_1b_1c_1$, where a is the head of a_1, c is the tail of c_1, then the decomposition $s = a_1b_1c_1$ is denied. Furthermore, the seeker can deny a term conjecture directly, or request to generate a set of samples from the term conjecture for confirmation.

Maybe there dose not exist any allowable decomposition, it means that the sample set is not rich enough, or the current concept system is not complete yet, not a self-contained one. At this time, what need to do is to supply new samples or introduce auxiliary concept.

It is possible that the seeker would rather give his(her) own decomposition and introduce auxiliary concept, even though there exist allowable decompositions already.

(3) For a in allowable decomposition, assume a temporary nonterminal X_a, deny it after this step completed. If a is not an empty word, then let $Sample(X_a) = \{a\}$, and use the algorithm stated here to find out $Term(X_a, a)$. If a is the empty word, then let $Term(X_a, a) = \{\lambda\}$. Similarly, to find out $Term(X_c, c)$ for c. Finally put elements of set

$$\{T_aX_bT_c \mid T_a \in Term(X_a, a),\ T_c \in Term(X_c, c)\}$$

into $Term(X, s)$, to replace the original term conjecture aX_bc.

3.3 Definition and Application of Operations

Operations are so-called CFRFs, whose operands have CFL structures. Unlike other recursive functions, both definition and evaluation of CFRF should be carried out according to the structure of CFL. In concrete, definition is accomplished by structural induction, and evaluation needs structural analysis. The inductive definition of operation can be assisted by machine, which makes definition easier and guarantees the definition of operation is complete.

3.3.1 Definition Process

The basic definition method of operation is structural induction on CFLs. The user is allowed to define operations either directly or interactively. For simple operation, the user can compose its definition directly. Otherwise, the user can construct the definition interactively in the assistance of the system. It is not difficult to implement the interactive construction of operations.

The basic way to define operations is mutually recursive definition, i.e. to define several interrelated operations simultaneously.

The process to define a set of operations is as follows.

At first there are a set of known concept names denoted by S and a set of being defined operation names denoted by F. The operations in F are defined one by one.

When defining an operation, if the operation is complex, then the user interactively constructs its definition. If the user thinks that he can write the definition directly, then he inputs the definition in an editor window. The definition given directly by user should be in FDL, the operation definition language of the system. The answers offered by user in interactive construction are expressions in FDL. Complete definition will be formed after the interaction succeeds.

Validity checking is made each time the user gives a definition directly or gives an expression interactively. Invalid definition or expression is denied.

For new operations (those do not defined yet and are not in F) that occur in a definition or an expression, the user is asked to give their domains and ranges, and their names are added into F. If there are new concept names in the domains or ranges just given, then these concept names are added into S.

At last, when all operations in F are defined, the definition process completes.

3.3.2 Operation Application

To apply operations on samples of concepts is to evaluate CFRFs.

Parsing CFRF does not limit CFLs to any subclass, and empty word is permitted. So a general CFL parser capable of dealing with empty word is required. One competent algorithm is the Earley's algorithm [6]. Earley's algorithm generates the right parse of

a sentence, represented as a sequence of numbers of productions. For a sentence with length of n, the worst case of parse time is $O(n^3)$.

Evaluation of CFRF As stated above, evaluation of CFRF is closely related with structural analysis or parsing. Based on the selected parsing method, an evaluation algorithm for CFRF is introduced below.

Assume a n-ary function $g : L_1 \times \ldots \times L_n \rightarrow L$ is defined by structural induction on L_1, and L_1 is defined by $G_1 = (V_N, V_T, X_1, P_1)$.

For sentences $u_1, \ldots, u_n$ of $L_1, \ldots, L_n$, when evaluate $g(u_1, \ldots, u_n)$, first parse u_1 according to the grammar of L_1, to generate a right parse $i_1 i_2 \ldots i_m$, represented by the numbers of productions in P_1. The right parse of u_1 would be one of the following two cases:

- Only one production i_1. At this time production i_1 must be $X_1 \rightarrow u_1$. From the definition of g find out the equation whose first parameter is u_1 (this equation exists according to the structural induction method), and replace the variables in the right-hand side expression e of that equation by the binding values. If in expression e there is no other function except concatenation, then evaluation is completed, otherwise continue to evaluate such functions.

