Annales Univ. Sci. Budapest., Sect. Comp. **9** (1988) 33-50

ALGEBRAIC SPECIFICATION OF AN ABSTRACT HIGH-LEVEL DEBUGGER

NGUYEN HUU CHIEN and LÁSZLÓ VARGA

Abstract. In this paper the algebraic specification technique is applied for describing an abstract debugger. Abstract high-level debugger is regarded as a parameterized data type. A high-level debugging mechanism is based on GPE (Generalized Path Expression) [1]. The specification of the debugger is composed of specification of its data types and a set of operations on them. One of aims of the specification is to provide a mathematical system for the debugger, which could be analysed by formal methods.

1. Introduction

Formal specification in programming is a current trend in development of reliable software systems. Some methods of specification are extensively considered, one of them the algebraic technique emerges as a popular and interesting one. An application of the algebraic specification in development of programming systems is described for example in [4]. In our paper we use the algebraic method for specifying an abstract high-level debugger.

In [1] the authors used GPE as a debugging mechanism to monitor the dynamic behavior of a program. In this paper we specify this mechanism with the algebraic method. The debugger is regarded as parameterized data types. The specification of the debugger is composed of specification of its data types and most

common operations. The data types are GPE's and actual behavior of programs. The most common operations of the debugger are ones for manipulating GPE's and behaviors.

2. The specification of the debugger

2.1. The specification of the data types

Let us denote

$$\begin{aligned} arithop &= \{+, -, *, /, **\} \\ rel &= \{=, <, >, \leq, \geq\} \\ logic &= \{\vee, \wedge, \rightarrow\}. \end{aligned}$$

We assume that the data types *integer, Boolean* and *arithmetic expression* are predefined with the usual oparations on the sets *arithop, rel* and *logic*. Then, we construct the data types *set* and GPE, the main structure in the debugger. Let "assertion" be a set of assertions which are defined on the same set of elements.

2.1.1. Set data type

Let set be a set type with the usual operations *insert, delete*, and *has*. In addition, we define the operations *union, select, find* and *decartess*.

First, we introduce *pair* as a data type, which returns a pair of given elements.

pair type:

$$\begin{aligned} set - pair &: element \times element \rightarrow pair \\ left &: pair \rightarrow element \\ right &: pair \rightarrow element \end{aligned}$$

semantics:

$$\begin{aligned} left(set - pair(e, e')) &= e \\ right(set - pair(e, e')) &= e' \end{aligned}$$

REMARK. For simplicity throught the rest of the paper we denote $set - pair(e, e')$ by (e, e').
Next, we define the additional operations for the data type *set*.

$$\begin{aligned} select &: set \times assertion \to set \\ find &: set \times assertion \to element \\ union &: set \times set \to set \\ decartess &: set \times set \to set \end{aligned}$$

semantics:

$$union(empty, empty) = empty$$
$$union(insert(s, e), insert(s', e), insert(s', e')) =$$
$$= insert(insert(union(s, s'), e), e')$$

$$select(empty, a) = empty$$
$$select(insert(s, e), a) =$$
$$= \underline{\text{if}}\ a(e)\ \underline{\text{then}}\ insert(select(s, a), e)$$
$$\underline{\text{else}}\ select(s, a)$$

$$find(empty, a) = \text{UNDEFINED}$$
$$find(insert(s, e), a) = \underline{\text{if}}\ a(e)\underline{\text{then}}\ e$$
$$\underline{\text{else}}\ find(s, a)$$

$$decartess(empty, empty) = empty$$
$$decartess(insert(s, e), insert(s', e')) =$$
$$= union(union(union(decartess(s, s'),$$
$$decartess(insert(empty, e), s')),$$
$$decartess(s, insert(empty, e')),$$
$$insert(empty, set - pair(e, e')))$$

Note that the operation *union* computes the union of sets, the operation *select* extracts elements of a set which satisfy a given

assertion, the operation *find* look for an element which satisfies a given assertion, and the operation *decartess* determines the Decartess product of sets.

2.1.2. GPE data type

In the following we define GPE's as data type. GPE is used for specifying the expected behavior of programs, and the operations of the debugger are excuted to manipulate GPE.

