EASE: An Embedded Algebraic Specification Environment

Jiahua Qian, Yong Zheng, Liqun Jin, & Jinhua Zhang
Dept. of Computer Science, Fudan University
Shanghai, China

ABSTRACT

An attempt is made to applying ideas about algebraic specification in the context of a programming language. EASE, an embedded algebraic specification environment, is developed based on a Pascal-oriented interactive programming environment FPE by allowing embedded algebraic specification (EAS) in the place of Pascal code. It provides a framework for the formal development of a program from a specification. This paper discusses the design of EAS, the use of EASE as a term rewriting system.

1. Introduction

Beginning with [7] and [6], work on an algebraic approach to program specification has focused on developing techniques of specifying programs (abstract data type in particular) and more recently on formalizing the notion of refinement as used in stepwise refinement (see e.g. [3,4]). The ultimate goal of this work is to provide a formal basis for program development, which would support a methodology for systematic development of programs from specifications by means of verified term rewriting. However, comparatively little work has been done on applying the results to programming, with a few exceptions such as CIP-L [1].

This paper represents our attempt at applying algebraic specifications in the context of a programming language. Currently, a Pascal-oriented interactive programming environment FPE is being extended by allowing embedded algebraic specification (EAS) in the place of Pascal code. The resulting system, called EASE (Embedded Algebraic Specification Environment), provides a framework for the formal development of a program from a specification. EASE can automatically generate code for any formally specified object by term rewriting, and can immediately execute and test this object.

We begin in section 2 by specifying the design of EAS, and in section 3 we describe the extension of interactive environment called Fudan Programming Environment (FPE). Section 4 represents the framework of EASE, and the verification and transformation of EAS specifications are specified in section 5 and section 6, respectively.

2. Embedded Algebraic Specification EAS

A recent trend in programming is the development of the abstract data type (ADT) or data abstraction. Several methods for specifying programs and ADT have been developed. We have chosen the algebraic axiomatic technique developed by Guttag [7] as our mathematical model of ADT specification. This approach is closely related to the work of Zilles [11], Goguen, Thater and Wagner [5], and Spitzn and Wegbreit [10]. Its formal basis stems from the heterogeneous algebras of Birkhoff and Lipson [2]. In the rest of the section, we will describe the design of EAS specification.

An algebraic specification of an abstract data type consists of two parts: a syntactic specification and a semantic specification. The syntactic specification defines the name, domains and ranges of the operations associated with the type. It provides the syntactic and type checking information that many programming languages require. The semantic specification containing a set of axioms in the form of equations defines the meaning of the operation by stating their relationship to each other.

The specifications can be generic or explicit. A generic specification is a schema that resides in a program unit and contains parameters (e.g. other sorts) that are instantiated when the specification is used in an actual program. An explicit specification is a refinement of a generic specification that substitutes actual arguments for the specification parameters.

EASE allows user reuse of all predefined EAS specifications, but users seldom use them all in a given program. Hence, the concept of an ADT-unit is
introduced to address this problem. The BAS specifications are split into related groups, i.e. ADT-units. Users can use only the ADT-units his program needs.

An ADT-unit is a collection of abstract data type (ADT) definitions and a set of associated operations. In short, an ADT-unit is a library of reusable BAS specifications.

The general structure of an ADT-unit is:

```
ADT-UNIT <unitname> IS
  USE SORT <unitlist>;
  (ADT definition,);
  OPERATOR ...
  AXIOMS ...
END
```

and the syntax of an ADT definition in an ADT-unit has the following structure:

```
SORT <sortname> IS
  --- object created
  --- sort parameters
  --- sorts based on
  CONSTRUCTOR ...
  OPERATOR ...
  AXIOMS ...
END
```

It is noticed that the structures of operators and axioms in an ADT-unit are the same as that in an ADT definition, but with some significant differences. In an ADT-unit, each operator is the operation on those sorts defined in that ADT-unit, but each operator in an ADT definition is the operation only on that defined sort.

There is a restriction about axioms: All the axioms satisfy the condition that the outermost function in the left hand side (LHS) is a operator and the remaining functions in the LHS are constructors. In other systems, there are more restrictions about axioms: (1)Linear restriction of algebraic axioms: (An algebraic specification with this restriction is called linear algebraic specification, otherwise it is called non-linear algebraic specification). (2)All the constructors in the LHS belong to the constructor set of current sort. We release these two restrictions in our EAS specification language.