- More than one productions exist. At this time production i is $X_1 \rightarrow P$. Suppose P contains nonterminals $Z_1, \ldots, Z_r$, it is easy to calculate from $i_2 \ldots i_m$ the corresponding right parses and values of the nonterminals. From the definition of g find out the equation whose first parameter matches the structure of P, and replace the variables in the right-hand side expression e of that equation by the binding values, moreover, the right parses of $Z_1, \ldots, Z_r$ are kept in the corresponding variables. If in expression e there is no other function except concatenation, then evaluation is completed, otherwise continue to evaluate such functions.

3.4 Structure and Management of Specification Base

3.4.1 Data in Specification Bases

Three kinds of data are stored in a specification base: fixed-length data, such as name (concept name or operation name), type, author, date, author's address, version, abstract, and so on, which we refer as record; association, which is variable-length data that indicate the reference information of concepts and operations; concept or operation definitions, which are also variable in length.

A group of definitions called definition group is stored as the basic unit in specification base. A definition group of concepts consists of a group of interrelated concepts. A definition group of operations consists of a group of mutually defined operations.

3.4.2 Structure of Specification base

Databases can be classified into relational, hierarchical, and network databases, according to data relations. Data relation in the specification bases is so intricate that it is rather difficult, and certainly inefficient, to organize data by relational model. According to the characteristics of specification bases, a restricted network structure is used. We treat the relations among definition groups as a directed acyclic graph, relations among concepts or operations within a definition group is not restricted.

3.4.3 Management of Specification Base

The management of specification base entails different scenarios for different kind of data. In brief, extensible hashing combined with partial-match retrieval and indexed descriptor file is imposed on the management of record and linked table is used to organize both definitions and associations.

Extensible hashing is a fast access method based on hash function, it has a dynamic structure that grows and shrinks gracefully as the records are inserted or deleted, and can handle the direct access to records efficiently.

Moreover, partial-match retrieval is adopted to construct a hash mapping (called pseudokey) out of multiple hash mappings stemmed from multiple keys in one record as the case is in our specification base. For this aim, first, for a record with n keys $R = (k_1, k_2, \ldots, k_n)$, let h_i be a hashing function from the key space K_i of the field k_i to the hashing mapping space H_i, thus $h_i : K_i \rightarrow H_i$ $(i = 1, \ldots n)$. The next step is to form a choice vector $CV = (C_1, C_2, \ldots, C_m)$ (m is maximal length of pseudokey). A simple, reasonable rule to form CV is given as following: C_i $(1 \leq i \leq m)$ is decided by the retrieval rate of key k_i, C_i is relatively big for high retrieval frequency of k_i, but C_i is set to 1, no matter how often the key k_i are queried, if k_i is a boolean value. Then we assemble pseudokey using choice vector: the ith position of pseudokey will be filled by the first so far unused bit in $h_{c_i}(k_{c_i})$.

To minimize the retrieval range further, indexed descriptor file is used. We abstract data in a record into a bit string that is called descriptor of the bucket.

The collections of all bucket descriptors comprise the indexed descriptor file.

While querying records from our specification bases, we located the candidate buckets first by extensible hashing, then compare the query's descriptor D_q with those buckets descriptors D_b. If the bits equal to 1 in D_q has the same bit values in D_b, we look up this buckets for the records we want, otherwise, just discard the bucket without scanning each record in it.

3.5 The Browser

The browser provides a tool for user to browse specification bases that have been created. With the browser the user can either browse the definitions of concepts or operations in current base, or view concept dependency graphs. By using two stacks to keep browsing trace, objects can be viewed flexibly in hypertext style. Definitions can also be viewed in sequential order, or by attribute retrieval.

Concept dependency graph is a directed graph, in which a node is concept, and there is an edge from node X to node Y if and only if there is a production $X \rightarrow \ldots Y \ldots$, in this case we say X depends on Y. To display the dependency relations among concepts clearly, each cycle in a concept dependency graph is merged into a special node, then the graph becomes a simpler directed acyclic graph and is pretty-printed.

4 A Nontrivial Example

The example is the formal differentiation of elementary functions. Elementary functions are commonly used mathematic functions, such as polynomial functions, trigonometric functions, anti-trigonometric functions, exponential functions, logarithmic functions, and so on. In the example, the concept of elementary function will be acquired, and the operation on the function for differentiation evaluation will be defined.

