Synthesis of abstract algorithms

By L. VARGA

1. Introduction

In the second half of the sixties the programming methodology evolved on the basis of the recognation that the correctness of a program, corresponding to a given specification can be proven, and during the last ten years, the programming methodology has become one of the most important branches of computer science.

The programming methodology has been changed radically and it now includes broad research areas that deal with both practical and theoretical questions of program development and management. The current research directions in programming methodology is summarized in [14] and a more detailed description of its principal subareas can be found in the books [13], [15].

Within the scope of programming methodology important research has been concentrated on studying the correctness of programs. In recent years there has been increasing activity in this field. There are two different approaches to achieving program correctness.

1. The *engineering approach* has been aiming at turning the art of programming into an engineering science. Its aim is to develop more efficient software tools and specify standard that can be used in the process of development of programs as a means to improve the relaibility and reduce the software cost. Complete systems called automated software evaluation systems, for different phase of software life cycle have been developed such a system includes automatic tools for requirement and design analysis, testing, maintenance etc. A comprehensive survey of the software tools and automated software evaluation systems can be found in [12].

Altough the engineering approach generally is not capable of demonstrating the correctness of a program, this approach seems to be an effective approach to the validation of programs in practice.

2. The *analytic approach* uses program verification methods to ensure that the desired program conformes to its correctness specifications. The role of software verification, the proof techniques and various verification systems are discussed in the survey papers [5], [7].

As far as the analytic approach is concerned, in the beginning the attention was focused on a *posteriori verification* of programs. That is, the problem of the program correctness proof was approached in the following way. Given a program

and a specification, it is to be proven, that the program realizes exactly the mapping stated in the specification.

Later on, however it turned out, that different programs, which realize the same mapping may be essentially different from the point of view of the correctness proof. The difficulty of a proof depends on the complexity of the program.

This led to the conclusion, that the correctness of a program has to be established during its construction. Programs have to be designed such a way, that the proof of their correctness should be simple. In fact, first a correctness proof has to be constructed and than a corresponding program to this proof should be given. This is the *constructive approach* to achieving program correctness, which represents one of the most significant advance in programming methodology. This approach has produced various program design and construction methods ([1], [3], [10], [17]). This methods require as input a specification of what is to be achieved and produce as output a program text which is a specification of how is to be achieved.

The methods initiated by the constructive approach have made a fundamental contribution to the *synthesis of programs* in extracting principles for deriving programs systematically from their specifications. These principles are formulated precisely enough to be carried out by an automatic synthesis system in [8], [9].

The main steps of an automatic program synthesis system are.

1. The system accepts specifications, which describe some function to be realized by means of primitives of a well defined system. Generally these primitives are the statements of a programming language.

The basic approach is to transform the specifications step by step according to certain transformation rules, which are guided by two kinds of strategic controls:

2. Some transformations attempt to transform the specifications into equivalent specifications or replace them by stronger assertions about the states of the desired programs. The aim is to produce an appropriate form for applying programming strategies.

3. Other transformations attempt to transform the specification into the desired program text, decomposing a given program description into subprogram descriptions.

These new descriptions are transformed into newer ones repeatedly until a program text of a source programming language is obtained.

In this paper we concentrate on the third problem, and programming strategies are formulated for developing the desired abstract programs step by step using the Hoare's deductive system [3]. The levels of abstraction are used with the Vienna Definition Language [16]. This language permits concentration on logical solutions to problems, rather than the form and contraints within that the solution must be stated. The language helps the programmers to think in terms of hierarchy of macro statements and express structured programming logic in stepwise refinement of a program and its data structure.

We are influenced by the strategies formulated by N. Dershowitz and Z. Manna [2]. The essence of our paper is the presentation of similar strategies applied to VDL-programs.

The programming strategies are based on the Hoare's deductive system. Extending the Hoare's methods to VDL-statements of similar structure to the statements of usual programming languages is relatively simple. However the indeterminism — which means, that the VDL-language allows programs to be written

in that the execution order of the statements is not predefined — presents a special problem. The programming strategies of such structures can be formulated by using the results of correctness proof of parallel programs [4]. A detailed treatment of correctness proofs of parallel programs can be found in Gries and Owicky's papers [11].

The next section presents the basic strategies for developing VDL-programs from appropriate forms of specifications.

In Section 3 the VDL-graph is defined as an abstraction of a class of data structures. The VDL-graph specify a connected graph which has one entry node at least, but may have several terminal nodes and there must be a path from one entry node at least to a terminal node through every node in the graph.

In Section 4 the deductive technique is illustrated by the example of an abstract graph walk algorithm. Sections 5 and 6 demonstrate the application of VDL-graph to specifying a linkage editor and an inverse assembler model, respectively.

2. Strategies for stepwise refinement

In this section the main strategies are formulated for developing a VDL-program from its specification.