Let us denote

$$T = \{act, term\}$$
$$biop = \{;, \oplus\}$$

and let P be a set of event names.

a) type *count*

$$set - count : P \times T \rightarrow count$$
$$event - name : count \rightarrow P$$
$$event - type : count \rightarrow T$$

semantics:

$$event - name(set - count(p,t)) = p$$
$$event - type(set - count(p,t)) = t$$

b) type *predicate*(count,arithmetic-expr,Boolean)

$$set - rel1 : count \times rel \times count \rightarrow predicate$$
$$set - rel2 : arithmetic - expr \times rel \times arithmetic - expr \rightarrow predicate$$
$$set - pr1 : predicate \times logic \times predicate \rightarrow predicate$$
$$set - pr2 : \{\neg\} \times predicate \rightarrow predicate$$
$$is - rel1 : predicate \rightarrow Boolean$$
$$is - rel2 : predicate \rightarrow Boolean$$
$$is - pr1 : predicate \rightarrow Boolean$$
$$is - pr2 : predicate \rightarrow Boolean$$

semantics:

$$is-rel1(q) = \underline{\text{if}}\ q = set-rel1(c1, r, c2)\ \underline{\text{then}}\ true\ \underline{\text{else}}\ false$$
$$is-rel2(q) = \underline{\text{if}}\ q = set-rel2(a1, r, a2)\ \underline{\text{then}}\ true\ \underline{\text{else}}\ false$$
$$is-pr1(q) = \underline{\text{if}}\ q = set-pr1(q1, op, q2)\ \underline{\text{then}}\ true\ \underline{\text{else}}\ false$$
$$is-pr2(q) = \underline{\text{if}}\ q = set-pr2(\ l, q1)\ \underline{\text{then}}\ true\ \underline{\text{else}}\ false$$

c) type *operand*(predicate)

$$set-operand : P \times predicate \rightarrow operand$$
$$event : operand \rightarrow P$$
$$pre : operand \rightarrow predicate$$

semantics:

$$event(set-operand(p, q)) = p$$
$$pre(set-operand(p, q)) = q$$

d) type GPE(operand,integer,Boolean,set)

$$set-GPE0 : operand \rightarrow GPE$$
$$set-GPE1 : GPE \times biop \times GPE \rightarrow GPE$$
$$set-GPE2 : GPE \times \{*\} \rightarrow GPE$$
$$is-operand : GPE \rightarrow Boolean$$
$$is-GPE1 : GPE \rightarrow Boolean$$
$$is-GPE2 : GPE \rightarrow Boolean$$
$$operand : GPE \rightarrow operand$$
$$number : GPE \rightarrow integer$$
$$expr : GPE \rightarrow set$$

semantics:

$$
\begin{aligned}
is-operand(E) = {} & \underline{\text{if}}\ E = set-GPE0(o)\ \underline{\text{then}}\ true \\
& \underline{\text{else}}\ false \\
is-GPE1(E) = {} & \underline{\text{if}}\ E = set-GPE1(E1, op, E2)\ \underline{\text{then}}\ true \\
& \underline{\text{else}}\ false \\
is-GPE2(E) = {} & \underline{\text{if}}\ E = set-GPE2(E1, *)\ \underline{\text{then}}\ true \\
& \underline{\text{else}}\ false \\
operand(E) = {} & \underline{\text{if}}\ is-operand(E) \& E = set-GPE0(o)\ \underline{\text{then}}\ o \\
& \underline{\text{else}}\ \text{UNDEFINED}
\end{aligned}
$$

$$
\begin{aligned}
number(E) = {} & \underline{\text{if}}\ is-operand(E)\ \underline{\text{then}}\ 1 \\
& \underline{\text{else}} \\
& \underline{\text{if}} E = set-GPE1(E1, op, E2)\ \underline{\text{then}} \\
& number(E1) + number(E2) \\
& \underline{\text{else}} \\
& \underline{\text{if}}\ E = set-GPE2(E1, *)\ \underline{\text{then}} \\
& number(E1)
\end{aligned}
$$

$$
\begin{aligned}
expr(E) = {} & \underline{\text{if}}\ E = set_G PE0(o)\ \underline{\text{then}}\ insert(empty, E) \\
& \underline{\text{else}} \\
& \underline{\text{if}}\ E = set_G PE1(E1, op, E2)\ \underline{\text{then}} \\
& union(expr(E1), expr(E2)) \\
& \underline{\text{else}} \\
& \underline{\text{if}}\ E = set-GPE2(E1, *)\ \underline{\text{then}} \\
& expr(E1)
\end{aligned}
$$

Note that the operation *number* gives number of operands of a GPE, and *expr* defines set of operands of a GPE.