We will first define the concept of elementary function. The concept names and samples we are given are listed in Table 1. Samples of a concept are separated by white space in the table. Some known concepts are used in the acquisition: `FuncName, Num, addOp, mulOp, id`, in which `Num` is the concept for number, `id` is the concept for identifier, grammars of `FuncName, addOp, mulOp` are given directly as shown below.

```
<FuncName>->exp|ln|sin|cos|tg|ctg|sec
    |csc|aSin|aCos|aTg|aCtg|aSec|aCsc
<addOp>->+|-
<mulOp>->*|/
```

The conversation with SAQ is shown in the appendix. The learned grammar for elementary function is as follows.

```
<elemFunc>-><term>
<elemFunc>-><elemFunc><addOp><term>
<term>-><factor>
<term>-><term><mulOp><factor>
<factor>-><primExpr>
<factor>->-<primExpr>
<factor>-><factor>^<primExpr>
<primExpr>-><Num>
<primExpr>-><Var>
<primExpr>->(<elemFunc>)
<primExpr>-><FuncName>(<elemFunc>)
<Var>-><id>
```

We will next define operation for differentiation. This can be started by giving a function name and its domain and range: `Diff: elemFunc -> elemFunc`, then we begin to define `Diff` by the support of SAQ. The dialog is as follows.

```
Now the conversation begins.
I will ask you questions, and you should
answer the questions. When you have answered
one question, press the 'Ok' button at the
bottom of this window, then I will know your
answer. Any question starts with prompt '>'
and perhaps has hint or instruction starting
with '-->' to instruct you to answer correctly.
For detailed information, see Help.
Good luck!

>if x0 is variable of <term>
   then Diff(x0)=?
tmDiff(x0)
>if x1 is variable of <elemFunc>
   x2 is variable of <addOp>
   x3 is variable of <term>
   then Diff(x1[]x2[]x3)=?
scat(Diff(x1),x2,tmDiff(x3))
>Definition of function Diff succeed.
The definition is:

dec Diff: elemFunc -> elemFunc;
var x0,x3 :term;
    x1 :elemFunc;
    x2 :addOp;
def Diff(x0)=tmDiff(x0);
   Diff(x1[]x2[]x3)=scat(Diff(x1),x2,tmDiff(x3));
```

In the definition, `scat` is the primitive function of concatenation, `[]` is the infix operator for concatenation. When unknown function name `tmDiff` appears, SAQ will ask user to give its domain and range and then add it to the new function set. Therefore we can continue to define it after defining present function. By this manner, all related functions will be defined.

Table 1 Samples of concepts

ConceptName	Samples
elemFunc	1 1+1
term	1 1*1
factor	1 1^1 -1
primExpr	1 x (1+1) exp(1+1) ln(1+1) sin(1+1) aSin(1+1)
Var	x

And the definitions of all functions are listed in the appendix. In the functions, two of them, i.e. `efNeg` and `tmNeg`, are defined directly.

5 Experiments and Conclusion

SAQ has been implemented on Sun SPARCstation. The development platform of SAQ is Solaris2.3 and XView toolkit. User interface is developed by the user interface development tool Devguide. Source code of SAQ consists of about 78,000 lines of C statements.

Many trivial and nontrivial experiments have been done successfully. Nontrivial experiments of SAQ include, besides the one of elementary function, acquisition of concepts for binary and decimal numbers and definition of relevant operations; acquisition of the grammar for a Lisp like small functional language and definition of an interpreter for the language. It shows that specifications can be acquired easily, and the constructed specifications are concise.

There are also some deficiencies in SAQ. For examples, when learning concept with larger definition body, searching speed is slow; and the evaluation efficiency of complex operation is also low. Future work include raising the efficiency of concept learning algorithm and the efficiency of operation application, so that SAQ could be more practical.

Acknowledgment

The authors would like to thank Wang Honghao, who prepared part of the example presented in this paper.

References

[1] Yunmei Dong. Collection of SAQ reports no.1-7. Technical Report ISCAS-LCS-95-09, Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences, August 1995.