The following notation will be used

$$\{R\}\ P\ \{Q\}$$

where Q and R are logical statements about states of the abstract machine (VDL-machine) and P is a VDL-program (program-tree). This may be interpreted as follows: If Q is true before execution of a VDL-program P, and if the execution terminates, then R will hold after executing P. This notation expresses the partial correctness of a VDL-program P with respect to its input specification Q and output specification R.

Our initial goal is to synthesize a program of the general form

$$\{R(\xi)\}\ stmt(x_1, x_2, \dots, x_n)\ \{Q(\xi_0) \wedge x_1 = f_1(\xi_0) \wedge x_2 = f_2(\xi_0) \wedge \dots \wedge x_n = f_n(\xi_0)\}$$

where ξ_0 is the initial state of the abstract machine and $f_1, f_2, \dots, f_n$ are given functions. We require that the output state ξ of the desired program *stmt* satisfy the given specification $R(\xi)$, provided the initial state ξ_0 satisfies the given input specification $Q(\xi_0)$.

In order to synthesize the program this top-level goal may be transformed into equivalent goal or it may be replaced by a stronger goal, which can be achieved by an assignement (value returning) instruction or reduced to subgoals by using the following strategies.

2.1. The strategy of assignment. Given the goal of the form

$$\{R(e_0';\ \mu(\xi;\ \langle s-c_1\colon e_1'\rangle, \dots, \langle s-c_n\colon e_n'\rangle))\}\ e_0'\colon stmt\ (x_1, \dots, x_k)\ \{Q(x_1, \dots, x_k;\ \xi)\}$$

where the selectors $s-c_1, s-c_2, \dots, s-c_n$ are independent, and

$$e_i' = e_i(x_1, \dots, x_k;\ \xi), \quad i = 0, 1, \dots, n$$

then this goal can be achieved by the following value returning instruction

$$\begin{aligned} &stmt(x_1, x_2, \ldots, x_k) = \\ &\quad \text{PASS}: e_0(x_1, \ldots, x_k; \xi) \\ &\quad s-c_1: e_1(x_1, \ldots, x_k; \xi) \\ &\quad \vdots \\ &\quad s-c_n: e_n(x_1, \ldots, x_k; \xi). \end{aligned}$$

2.2. The conditional strategy. A goal of the form

$$\{q_1 \vee q_2 \vee \ldots \vee q_n\}\ stmt\ \{p\}$$

can be reduced by the conditional instruction

$$\begin{aligned} &stmt = \\ &\quad p_1 \to stmt_1 \\ &\quad p_2 \to stmt_2 \\ &\quad \vdots \\ &\quad T \to stmt_n \end{aligned}$$

to the subgoals

$$\{q_1\}\ stmt_1\ \{p \wedge p_1\},\ \{q_2\}\ stmt_2\ \{p \wedge \neg p_1 \wedge p_2\},\ \ldots,\ \{q_n\}\ stmt_n\ \{p \wedge \neg p_1 \wedge \ldots \wedge \neg p_{n-1}\}.$$

Any control tree can be constructed by using only the following two macro definitions:

$$\begin{aligned} &stmt = \\ &\quad stmt_1; \\ &\quad\quad stmt_2 \end{aligned}$$

and

$$\begin{aligned} &stmt = \\ &\quad null; \\ &\quad\quad stmt_1, \\ &\quad\quad \vdots \\ &\quad\quad stmt_n \end{aligned}$$

The strategies for these basic forms will now be given.

2.3. The strategy of composition. A goal

$$\{r\}\ stmt\ \{p\}$$

can be decomposed by the instruction

$$\begin{aligned} &stmt = \\ &\quad stmt_1; \\ &\quad\quad stmt_2 \end{aligned}$$

to the subgoals

$$\{q\}\ stmt_2\ \{p\} \text{ and } \{r\}\ stmt_1\ \{q\}.$$

2.4. The strategy of indeterminism. Given a conjunctive goal of the form

$$\{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}\ stmt\ \{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}$$

then it can be reduced by the instruction

$$\begin{aligned} stmt = & \\ & null; \\ & \quad stmt_1, \\ & \quad stmt_2, \\ & \quad \vdots \\ & \quad stmt_n \end{aligned}$$

to the following subgoals

$$\{q_1\}\ stmt_1\ \{p_1\},\ \{q_2\}\ stmt_2\ \{p_2\}, \ldots, \{q_n\}\ stmt_n\ \{p_n\}$$

provided these theorems are *interference-free.* This property of the theorems is defined as follows:

Definition 2.1. Given a control tree t with the theorem

$$\{q\}\ t\ \{p\} \tag{i}$$

and the value returning instruction *stmt* with some precondition pre(*stmt*). If the execution of *stmt* after t does not alter the validity of q, that is

$$\{q\}\ stmt\ \{\text{pre}\,(stmt) \wedge q\}$$

and the execution of *stmt* before any *st* within t does not alter the validity of the precondition of *st*, that is

$$\{\text{pre}\,(st)\}\ stmt\ \{\text{pre}\,(stmt) \wedge \text{pre}\,(st)\}$$

then we say that *stmt* does not interfere with theorem (i).