For example, the algebraic axioms of AND in Fig.1 are non-linear. So it can not be accepted by a transformation system which is based on linear algebraic specification, but it can be accepted by a transformation system which is based on non-linear algebraic specifications.

```
and(x,x) = x;
and(x,false) = false;
and(false,x) = false;
```

**Fig. 1**

Example: Fig. 2 gives an example of the EAS specification of symboltable. The constructors of the symboltable are INIT which creates a new symboltable, ENTER which enters a block, and ADDID which adds a new item into the symboltable. The operators are LEAVE which leaves a block, ISIN which checks whether an identifier belongs to the symboltable, and RETRIEVE which fetches the attributelist of an identifier from the symboltable. There are two parameters: IDENT and ATTR. Retrieve_error is a error value for exception handling. We use a prefix X to identify an error value.

```
sort symboltable is
  need sort ident, attr;
  constructor
    init;
    enter : symboltable;
    addid : symboltable, ident, attr;
  operator
    leave : symboltable -> symboltable;
    leave(init) = init;
    leave(enter(s)) = s;
    leave(addid(s, id, attrlist)) = leave(s);
  axiom is
    leave(init) = init;
    leave(enter(s)) = s;
    leave(addid(s, id, attrlist)) = leave(s);
  operator
    isin : symboltable, ident -> boolean;
    isin(init, id) = false;
    isin(enter(s), id) = isin(s, id);
    isin(addid(s, id1, attrlist), id2) = if eq(id1, id2)
      then true
      else isin(s, id2);
  axiom is
    isin(init, id) = false;
    isin(enter(s), id) = isin(s, id);
    isin(addid(s, id1, attrlist), id2) = if eq(id1, id2)
      then attrlist
      else retrieve(s, id2);
end
```

**Fig. 2**

3. Extension to the Interactive Environment FPE

3.1. Review of FPE

FPE is a Pascal-oriented programming environment developed at Fudan University, which operates on DEC Micro Vax/GPX. FPE offers the user a uniform and friendly interface for program editing, compiling, executing, testing and debugging. FPE mainly consists of the following
components and features.

1. A language based editor with both syntax-directed and text-oriented modes, which offers menu selection and text input.

2. An incremental semantic analyzer based-on-the nonlocal attributed grammar, which allows the program to be compiled while it is being edited and immediately feeds back the semantic errors.

3. An executing supervisor which controls the program execution and supports program debugging based on source language.

4. An unparser which deparses the syntax tree of a program to its formatted text.

Readers can consult [9] for details of FPE.

3.2. Extension of Pascal

For applying ideas about EAS specifications in the context of the programming language Pascal, we extend Pascal by allowing EAS specifications in the place of Pascal code.

1. An explicit specification is created by a refinement of a generic EAS specification via the USE clause after the corresponding ADT-unit is loaded. Syntactically, loading the ADT-units is expressed by: USE <ADT-unit-list> and a refinement of a generic EAS specification is:

   SORT <identifier> IS
   USE <sortname>
   [ BIND <parameterlist> ]
   END

where, ADT-unit-list is the name list of ADT-units to be loaded, IDENTIFIER is the refined sort name, SORTNAME is the name of the generic specification to be refined, and PARAMETERLIST is the list of actual functions and sorts that are substituted for the parameters in the generic specification definition. All of the defined operations in the original generic sort are now redefined in the context of this new refined sort, however, the actual mapping of original name to the new one is handled automatically and of no concern to the programmer.

3.3. Modification to the Toolset in FPE

The editor and prettyprinter of FPE have been modified to accept source programs and ADT-units which contain EAS specifications. The Pascal grammar driving FPE has been enhanced with EAS syntax.

At each stage during editing, the immediate feedback to user is made when the input is received. Every time user modifies an EAS specification which he or she is editing, the incremental semantic analyzer and verifier are invoked automatically to manipulate the internal representation of the EAS specification, and check the consistency and completeness incrementally.

The incremental semantic analyzer of FPE has been extended to treat an explicit sort specification as if it is a Pascal type declaration, and to allow a programmer to use an explicit sort any place a type is found. Meanwhile, the constructors and operators associated with each sort may be used as the functions or procedures in Pascal.

Moreover, the extended semantic analyzer also supports incremental type checking and parameter matching at the level of axiom definition of EAS specification, which allows partial semantic constraints to be checked while EAS specification is being edited. Once an error has been detected, an error message is made available to user, and error handling is on-line.

The executing supervisor of FPE is also extended by allowing the programs which embody EAS specifications to be executed immediately, and supporting program debugging based on EAS specifications. Therefore, it can be used as a rapid prototyping tool to test the EAS specifications built in the system.