2.2. The specification of the operations

The operations of the abstract debugger are built from operations defined on GPE's with other ones related. We dealt only with the most common operations. These are as follows:

index - transforms a GPE E into a so-called indexed GPE in the following way: It supplies the operands of E with integers $i, i+1, \ldots$ continously, in such a manner so that any operand should receive indexes at different occurrences.

first - returns a set of event names which can occur at beginning.

last - gives a set of event names which can occur at end.

relation - returns a set of pairs of event names which can occur one after another.

next - keeps event names which are allowed to occur at next time.

valid - checks if an actual behavior is valid with respect to any GPE.

In the following we specify these operations.

Let us denote by indGPE a type GPE with the parameters integer,integer,Boolean and set, that is

$$indGPE = GPE(operand\ integer, integer, Boolean, set).$$

This GPE type is called indexed GPE, and its elements are called indexed expressions and are denoted by indE.

a) The operation *index*

$$index(GPE, integer, indGPE)$$
$$index : GPE\ integer \rightarrow indGPE$$

semantics:

$$index(set - GPE0(o), i) = set - GPE0((o, 1))$$
$$index(set - GPE1(E1, op, E2), 1) =$$
$$= set - GPE1(index(E1, 1), op,$$
$$index(E2, 1 + number(E1)))$$
$$index(set - GPE2(E, *)) = set - GPE2(index(E, 1), *)$$

REMARK. If $i = 1$, then we write $index(E, 1)$ as $index(E)$.

b) The first operation

$$first(indGPE, set)$$
$$first : indGPE \rightarrow set$$

semantics:

$$first(set - GPE0(o, 1)) = insert(empty, set - GPE0(o, 1))$$
$$first(set - GPE1(indE1, , , indE2) = first(indE1)$$
$$first(set - GPE1(set - GPE2(indE1, *), , , indE2)) =$$
$$= union(first(indE1), first(indE2))$$
$$first(set - GPE1(indE1, \oplus, indE2)) =$$
$$= union(first(indE1), first(indE2))$$
$$first(set - GPE2(indE, *)) = first(indE)$$

c) The last operation

$$last(indGPE, set)$$
$$last : indGPE \rightarrow set$$

semantics:

$$last(set - GPE0(o, 1)) = insert(empty, set - GPE0(o, 1))$$
$$last(set - GPE1(indE1, , , indE2) = last(indE2)$$
$$last(set - GPE1(indE1, , , set - GPE2(indE2, *))) =$$
$$= union(last(indE1), last(indE2))$$
$$last(set - GPE1(indE1, \oplus, indE2)) =$$
$$= union(last(indE1), last(indE2))$$
$$last(set - GPE2(indE, *)) = last(indE).$$

d) The relation operation.

$$relation(indGPE, set)$$
$$relation : indGPE \rightarrow set$$

semantics:

$$relation(set - GPE0(o, 1) = empty$$
$$relation(set - GPE1(indE1, , , indE2)) =$$
$$= union(union(relation(indE1), relation(indE2)),$$
$$decartess(last(indE1), first(indE2)))$$
$$relation(set - GPE1(indE1, \oplus, indE2)) =$$
$$= union(relation(indE1), relation(indE2))$$
$$relation(set - GPE2(indE, *)) =$$
$$= union(relation(indE),$$
$$decartess(last(indE), first(indE))).$$

e) The data type *behavior* and the operations *valid* and *next*.