Definition 2.2. Given the theorems

$$\{q_1\}\ stmt_1\ \{p_1\},\ \{q_2\}\ stmt_2\ \{p_2\}, \ldots, \{q_n\}\ stmt_n\ \{p_n\} \tag{ii}$$

and let st_i be a value returning instruction within $stmt_i$. If for all i, $i=1, 2, \ldots, n$ st_i does not interfere with

$$\{q_j\}\ stmt_j\ \{p_j\}; \quad j = 1, 2, \ldots, n, \quad j \neq i$$

then the theorems (ii) are interference-free.

Accordingly, in applying the strategy of indeterminism, we must ensure the interference-free. If $q_1, q_2, \ldots, q_n$ are statements about different components of the state ξ and similarly $p_1, p_2, \ldots, p_n$ do not contain common variables and the macros $stmt_1, stmt_2, \ldots, stmt_n$ operate on independent components of ξ then the interference-freeness obviously satisfies.

The conditional strategy has an important special case:

2.5. The strategy of iteration. A goal of the form

$$\{q\}\ stmt\ \{p\}$$

can be decomposed by the iteration

$$
\begin{aligned}
& stmt = \\
& \quad p_1 \rightarrow stmt_2 \\
& \quad T \rightarrow stmt; \\
& \qquad\qquad stmt_1
\end{aligned}
$$

to the subgoals

$$\{p\}\ stmt_1\ \{p \wedge \neg p_1\} \quad \text{and} \quad \{q\}\ stmt_2\ \{p \wedge p_1\}$$

provided the iteration terminates.

Here the conjunctive goal $p \wedge p_1$ is achieved by forming an iteration so that the predicate p remains invariant during the iteration until the predicate p_1 is found false.

In the special case of $q = p \wedge p_1$, the instruction

$$
\begin{aligned}
& stmt = \\
& \quad p_1 \rightarrow null \\
& \quad T \rightarrow stmt; \\
& \qquad\qquad stmt_1
\end{aligned}
$$

can be used for reducing our goal to the subgoal

$$\{p\}\ stmt_1\ \{p \wedge \neg p_1\}.$$

At last a rule will be given here, which can be used for proving the termination of an iteration.

2.6. The rule of termination. Let the iteration

$$
\begin{aligned}
& stmt = \\
& \quad p_1 \rightarrow null \\
& \quad T \rightarrow stmt; \\
& \qquad\qquad stmt_1
\end{aligned}
$$

with the precondition

$$\text{pre}\,(stmt) \equiv p$$

be given.

Let u be an integer function of the appropriate variables. If

a) $p \supset u \geqq 0$

b) $p \wedge \neg p_1 \supset u > 0$

c) $\{u' < u\}\ stmt_1\ \{p \wedge \neg p_1\}$ (u' is the value of u after $stmt_1$)

d) and any assignment statement, that can be executed parallel with the statement *stmt* does not interfere with the theorem c;

then the iteration terminates.

For example, the termination of the iteration

$$
\begin{aligned}
& process\,(t) = \\
& \quad \text{length (list)} = 0 \rightarrow null \\
& \quad T \rightarrow process\,(\text{tail}\,(t)); \\
& \qquad proc\,(\text{head}\,(t))
\end{aligned}
$$

is guaranted, because for the function

$$u(t) = \text{length}(t)$$

with the precondition

$$\text{is-pred-list}(t)$$

all the criterions a)—d) hold.

3. The VDL-graph

The graph, that can be walked from its entry nodes, plays an important role in programming. In this section the definition of the VDL-graph is given, which can be viewed as an abstraction of graph data structures.

Let

$$\text{is-node-set} = (\{\langle s\colon \text{is-node}\rangle | \text{is-select}(s)\})$$
$$\text{is-node} = (\langle s\text{-value}\colon \text{is-pred}\rangle, \langle s\text{-desc}\colon \text{is-select-list}\rangle)$$

where "is-select" and "is-pred" represent arbitrary predicates. Such an object is shown in Figure 3.1, where

$$a \in \{x | \text{is-pred}(x)\}$$

and

$$s, s_i \in \{s' | \text{is-select}(s')\}$$

Let

$$\text{is-node-set}(f) = T$$

The notation $t \in f$ is used if

$$(\exists s, \text{is-select}(s) = T)(s(f) = t).$$

Definition 3.1. Let $t \in f$ and $n \in f$. The node n *refers* to t if and only if

$$(\exists i,\ 1 \leqq i \leqq \text{length}(s\text{-desc}(n)))(\text{elem}(i)(s\text{-desc}(n))(f) = t).$$