4. Overview of EASE

EASE makes an attempt to apply ideas about abstract data type specifications in the context of programming language Pascal. It allows embedded algebraic specification (EAS) in the place of Pascal code. An outstanding feature of EAS is that not only can it be executed immediately, but also be efficiently compiled into final source code. During the transformation, EASE permits software reuse at the level of EAS specification. Concretely, EASE is also based on attribute grammar. It represents an EAS specification as an attributed abstract syntax tree. The specifications are modified by tree operations such as pruning, grafting, and deriving. After each modification to an attributed tree, some of the attributes require new values, incremental semantic analysis and verification are performed by updating attribute values in response to modifications. Up to date, EASE includes following components (Fig.3):

(1) Modified FPE, an extension to the interactive programming environment FPE.
(2) Incremental verifier, verify the set of axioms in EAS specification.
(3) Transformer, transforms EAS specifications into the internal representation --- attributed abstract syntax tree.
(4) I&A, the instantiator and assembler, instantiate the attributed abstract syntax tree template, link the instantiated tree templates to the program internal representation, and create standard Pascal source program.
Program IRep (Internal REPresentation), Unit IRep : attributed semantic tree

Fig. 3

1. Introduction

IRep-M, is a tool that manages all kinds of software libraries in EASE.

2. User Interface

User interface, includes multi-window management, file system, on-line help, and other auxiliary facilities.

In the rest of the paper, a brief description about the verification and transformation of the EAS specification will be specified.

5. Verification of BAS Specification

Proving completeness is important in algebraic specification. Unfortunately, there are no general algorithms to decide the completeness[3]. Thus one has to define sufficient conditions that work in practice. Many criteria have been proposed.

Huet and Hullot[8] proposed the sufficient completeness criterion for linear algebraic specification. In the following, we introduce an approach to check the sufficient completeness for non-linear algebraic specification.

A data pattern of an ADT is a well-formed item which is consisted of constructors and variables. A data item is a data pattern which does not contain variables. The domain of a n-tuple of data pattern dp = <dp1,...,dpn> is a set of n-tuple of data item d = <d1,...,dn> where d is matched with dp. We use <dp1,...,dpn> to denote the domain of this n-tuple. The difference of two n-tuples of data patterns dp and dq is a set of n-tuple of data item d where d is matched with dp and is not matched with dq.

Using the former concepts, we say a set of n-tuple of data pattern A overlay a set of n-tuple of data pattern B if and only if (1) B is an empty set; or 2) For any a in A and b in B (A-{a}) U (a-b) overlays (B-{b}) U (b-a). Then, we define the set of the left hand sides of axioms \{f(t1,...,tn),...,f(tm1,...,tmn)\} is sufficient-completeness if and only if the set \{<t1,...,tn>,...,<tm1,...,tmn>\} overlays \{<x1,...,xn>\}, where xi (i=1,...,n) are variables, and any of two variables are not equal. The checking steps is:

1) first let A = \{<t1,...,tn>,...,<tm1,...,tmn>\} and B = \{<x1,...,xn>\},
2) then repeat the second part of the definition of overlay to reduce the set B until B is empty or unchanged. If the B is empty, A is sufficient completeness, otherwise A is not sufficient completeness.

For example, the following axiom set

\[
\text{operator}\quad \text{less} : \text{nat} \rightarrow \text{boolean};
\]
\[
\text{axioms}\quad \text{less}(0, \text{succ}(x)) = \text{true}; \quad \text{less}(\text{succ}(x), 0) = \text{false};
\]
\[
\text{less}(x, x) = \text{false};
\]

is not sufficient completeness as the following steps show:

0) At first, A = \{<0, \text{succ}(x)>, <\text{succ}(x), 0>, <x, x>\}, B = <x, x>;
1) Let a = <0, \text{succ}(x)> and b = <x, x>, then a - b = [], b - a = <\text{succ}(x), 0>, (0, 0); so A = \{<\text{succ}(x), 0>, <x, x>\}, B = <\text{succ}(x), 0>, (0, 0), <\text{succ}(x), \text{succ}(y)>;
2) Let a = <\text{succ}(x), 0> and b = <\text{succ}(x), 0>, then a - b = [], so A = \{<x, x>\}, B = <0, 0>, <\text{succ}(x), \text{succ}(y)>;
3) Let a = <x, x> and b = <0, 0>,
then $a = \text{succ}(x)$, $\text{succ}(x)$, $b$.

so $A = \{\text{succ}(x), \text{succ}(x)\}$, $B = \{\text{succ}(x), \text{succ}(y)\}$.