$$behavior\ type(P, GPE, Boolean, set, predicate, integer, count)$$

$$new :\rightarrow behavior$$
$$set - behavior : behavior \times P \rightarrow behavior$$
$$f_a : behavior \times P \rightarrow integer$$
$$f_t : behavior \times P \rightarrow integer$$
$$f : behavior \times count \rightarrow integer$$
$$I : behavior \times predicate \rightarrow Boolean$$
$$valid : GPE \times behavior \rightarrow Boolean$$
$$next : GPE \times bahavior \rightarrow set$$

semantics:

$$f_a(new,p) = 0$$
$$f_a(set\text{-}behavior(B,p),p') =$$
$$= \underline{\text{if}}\ p = p'\ \underline{\text{then}}\ f_a(B,p) + 1$$
$$\underline{\text{else}} f_a(B,p')$$

$$f_t(new,p) = 0$$
$$f_t(set\text{-}behavior(B,p),p') =$$
$$= \underline{\text{if}}\ p = p'\ \underline{\text{then}}\ f_t(B,p) + 1$$
$$\underline{\text{else}} f_t(B,p')$$

$$f(B,c) = \underline{\text{if}}\ event - type(c) = act\ \underline{\text{then}}\ f_a(B, event - name(c))$$
$$\underline{\text{else}}\ f_t(B, event - name(c))$$

$$I(B, set - rel1(c1,r,c2)) = f(B,c1)rf(B,c2)$$
$$I(B, set - rel2(a1,r,a2)) = a1ra2$$
$$I(B, set - pr1(q1,op,q2)) = I(B,q1)opI(B,q2)$$
$$I(B, set - pr2(\neg,q)) = \neg I(B,q)$$

$$valid(E,new) = true$$
$$next(E,new) = first(index(E))$$

$$valid(E, set - behavior(B,p)) =$$
$$= \underline{\text{if}}\ valid(E,B)\&$$
$$find(next(E,B), I(B, pre(operand(e)))\&$$
$$event(operand(e)) = p) = e$$
$$\underline{\text{then}}\ true$$
$$\underline{\text{else}}\ false$$

$$
\begin{aligned}
&next(E, set-behavior(B,p)) = \\
&\quad = \underline{\text{if}}\ valid(E, set-behavior(B,p)) \\
&\qquad \underline{\text{then}} \\
&\qquad select(expr(index(E)), \\
&\qquad\quad find(next(E,B), I(B, pre(operand(e))) \& \\
&\qquad\qquad event(operand(e)) = p) = e \\
&\qquad\quad \& has(relation(index(E)), (e,e'))) \\
&\qquad \underline{\text{else}}\ \text{UNDEFINED}
\end{aligned}
$$

REMARK:

1. The elements e and e' in the above formulas denote the elements of *next* and *expr*, respectively.

2. The operations *valid* and *next* can be interpreted as follows:

$valid(E, set-behavior(B,p))$ becomes true iff $valid(E,B)$ becomes true and there is an element e of $next(E,B)$ the name of which is P and the associated predicate becomes true.

$next(E, set-behavior(B,p))$ is defined iff $valid(E, set-behavior(B,p))$ is true, and then it consists of the elements e' of $expr(index(E))$ for which there is an event e of $next(E,B)$ the name of which is p and the associated predicate is true, and $(e,e') \in relation(index(E))$.

3. An analysis of the debugger

Now we have a formal system which could be analysed by mathematical methods. In the following we prove some properties of the operations of the debugger.

First we give three lemmas without proof.

Lemma 1. *For the predicates* $q1$ *and* $q2$

$$q1 = q2 \rightarrow \forall B(I(B,q1) = I(B,q2))$$

REMARK. The equality $q1 = q2$ is defined as follows:

$$
\begin{aligned}
q1 = q2 \leftrightarrow (&q1 = set - rel1(c1, r1, c2)\\
&\&q2 = set - rel1(c3, r2, c4)\\
&\&c1 = c3 \& c2 = c4 \& r1 = r2)\\
(&q1 = set - rel2(a1, r1, a2)\\
&\&q2 = set - rel2(a3, r2, a4)\\
&\&a1 = a3 \& a2 = a4 \& r1 = r2\\
(&q1 = set - pr1(q3, op1, q4)\\
&\&q2 = set - pr1(q5, op2, q6)\\
&\&q3 = q5 \& q4 = q6 \& op1 = op2)\\
(&q1 = set - pr2(|, q3) \& q2 = set - pr2(|, q4)\\
&\&q3 = q4)
\end{aligned}
$$

Lemma 2. *In case of type set*

$$
\begin{aligned}
find(h, a) = e \leftrightarrow &has(h, e) \& a(e)\\
&\leftrightarrow has(select(h, a), e).
\end{aligned}
$$