Notationally, we shall use the form

$$n \Rightarrow t.$$

Definition 3.2. The node t_k is *reachable* from node t_1, or there exists a *reference path* from t_1 to t_k if and only if

$$t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow t_k, \quad (t_i \in f, \quad i = 1, 2, \ldots, k).$$

We shall use the following notation for the reference path

$$t_1 \Rightarrow * t_k.$$

Definition 3.3. The set of VDL-*graph* is

$$\{g | \text{is-pred-graph}(g)\}$$

where

$$\text{is-pred-graph} = \text{is-node-set}$$

and there exists a non-empty subset $M(g)$ of the nodes of g distinquished with the property that any node $n \in g$ and $n \notin M(g)$ can be reached from at least one element of $M(g)$.

The elements of $M(g)$ are called *directly reachable nodes.*

Consequently each node of a VDL-graph can be reached from at least one directly reachable node.

Definition 3.4. Let root (i) be the function, for which

$$\text{root}\,(i) = s_i, \quad i = 1, 2, \ldots, n$$

if

$$M(g) = \{s_1(g), s_2(g), \ldots, s_n(g)\}$$

and let

$$\text{value}\,(n) = s\text{-value}\,(n),$$

$$\text{next}\,(i)(n) = \big(\text{elem}\,(i)(s\text{-desc}\,(n))(g), \;\; 1 \leqq i \leqq \text{length}\,(s\text{-desc}\,(n)\big).$$

These functions can be used as selectors, for example

$$\text{value.next}\,(2).\text{next}\,(1).\text{root}\,(3)(g) = \text{value}\,(\text{next}\,(2)(\text{next}\,(1)(\text{root}\,(3)(g)))).$$

Using the functions defined above, the structure of a VDL-graph can be visualized by a graph. For example, the VDL-graph, denoted by the following relations

$$\begin{aligned}
g &= \mu_0(\langle \text{root}\,(1)\colon\ n_1\rangle, \langle \text{root}\,(2)\colon\ n_2\rangle, \langle s_3\colon\ n_3\rangle, \langle s_4\colon\ n_4\rangle, \langle s_5\colon\ n_5\rangle),\\
n_1 &= \mu_0(\langle s\text{-value}\colon\ a\rangle, \langle s\text{-desc}\colon\ \langle s_3, s_4\rangle\rangle),\\
n_2 &= \mu_0(\langle s\text{-value}\colon\ b\rangle, \langle s\text{-desc}\colon\ \langle s_4\rangle\rangle),\\
n_3 &= \mu_0(\langle s\text{-value}\colon\ c\rangle, \langle s\text{-desc}\colon\ \langle\ \rangle\rangle),\\
n_4 &= \mu_0(\langle s\text{-value}\colon\ d\rangle, \langle s\text{-desc}\colon\ \langle s_5, s_2\rangle\rangle),\\
n_5 &= \mu_0(\langle s\text{-value}\colon\ e\rangle, \langle s\text{-desc}\colon\ \langle\ \rangle\rangle),
\end{aligned}$$

can be represented by the graph shown in Figure 3.2.

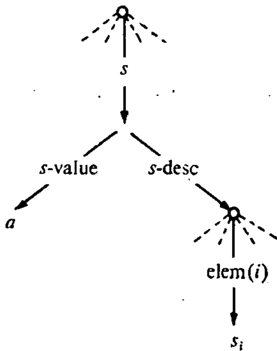

Fig. 3.1
A node set

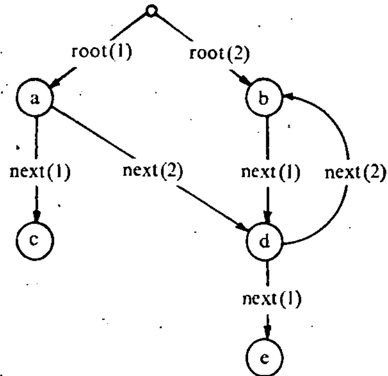

Fig. 3.2
A VDL-graph

The nodes of the graph in the Fig. 3.2 are circles and the values yielded by the nodes are put in the circles. The relationships between the nodes are represented by arrows and the arrows are named by the function next (i).

The figure of a VDL-graph reflects its structure in this way, but the formula of a VDL-graph does not do it directly. However it is not difficult to construct a formula that also satisfies this requirement. This problem is not dealt here.

In Definition 3.4 selection operations are defined on the VDL-graph. Construction operations can also defined on it, but we intend to deal with statical VDL-graph only, hence construction operations are not defined.