[2] Yunmei Dong et. al. Collection of SAQ reports no.8-16. Technical Report ISCAS-LCS-96-1, Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences, Mar. 1996.

[3] Yunmei Dong. MLIRF method for specification acquisition and reuse. (in Chinese) Proc. of the 9th National Conf. of China Computer Federation, 21–27, May 1996.

[4] Yunmei Dong. An interactive learning algorithm for acquisition of concepts represented as CFL. J. Comput. Sci. & Technol., vol. 13, no. 1, 1–8, 1998.

[5] Yunmei Dong. Recursive functions defined on context-free languages (I). (in Chinese), Technical Report ISCAS-LCS-98-14, Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences, 1998.

[6] J. Earley. An efficient context-free parsing algorithm. Comm. ACM. 13, 94–102, 1970.

[7] E. M. Gold. Language identification in the limit. Information and Control, 10, 447–74, 1967.

[8] M. Harada. An automatic programming system SPACE with highly visualized and abstract program specification. IEICE Transactions on Information and Systems, vol. E78-D, no. 4, 403–19, April 1995.

[9] V. E. Kelly and U. Nonnenmann. Reducing the complexity of formal specification acquisition. M. R. Lowry and R. D. McCartney ed. Automating software design, 41–64, AAAI Press, 1991.

[10] A. Takura, Y. Ueda, T. Haizuka, and T. Ohta. Requirement specification acquisition of communications services. IEICE Transactions on Information and Systems, vol. E79-D, no. 12, 1716–25, Dec. 1996.

[11] Takashi Yokomori. Learning context-free language efficiently — a report on recent results in Japan. LNAI 397, 104–23, 1989.

Appendix. Complementary Materials For Formal Differentiation of Elementary Functions