At this time, we can not go forward to reduce $B$ to empty. So, $A$ dose not overlay $B$. It means that the axiom set of less is not sufficient completeness.

According to this approach, we have developed a verifier in EASE. The verifier is expressed by a set of semantic functions, and the completeness is checked by the incremental attribute evaluation. Algorithms has been given for checking the overlayship and computing the difference between two data pattern tuple sets. The verifier also immediately feeds back the status of current equation set (whether it is complete or not, why it is incomplete, and some more axioms are suggested to approach completeness). For example, after checking the former axiom set of less, the verifier will report the following information:

The following axiom forms should be added to approach completeness:

$\text{less}(\text{succ}(x),\text{succ}(y)) = \ldots$;

Hence, the verifier not only checks the axiom completeness, but also directs the user to develop a sufficiently complete algebraic specification.

6. Transformation of EAS Specification

In the EASE system, an EAS specification, which is semantically correct and whose axioms are sufficiently complete, is transformed to a Pascal-based implementation by the transformer, when it is referred by an EPascal program. The Pascal-based implementation is consisted of type definitions and function definitions, and is linked to the referring program. Users can use the debugging tool of EASE to test the program or use the standard tool of EASE to generate a standard Pascal program as the result of the software development.

The internal representation of the EAS specification and the EPascal program in EASE are abstract semantic trees, so the objects of the transformer are tree-structure. The transformer can be regarded as a series of operations on the trees. All tree operators are formed a software package and are shared by all components of EASE.

The main transformation process is: 1) deriving the data type definition from the constructor set, 2) deriving the function definitions of the constructors from the signature of the constructors, 3) deriving the function definitions of the operators from the signature of the operators and algebraic axioms. We use a set of transformation rules to implement the transformer. For more detail, see [12]. In the following, we use the symboltable example to illustrate the transformation process.

6.1 Derive the data type definition

The data of an ADT can be expressed by the set of data item. A data item can be expressed by a tree-structure data[12]. Pointer type and variant record type of Pascal can be used to implement the tree-structure data. In the transformer, data transformation rule transforms the set of constructors to a set of pointer type and variant record type definitions.

The type definitions of symboltable are in Fig.4.

6.2 Derive the function definition of constructors

When the data of an ADT is defined by using pointer type and variant record type, each constructor is transformed to a function definition which generates a new variant record, assigns the arguments of the constructor to the corresponding field of the new variant record and return the new variant record.

Some of the function definitions of the symboltable are in Fig.5.

6.3 Derive the function definition of operators

The semantics of the operator is defined by a set of algebraic axioms. It can be seen as a term-rewriting system. We design a transformation algorithm for non-linear algebraic specification[12]. It use non-linear pattern matching approach. It transforms the set of the algebraic axioms of an operator to a new linear semantically-equivalent algebraic axiom in which there is only one function name in the left hand side. This kind of axiom can be immediately implemented as a Pascal function by using the condition structure in Pascal to replace the if-then-else structure in the axiom.
FUNCTION init() : symboltable;
VAR result : symboltable;
BEGIN
new(result);
result^.symboltable_tag := symboltable_init;
init := result;
END;

FUNCTION addid(addidl : symboltable, addid2 : ident, addid3 : attr) : symboltable;
VAR result : symboltable;
BEGIN
new(result);
result^.symboltable_tag := symboltable_addid;
result^.addidl := addidl;
result^.addid2 := addid2;
result^.addid3 := addid3;
addid := result;
END;

Fig. 5

There are many recursive structure in the axioms. We use recursive optimization rules to remove some recursive structures[12]. The definition of function isin is shown in Fig.6.

FUNCTION isin(isin1 : symboltable, isin2 : ident) : boolean;
VAR result : boolean;
BEGIN
IF isin1^.symboltable_tag == symboltable_init
THEN result := false
ELSE IF isin1^.symboltable_tag == symboltable_enter
THEN result := isin(isin1^.enter1,isin2)
ELSE if ident_eq(isin1^.addid1,isin1^.addid2,isin2)
THEN result := true
ELSE result := isin(isin1^.addid1,isin2);
isin := result;
END;

Fig. 6

7.Conclusion

We have presented our attempt to apply the ideas about algebraic specifications in the context of a programming language. A system called EASE, which allows embedded algebraic specification (EAS), is also described. In general, EASE can automatically generate code for any formally specified object by term rewriting, and can immediately execute and test this object. Therefore, it can be used as a tool for support program development, program prototyping, and specification reuse. We believe our approach can be refined into a production quality system for support the formal development of programs from their specifications. We shall be exploring refinement of our approach to accomplish this end.

8.References