4. The synthesis of the graph walk algorithm

The graph walk is a fundamental operation. Most of the selection and construction operations of a graph can be established on it.

A graph walk can be carried out according to different strategies. In a graph walk algorithm each node of the graph is processed one after the other. The walk strategy determines the order of the nodes to be processed. In the following, we present a systematic development of a general graph walk algorithm, where the walk strategy and the operations over the nodes are not specified. In this way an abstraction of the graph walk algorithms is given from which concrete graph walks can be deduce by the specification of the walk strategy and the operation over the nodes.

Our top level goal is

Goal 1.

$$\{\text{is-target-graph}\ (g')\}\ g'\colon \mathit{walk}\ (g)\ \{\text{is-source-graph}\ (g)\}$$

where the map

$$\text{trans}\colon\ \{x|\text{is-source}\ (x)\} \to \{y|\text{is-target}\ (y)\}$$

is not specified. Therefore the function trans will be used as a parameter of the desired algorithm:

$$\mathit{walk}\ (g;\ \text{trans}).$$

Let the set of states of the abstract machine be

$$\{\xi|\text{is-state}\ (\xi)\}$$

where

$$\text{is-state} = (\langle s\text{-graph: is-pred-graph}\rangle,\ \langle s\text{-table: is-table}\rangle,\ \langle s\text{-control:is-control}\rangle).$$

The component s-graph (ξ) is the graph g to be walked. The component s-table (ξ) is used to mark which nodes of the graph g have been processed. Therefore

$$\text{is-table} = (\{\langle s\text{: is-value}\rangle|\text{is-select}\ (s)\})$$

where

$$\widehat{\text{is-value}} = \{Y, N\}$$

so that

$$s(s\text{-table}\ (\xi)) = Y$$

if and only if the s-value $(s(g))$ has been maped to a target value.

We do not intend to specify the walk strategy. Therefore we introduce the function

next-selector

as follows:

Definition 4.1. The function next-selector is a function over the set

$$\{t \mid \text{is-table}\,(t)\}$$

and the range of the function is

$$\widehat{\text{is-select}} \cup \{\Omega\}$$

so that if

$$(\exists s,\ \text{is-select}\,(s) = T)(s(t) = F)$$

then next-selector provides a selector s with the property

$$s(t) = N$$

else

$$\text{next-selector}\,(t) = \Omega.$$

Informally, the function next-selector (t) provides one of the selectors of the table t as $s(t)=N$, if such an s exists and the object Ω otherwise.

It is supposed that if the function next-selector is applied to the same table t several times the result is the same.

The function next-selector will also be used as a formal parameter of the desired algorithm and the formal parameter g will be omitted, because it is a component of the state ξ:

$$\textit{walk}\ (;\ \text{next-selector, trans})$$

and it is not a value returning macro.

We can now define the initial state ξ_0 of the abstract machine as follows

$$\xi_0 = \mu_0(\langle s\text{-graph:}\ g\rangle, \langle s\text{-table:}\ t_0\rangle, \langle s\text{-control:}\textit{walk}\ (;\text{next-selector, trans})\rangle)$$

where

$$\text{is-source-graph}\ (g) = T$$

and

$$t_0 = \mu_0(\{\langle s\colon\ N\rangle \mid s(g) \in M(g)\}).$$

Hence the input specification is

$$\varphi(\xi_0)\colon\ s\text{-table}\ (\xi_0) = \mu_0(\{\langle s\colon\ N\rangle \mid s(g) \in M(g)\}) \wedge\ \text{is-source-graph}\ (g) \wedge g = s\text{-graph}\ (g),$$

and our goal is

Goal 2.

$$\{\text{is-target-graph}\ (s\text{-graph}\ (\xi))\}\ \textit{walk}\ (;\ \text{next-selector, trans})\ \{\varphi(\xi_0)\}$$

where the formal specifications of the formal parameters next-selector and trans are disregarded.

In order to synthesize the program, we must find a sequence of transformations to yield an equivalent description of the specification, that can be reduced by apply-

ing one of the strategies given in Section 2. First let us intend to prepare the application of the strategy of iteration.

To produce an appropriate form of the output specification, let us specify the invariable properties of the data structures.