Lemma 3. *For any E and B*

$$has(next(E, B), e) \rightarrow has(expr(index(E)), e).$$

Theorem 1. *Let E and E' be two GPE's.*
If

$$
\begin{aligned}
(i)\ &has(first(index(E)), e) \rightarrow\\
&find(first(index(E')),\\
&\quad pre(operand(e')) = pre(operand(e)) \&\\
&\quad event(operand(e')) = event(operand(e))) = e'
\end{aligned}
$$

(ii) $has(relation(index(E)), (e1, e2)) \rightarrow$
$(has(expr(index(E')), e3)\&$
$pre(operand(e3)) = pre(operand(e1))\&$
$event(operand(e3)) = event(operand(e1)) \rightarrow$
$(find(expr(index(E')),$
$pre(operand(e4)) = pre(operand(e2))\&$
$event(operand(e4)) = event(operand(e2))) = e$
$\&has(relation(index(E')), (e3, e4))))$

then

(iii) for any B behavior
$has(next(E, B), e) \rightarrow$
$find(next(E', B),$
$pre(operand(e')) = pre(operand(e))\&$
$event(operand(e')) = event(operand(e))) = e'$

Proof. We prove this by induction on the structure of B.

1. Since

$$next(E, new) = first(index(E))$$
$$next(E', new) = first(index(E')),$$

so by (i) the lemma holds.

2. Assuming that the lemma holds for B, we show that it holds for $B' = set - behavior(B, p)$, too.

$$\begin{aligned} &has(next(E, B'), e) \\ &= has(select(expr(index(E)), \\ &\qquad find(next(E, B), \\ &\qquad\quad I(B, pre(operand(e'))))\& \\ &\qquad\quad event(operand(e')) = p) = e' \\ &\qquad \&has(relation(index(E)), (e', e''))), e) \end{aligned}$$

$$\rightarrow find(next(E,B), I(B, pre(operand(e'))) \&$$
$$event(operand(e')) = p) = e$$
$$\&has(relation(index(E)), (e'e)) \qquad (Lemma\ 2)$$

$$\rightarrow has(next(E,B), e') \& I(B, pre(operand(e'))) \&$$
$$event(operand(e')) = p \&$$
$$has(relation(index(E)), (e', e)) \qquad (Lemma\ 2)$$

$$\rightarrow find(next(E', B), pre(operand(e'')) = pre(operand(e'))$$
$$\&event(operand(e'') = event(operand(e')) = e''$$
$$\& I(B, pre(operand(e'))) \&$$
$$event(operand(e')) = p \&$$
$$has(relation(index(E)), (e', e)) \qquad (induction)$$

$$\rightarrow has(next(E', B), e'') \& pre(operand(e'')) =$$
$$pre(operand(e')) \&$$
$$event(operand(e'')) = event(operand(e')) \&$$
$$I(B, pre(operand(e'))) \& event(operand(e')) = p \&$$
$$has(relation(index(E)), (e', e)) \qquad (Lemma\ 2)$$

$$\rightarrow find(next(E', B), I(B, pre(operand(e''))) \&$$
$$event(operand(e'')) = p) = e''$$
$$\&has(relation(index(E)), (e', e)) \&$$
$$has(expr(index(E')), e'') \&$$
$$pre(operand(e'')) = pre(operand(e')) \&$$
$$event(operand(e'')) = event(operand(e'))$$
$$(Lemma\ 2, 3)$$

$$\rightarrow find(next(E', B), I(B, pre(operand(e'')))\&$$
$$event(operand(e'')) = p) = e''$$
$$\&find(expr(index(E')), pre(operand(e''')) =$$
$$pre(operand(e))\&$$
$$event(operand(e''')) = event(operand(e)) = e'''$$
$$\&has(relation(index(E')), (e'', e''')) \qquad (by(ii))$$

$$\rightarrow find(next(E', B), I(B, pre(operand(e'')))\&$$
$$event(operand(e'')) = p) = e''$$
$$\&has(expr(index(E')), e''')\&$$
$$pre(operand(e''')) = pre(operand(e))\&$$
$$event(operand(e''')) = event(operand(e))\&$$
$$has(relation(index(E')), (e'', e''')) \qquad (Lemma\ 2)$$