The Conversation of Concept Acquisition

The ## is used to separate samples, and "..." are added by the authors to omit samples for saving space.

```
Do you want to accept the GUESS(from "1"):
<elemFunc>-><term>
and it generates new samples below:
1 ## 1*1
Yes.
Do you want to accept the GUESS(from "1+1"):
<elemFunc>-><elemFunc><addOp><elemFunc>
and it generates new samples below:
1 + 1+1 ## 1+1 - 1+1 ## 1+1 + 1 ## 1+1 + 1+1
No.
Do you want to accept the GUESS(from "1+1"):
<elemFunc>-><elemFunc><addOp><term>
and it generates new samples below:
1 + 1*1 ## 1+1 - 1*1 ## 1+1 + 1 ## 1+1 + 1*1
Yes.
Do you want to accept the GUESS(from "1"):
<term>-><elemFunc>
and it generates new samples below:
1 ## 1+1
No.
Do you want to accept the GUESS(from "1"):
<term>-><factor>
and it generates new samples below:
1 ## -1 ## 1^1
Yes.
Do you want to accept the GUESS(from "1*1"):
<term>-><term><mulOp><term>
and it generates new samples below:
1 * 1*1 ## 1*1 / 1*1 ## 1*1 * 1 ## 1*1 * 1*1
No.
Do you want to accept the GUESS(from "1*1"):
<term>-><term><mulOp><elemFunc>
and it generates new samples below:
1 * 1+1 ## 1*1 / 1+1 ## 1*1 * 1 ## 1*1 * 1+1
No.
Do you want to accept the GUESS(from "1*1"):
<term>-><term><mulOp><factor>
and it generates new samples below:
1 * 1^1 ## 1*1 / 1^1 ## 1*1 * 1 ## 1*1 * -1
 ## 1*1 * 1^1
Yes.
Do you want to accept the GUESS(from "1"):
<factor>-><term>
and it generates new samples below:
1 ## 1*1
No.
Do you want to accept the GUESS(from "1"):
<factor>-><primExpr>
and it generates new samples below:
1 ## x ## (1+1) ## exp(1+1) ## ln(1+1)
 ## sin(1+1) ## aSin(1+1)
Yes.
Do you want to accept the GUESS(from "-1"):
<factor>-><addOp><elemFunc>
and it generates new samples below:
- 1+1 ## + 1 ## + 1+1
No.
Do you want to accept the GUESS(from "-1"):
<factor>-><addOp><term>
and it generates new samples below:
- 1*1 ## + 1 ## + 1*1
No.
Do you want to accept the GUESS(from "-1"):
<factor>-><addOp><factor>
and it generates new samples below:
- 1^1 ## + 1 ## + -1 ## + 1^1
No.
Do you want to accept the GUESS(from "-1"):
<factor>-><addOp><primExpr>
and it generates new samples below:
- aSin(1+1)##+ 1##+ x##+ (1+1)##+ exp(1+1)
 ## + ln(1+1) ## + sin(1+1) ## + aSin(1+1)
No.
Do you want to accept the GUESS(from "-1"):
<factor>-><addOp><Num>
and it generates new samples below:
- 0 ## + 9 ## + 8 ## + 7 ## + 6 ## + 5 ## + 4
 ## + 3 ## + 2 ## + 1 ## + 0
No.
Do you want to accept the GUESS(from "-1"):
<factor>-><addOp>1
and it generates new samples below:
- 1 ## + 1
No.
Do you want to accept the GUESS(from "-1"):
<factor>->-<factor>
and it generates new samples below:
- 1 ## - -1 ## - 1^1
No.
Do you want to accept the GUESS(from "-1"):
<factor>->-<primExpr>
and it generates new samples below:
- 1 ## - x ## - (1+1) ## - exp(1+1)
 ## - ln(1+1) ## - sin(1+1) ## - aSin(1+1)
Yes.
Do you want to accept the GUESS(from "1^1"):
<factor>-><factor>^<factor>
and it generates new samples below:
1 ^ 1^1 ## -1 ^ 1 ## -1 ^ -1 ## -1 ^ 1^1
 ## 1^1 ^ 1 ## 1^1 ^ -1 ## 1^1 ^ 1^1
No.
Do you want to accept the GUESS(from "1^1"):
<factor>-><factor>^<term>
and it generates new samples below:
1 ^ 1*1 ## -1 ^ 1 ## -1 ^ 1*1 ## 1^1 ^ 1
 ## 1^1 ^ 1*1
No.
Do you want to accept the GUESS(from "1^1"):
<factor>-><factor>^<primExpr>
```

```
and it generates new samples below:
1 ^ aSin(1+1) ## -1 ^ 1 ## -1 ^ x ## -1 ^ (1+1)
 ## -1 ^ exp(1+1) ## -1 ^ ln(1+1)
 ## -1 ^ sin(1+1) ## -1 ^ aSin(1+1)
 ## 1^1 ^ 1 ## 1^1 ^ x ## 1^1 ^ (1+1)
 ##1^1^exp(1+1)##1^1^ln(1+1)##1^1^sin(1+1)
Yes.
Do you want to accept the GUESS(from "1"):
<primExpr>-><factor>
and it generates new samples below:
1 ## -1 ## 1^1
No.
Do you want to accept the GUESS(from "1"):
<primExpr>-><Num>
and it generates new samples below:
9 ## 8 ## 7 ## 6 ## 5 ## 4 ## 3 ## 2 ## 1 ## 0
Yes.
Do you want to accept the GUESS(from "x"):
<primExpr>-><Var>
and it generates new samples below:
x
Yes.
Do you want to accept the GUESS(from "(1+1)"):
<primExpr>->(<elemFunc>)
and it generates new samples below:
( 1 ) ## ( 1+1 )
Yes.
Do you want to accept the GUESS(from "exp(1+1)"):
<primExpr>-><FuncName><primExpr>
and it generates new samples below:
aCsc aSin(1+1) ## aSec 1 ## aSec x ## aSec (1+1)
...
No.
Do you want to accept the GUESS(from "exp(1+1)"):
<primExpr>-><id><primExpr>
and it generates new samples below:
_ aSin(1+1)
No.
Do you want to accept the GUESS(from "exp(1+1)"):
<primExpr>->exp<primExpr>
and it generates new samples below:
exp 1 ## exp x ## exp (1+1) ## exp exp(1+1)
 ## exp ln(1+1) ## exp sin(1+1) ## exp aSin(1+1)
No.
Do you want to accept the GUESS(from "exp(1+1)"):
<primExpr>-><FuncName>(<elemFunc>)
and it generates new samples below:
aCsc ( 1+1 ) ## aSec ( 1 ) ## aSec ( 1+1 )
...
Yes.
Do you want to accept the GUESS(from "x"):
<Var>-><id>
and it generates new samples below:
_A
Yes.
Then, we optimize the terms, and get result below:
(1) <elemFunc>-><term>
(2) <elemFunc>-><elemFunc><addOp><term>
(3) <term>-><factor>
(4) <term>-><term><mulOp><factor>
(5) <factor>-><primExpr>
(6) <factor>->-<primExpr>
(7) <factor>-><factor>^<primExpr>
(8) <primExpr>-><Num>
(9) <primExpr>-><Var>
(10) <primExpr>->(<elemFunc>)
(11) <primExpr>-><FuncName>(<elemFunc>)
(12) <Var>-><id>
```