Let

$$\alpha(s, \xi) \equiv s(s\text{-table}\,(\xi)).$$

Our graph walk strategy could be the following: Each node $s'(g)$ with the property

$$\alpha(s', \xi) = \Omega$$

must be reachable from at least one node $s(g)$ that waits for being processed with the property

$$\alpha(s, \xi) = N.$$

Hence, the formal specification of the data components of ξ is

$$Q_1(\xi)\colon\; R_1(\xi, g) \wedge R_2(\xi, g) \wedge R_3(\xi, g) \wedge R_4(\xi, g) \wedge R_5(g) \wedge g = s\text{-graph}\,(\xi),$$

where

$$R_1(\xi, g)\colon\; \big(\forall s, \alpha(s, \xi) \neq \Omega\big)\big(s(g) \neq \Omega\big) \wedge \text{is-table}\,\big(s\text{-table}\,(\xi)\big),$$
$$R_2(\xi, g)\colon\; \big(\forall s, \alpha(s, \xi) = Y\big)\big(\text{is-target}\,(s\text{-value}\,(s(g)))\big),$$
$$R_3(\xi, g)\colon\; \big(\forall s, \text{is-target}\,(s\text{-value}\,(s(g)))\big)\big(\alpha(s, \xi) = Y\big),$$
$$R_4(\xi, g)\colon\; \big(\forall s', s'(g) \neq \Omega \wedge \alpha(s', \xi) = \Omega\big)\big((\exists s, \alpha(s, \xi) = N)(s(g) \Rightarrow *s'(g))\big),$$
$$R_5(g)\colon\; \text{is-mixed-graph}\,(g) \wedge \text{is-mixed} = \text{is-source} \vee \text{is-target}.$$

Theorem 4.1.

$$Q_1(\xi_0) \supset \varphi(\xi_0).$$

Theorem 4.2.

$$Q_1(\xi) \wedge \text{next-selector}\,\big(s\text{-table}\,(\xi)\big) = \Omega \supset \text{is-target-graph}\,\big(s\text{-graph}(\xi)\big).$$

Hence our goal may be

Goal 3.

$$\big\{Q_1(\xi) \wedge \text{next-selector}\,\big(s\text{-table}(\xi)\big) = \Omega\big\}\; \textit{walk}\,(;\, \text{next-selector, trans})\; \{Q_1(\xi)\}.$$

This suggests achieving Goal 3 with a recursive call applied to the macro *walk* as follows:

$$\begin{aligned} &\textit{walk}\,(;\, \text{next-selector, trans}) = \\ &\quad \text{next-selector}\,\big(s\text{-table}\,(\xi)\big) = \Omega \rightarrow \textit{null}\,. \\ &\quad T \rightarrow \textit{walk}\,(;\, \text{next-selector, trans}); \\ &\qquad \textit{process}\,(;\, \text{next-selector, trans}) \end{aligned}$$

which reduces Goal 3 to the subgoal

Goal 4.

$$\{Q_1(\xi)\}\; \textit{process}\,(;\, \text{next-selector, trans})\; \big\{Q_1(\xi) \wedge \text{next-selector}\,\big(s\text{-table}\,(\xi)\big) \neq \Omega\big\}.$$

Furthermore we must ensure the termination of the iteration. To achieve the termination, we could require that the number of the nodes $s(g)$ with the property

$$\alpha(s, \xi) = Y$$

be strictly increased with each iteration. Let $U(\xi)$ be the number of nodes with the above property, then our goal is

Goal 5.

$$\{Q_1(\xi) \wedge U(\xi) > a\} \; \textit{process} \; (; \text{next-selector, trans})$$
$$\{Q_1(\xi) \wedge \text{next-selector} \; (s\text{-table} \; (\xi)) \neq \Omega \wedge U(\xi) = a\}.$$

Using the strategy of composition this can be achieved by the macro

process (; next-selector, trans) =
process-node (s; trans);
s: *produce-selector*(; next-selector)

reducing Goal 5 to two subgoals

Goal 6.

$$\{Q_1(\xi) \wedge \alpha(s, \xi) = Y \wedge U(\xi) = a+1\} \; \textit{process-node} \; (s; \text{trans})$$
$$\{Q_1(\xi) \wedge \alpha(s, \xi) = N \wedge U(\xi) = a\}$$

and

Goal 7.

$$\{Q_1(\xi) \wedge \alpha(s, \xi) = N \wedge U(\xi) = a\} \; s\text{: } \textit{produce-selector}(; \text{next-selector})$$
$$\{Q_1(\xi) \wedge \text{next-selector} \; (s\text{-table} \; (\xi)) \neq \Omega \wedge U(\xi) = a\}.$$

Goal 7 can be achieved by the strategy of assignment:

produce-selector (; next-selector) =
PASS: next-selector (s-table (ξ)).

In order to find a strategy for reducing Goal 5, let us isolate the effect of selector s on predicate $Q_1(\xi)$. Predicate Q_1 has five components. The predicate

$$\alpha(s, \xi) = Y \vee \alpha(s, \xi) = N$$

is of no effect on the first component of Q_1.

Let us consider the components R_2 and R_3.