$$\rightarrow has(select(expr(index(E')),$$
$$find(next(E', B), I(B, pre(operand(e'')))\&$$
$$event(operand(e'')) = p) = e''$$
$$\&has(relation(index(E')), (e'', e1))), e''')$$
$$\&pre(operand(e''')) = pre(operand(e))\&$$
$$event(operand(e''')) = event(operand(e)) \quad (Lemma\ 2)$$

$$\rightarrow has(next(E', B'), e''')\&$$
$$pre(operand(e''')) = pre(operand(e))\&$$
$$event(operand(e''')) = event(operand(e))$$
$$(by\ the\ semantics\ of\ next)$$

$$\rightarrow find(next(E', B'),$$
$$pre(operand(e''')) = pre(operand(e))\&$$
$$event(operand(e''')) = event(operand(e))) = e'''$$
$$(Lemma\ 2)\square$$

Theorem 2. *If (iii) holds, then*

$$valid(E, B) \rightarrow valid(E', B)$$

Proof. This is showed by induction on the structure of B, too.

1. It is evident that the formula holds for $B = new$.

2. We suppose that the formula holds for B. Let us denote $B' = set - behavior(B, P)$.

Then

$$valid(E, B') =$$
$$= valid(E, B)\&find(next(E, B), I(B, pre(operand(e)))\&$$
$$event(operand(e)) = p) = e$$
$$\rightarrow valid(E', B)\&find(next(E, B), I(B, pre(operand(e)))\&$$
$$event(operand(e)) = p) = e$$
$$(induction)$$
$$\rightarrow valid(E', B)\&has(next(E, B), e)\&$$
$$I(B, pre(operand(e)))$$
$$\&event(operand(e)) = p \qquad (Lemma\ 2)$$

$$\rightarrow valid(E', B)\&$$
$$find(next(E', B), pre(operand(e')) = pre(operand(e))\&$$
$$event(operand(e')) = event(operand(e))) = e'$$
$$\&I(B, pre(operand(e)))\&event(operand(e)) = p \quad (by(iii))$$

$$\rightarrow valid(E', B)\&$$
$$has(next(E', B), e')\&pre(operand(e')) = pre(operand(e))$$
$$\&event(operand(e')) = event(operand(e))\&$$
$$I(B, pre(operand(e)))\&event(operand(e)) = p \quad (Lemma\ 2)$$

$$\rightarrow valid(E', B)\&has(next(E', B), e')\&$$
$$I(B, pre(operand(e')))\&event(operand(e')) = p$$
$$(Lemma\ 1)$$

$$\rightarrow valid(E', B)\&$$
$$find(next(E', B), I(B, pre(operand(e'))) = e'$$
$$\&event(operand(e')) = p) = \qquad (Lemma\ 2)$$
$$= valid(E', B') \qquad \square$$

Denotion.

1. If two GPE's E and E' satisfy (i) and (ii), then this fact is denoted by

$$E \Rightarrow E'$$

2. If for two GPE's E and E'

$$E \Rightarrow E' \ \& \ E' \Rightarrow E$$

then we write

$$E \Longleftrightarrow E'.$$

By Theorem 1 and Theorem 2 we have the Corollary: If for two GPE's E and E'

$$E \Longleftrightarrow E'$$

then

$$\forall B \in behavior(valid(E, B) \leftrightarrow valid(E', B)) \quad \square$$

References

[1] BRUEGGE, B. and HIBBARD, P., Generalized Path Expressions:A High-Level Debugging Mechanism, *The Journal of Systems and Software* **3** 1983, 265-276.

[2] CHIEN, NG. H., A High-Level Debugging Method and Its Correctness, Candidate Disertation, Department of General Computer Sciences, Eötvös Loránd University, Budapest, 1987.

[3] GUTTAG, J. V. and HORNING, J. J., The Algebraic Specification of Abstract data types, *Acta Informatica,* **10**, 1978, 27-52.

[4] VARGA, L., An Example for Algebraic Specification of Programming Systems, Proc. of Conference on Automata, Languages and Programming Systems, Salgótarján, May 1986 (Hungary) K. Marx Univ. of Economics, Dept. of Math., Budapest DM 86-4, 1986 .

(Received January 4, 1988)

Ng. H. CHIEN L. VARGA

Eötvös Loránd University
Department of General Computer Science
H-1117,Budapest, Bogdánfy u. 10/b.

HUNGARY