Definition of Functions

```
dec Diff: elemFunc -> elemFunc;
var x0,x3 :term;
    x1 :elemFunc;
    x2 :addOp;
def Diff(x0)=tmDiff(x0);
   Diff(x1[]x2[]x3)=scat(Diff(x1),x2,tmDiff(x3));

dec efNeg : elemFunc -> elemFunc;
var t : term; e : elemFunc; a : addOp;
def efNeg(t) = tmNeg(t);
    efNeg(e[]a[]t) =
    if eq(a,"+") then scat(efNeg(e),"-",t)
    else scat(efNeg(e),"+",t);

dec ftDiff: factor -> elemFunc;
var x0,x1,x3 :primExpr;
    x2 :factor;
def ftDiff(x0)=peDiff(x0);
    ftDiff("-"[]x1)=efNeg(peDiff(x1));
    ftDiff(x2[]"^"[]x3)=scat(x2,"^",x3,"*(",
       tmDiff(scat(x3,"*","ln(",x2,")")),")");

dec peDiff: primExpr -> elemFunc;
var x0 :Num; x1 :Var;
    x2,x4 :elemFunc;
    x3 :FuncName;
def peDiff(x0)="0";
    peDiff(x1)=if eq(x1,"x") then "1" else "0";
    peDiff("("[]x2[]")")=Diff(x2);
    peDiff(x3[]"("[]x4[]")")=
if eq(x3,"exp") then
    scat(x3[]x4,"*",Diff(x4)) else
if eq(x3,"ln") then
    scat("(",Diff(x4),")","/","(",x4,")") else
if eq(x3,"sin") then
    scat("cos(",x4,")*",Diff(x4)) else
if eq(x3,"cos") then
    scat("-(","sin(",x4,")*",Diff(x4),")") else
if eq(x3,"tg") then
    scat("sec(",x4,")^2*",Diff(x4)) else
if eq(x3,"ctg") then
 scat("-(","csc(",x4,")^2*",Diff(x4),")")
else if eq(x3,"sec") then
 scat("tg(",x4,")*sec(",x4,")*",Diff(x4))
else if eq(x3,"csc") then
 scat("-(","ctg(",x4,")*csc(",x4,")*",
      Diff(x4),")")
else if eq(x3,"aSin") then
```

```
 scat("(",Diff(x4),")/","(1-",x4,"^2)^(1/2)")
else if eq(x3,"aCos") then
 scat("-((",Diff(x4),")/",
      "(1-",x4,"^2)^(1/2))")
else if eq(x3,"aTg") then
 scat("(",Diff(x4),")/(1+",x4,"^2)")
else if eq(x3,"aCtg") then
 scat("-((",x4,"^2*",Diff(x4),
       ")/(1+",x4,"^2))")
else if eq(x3,"aSec") then
 scat("(",Diff(x4),")/(",
       x4,"^2*((1-","1/",x4,"^2)^(1/2)))")
else scat("-",peDiff(scat("aSec(",x4,")")));

dec tmDiff: term -> elemFunc;
var x0,x3 :factor;
    x1 :term;
    x2 :mulOp;
def tmDiff(x0)=ftDiff(x0);
    tmDiff(x1[]x2[]x3)=if seq(x2,"*") then
 scat(tmDiff(x1),"*",x3,"+",x1,"*",ftDiff(x3))
else
   scat("((",tmDiff(x1),")*",x3,"-",
        x1,"*",ftDiff(x3),")/(",x3,"^2)");

dec     tmNeg : term -> elemFunc;
    var t : term;
def     tmNeg( t ) = "-"[]t;
```