Theorem 4.3.

$$R_2(\xi, g) \wedge R_3(\xi, g) \equiv R_{21}(\xi, g, s) \wedge R_{31}(\xi, g, s) \wedge Q_{21}(\xi, g, s),$$

where

$R_{21}(\xi, g, s)$: $(\forall s', \alpha(s', \xi) = Y \wedge s' \neq s)(\text{is-target} \; (s\text{-value} \; (s'(g))))$,
$R_{31}(\xi, g, s)$: $(\forall s', \text{is-target} \; (s\text{-value} \; (s'(g))) \wedge s' \neq s)(\alpha(s', \xi) = Y)$,
$Q_{21}(\xi, g, s)$: $\alpha(s, \xi) = Y \wedge \text{is-target} \; (s\text{-value} \; (s(g)))$.

Let us see the component R_4. We have different cases:

1. $s(g) \not\Rightarrow * s'(g)$ ($s'(g)$ is not reachable from $s(g)$),
2. $s(g) \Rightarrow s^*(g) \Rightarrow * s'(g)$ and $\alpha(s^*, \xi) \neq \Omega$,
3. $s(g) \Rightarrow s^*(g) \Rightarrow * s'(g)$ and $\alpha(s^*, \xi) = \Omega$,
4. $s(g) \Rightarrow s'(g)$.

Obviously, we have not to bother with the first two cases. Hence

Theorem 4.4.

$$R_4(\xi', g) \equiv R_{41}(\xi, g, s) \wedge Q_{22}(\xi, \xi', g, s)$$

where

$$R_{41}(\xi, g, s): \big(\forall s', s'(g) \neq \Omega \wedge \alpha(s', \xi) = \Omega \wedge \neg((s(g) \Rightarrow s^*(g) \Rightarrow * s'(g) \wedge \alpha(s^*, \xi) = \Omega) \vee s(g) \Rightarrow s'(g))\big)\big(\exists \bar{s}, \alpha(\bar{s}, \xi) = N \wedge \bar{s} \neq s\big)\big(\bar{s}(g) \Rightarrow * s'(g)\big),$$

$$Q_{22}(\xi, \xi', g, s): \big(\forall s', s(g) \Rightarrow s^*(g) \Rightarrow s'(g) \wedge \alpha(s^*, \xi) = \alpha(s', \xi) = \Omega\big)\big(\alpha(s^*, \xi') = N\big) \wedge \big(\forall s', s(g) \Rightarrow s'(g) \wedge \alpha(s', \xi) = \Omega\big)\big(\alpha(s', \xi') = N\big).$$

Theorem 4.5.

$$Q_1(\xi') \wedge \alpha(s, \xi') = Y \wedge U(\xi') = a+1 \equiv Q_2(\xi', s) \wedge Q_{21}(\xi', g, s) \wedge Q_{22}(\xi, \xi', g, s)$$

where

$$Q_2(\xi', s) \equiv R_1(\xi', g) \wedge R_{21}(\xi', g, s) \wedge R_{31}(\xi', g, s) \wedge R_{41}(\xi', g, s) \wedge R_5(g) \wedge g = s\text{-graph}(\xi').$$

Theorem 4.6.

$$Q_2(\xi, s) \wedge \alpha(s, \xi) = N \supset Q_1(\xi) \wedge \alpha(s, \xi) = N.$$

Hence our goal is

Goal 8.

$$\{Q_2(\xi', s) \wedge Q_{21}(\xi', g, s) \wedge Q_2(\xi, \xi', g, s)\}\ \textit{process-node}\,(s(s\text{-graph}(\xi)); s;\ \text{trans})\ \{Q_2(\xi, s) \wedge \alpha(s, \xi) = N\}.$$

We try to achieve it with the strategy of indeterminism of the form

process-node (n; s; trans) =
  process-value (trans (s-value (n)), s),
  process-desc (s-desc (n))

reducing Goal 8 to the subgoals

Goal 9.

$$\{Q_2(\xi, s) \wedge Q_{21}(\xi, g, s)\}\ \textit{process-value}\,(v, s)\ \{Q_2(\xi, s) \wedge \alpha(s, \xi) = N \wedge v = \text{trans}\,(s\text{-value}\,(s(g)))\}$$

and

Goal 10.

$$\{Q_2(\xi', s) \wedge Q_{22}(\xi', \xi, g, s)\}\ \textit{process-desc}\,(\text{list})\,\{Q_2(\xi, s) \wedge \text{list} = s\text{-desc}\,(s(g))\}$$

provided these are interference-free.

Goal 8 can be achieved by using the strategy of assignment:

$$\begin{aligned}&\text{process-value}\ (v, s) = \\ &\quad s\text{-graph}: \mu(s\text{-graph}\ (\xi);\ \langle s\text{-value}_0\, s:\ v\rangle) \\ &\quad s\text{-table}: \mu(s\text{-table}\ (\xi);\ \langle s:\ Y\rangle).\end{aligned}$$

Let us consider Goal 10. The significant part of the specification is Q_{22}. In order to try to achieve it by iteration, we attempt to apply the following transformation:

Theorem 4.7.

$$Q_{22}(\xi', \xi, g, s) \equiv Q_{221}(w_1, w_2) \wedge \text{length}\ (w_2) = 0,$$

where

$$\begin{gathered}Q_{221}(w_1, w_2):\ (\forall s', s(g) \Rightarrow s^*(g) \Rightarrow s'(g) \wedge \alpha(s^*, \xi) = \alpha(s', \xi) = \Omega \wedge s^* \in w_1) \\ (\alpha(s^*, \xi') = N) \wedge (\forall s', s(g) \Rightarrow s'(g) \wedge \alpha(s', \xi) = \Omega \wedge s' \in w_1) \\ (\alpha(s', \xi') = N) \wedge \widehat{w_1 w_2} = s\text{-desc}\ (s(g)).\end{gathered}$$

We can now achieve our goal by creating an iteration whose exit is length $(w_2) = 0$ and whose invariant assertion is $Q_2 \wedge Q_{221}$. The desired program is

$$\begin{aligned}&\text{process-desc}\ (w) = \\ &\quad \text{length}\ (w) = 0 \rightarrow \mathit{null} \\ &\quad T \rightarrow \text{process-desc}\ (\text{tail}\ (w)); \\ &\qquad\qquad \text{set}\ (\text{head}\ (w))\end{aligned}$$

which reduces Goal 10 to the subgoal

Goal 11.

$$\begin{gathered}\{Q_2(\xi', s) \wedge Q_{221}(\widehat{w_1 \langle} s^*\rangle,\ \text{tail}\ (w_2))\}\ \mathit{set}\ (s^*) \\ \{Q_2(\xi, s) \wedge Q_{221}(w_1, w_2) \wedge s^* = \text{head}\ (w_2) \wedge \text{length}\ (w_2) \neq 0\}.\end{gathered}$$

Obviously, the termination is now ensured.

Goal 11 can be achieved by the conditional strategy:

$$\begin{aligned}&\text{set}\ (s) = \\ &\quad s(s\text{-table}\ (\xi)) \neq \Omega \rightarrow \mathit{null} \\ &\quad T \rightarrow \text{link}\ (s)\end{aligned}$$

which reduces Goal 11 to the subgoal

Goal 12.

$$\{\alpha(s, \xi) = N\}\ \mathit{link}\ (s)\ \{\alpha(s, \xi) = \Omega\}.$$

Goal 12 can be achieved by a simple assignment:

$$\begin{aligned}&\text{link}\ (s) = \\ &\quad s\text{-table}: \mu(s\text{-table}\ (\xi);\ \langle s: N\rangle).\end{aligned}$$

Theorem 4.8. The theorems in Goal 9 and Goal 10 are interference-free. The complete program is

walk (; next-selector, trans) =
next-selector (s-table (ξ)) $= \Omega \rightarrow$ *null*
$T \rightarrow$ *walk* (; next-selector, trans);
process (; next-selector, trans)

process (; next-selector, trans) =
process-node (s(s-graph (ξ)), s; trans);
s: *produce-selector* (; next-selector)

produce-selector (; next-selector) =
PASS: next-selector (s-table (ξ))

process-node (n, s; trans) =
process-value (trans (s-value(n)), s),
process-desc (s-desc (n))

process-value (v, s) =
s-graph: μ(s-graph (ξ); $\langle s$-value.s: $v\rangle$)
s-table: μ(s-table (ξ); $\langle s$: $Y\rangle$)

process-desc (list) =
length (list) $= 0 \rightarrow$ *null*
$T \rightarrow$ *process-desc* (tail (list));
set (head (list))

set (s) =
s(s-table (ξ)) $\neq \Omega \rightarrow$ *null*
$T \rightarrow$ *link* (s)

link (s) =
s-table: μ(s-table (ξ); $\langle s$:$N\rangle$).

5. An abstract linkage editor

Let us consider a programming system where the segments refer to each other only by the segments name. Then the graph walk algorithm can be applied for defining a linkage editor of this system as follows:

Let

is-r/b-program = is-segment-code-graph

and

is-select = is-segment-name.

In detail:

is-r/b-program = ({$\langle s$: is-node$\rangle$|is-segment-name (s)}),
is-node = ($\langle s$-value: is-segment-code$\rangle$, $\langle s$-desc: is-segment-name-list$\rangle$).

Let

editor (t)

be a function that maps a segment-code to an appropriate form as needed for linking. The actual mapping is not relevant here.

Then an abstract linkage editor can be characterized by the VDL-machine with the initial state:

$$\xi_0 = \mu_0(\langle s\text{-input}: p\rangle, \langle s\text{-table}: t_0\rangle, \langle s\text{-control}: \textit{walk}\ (;\ \text{next-selector, editor})\rangle)$$

where

$$\text{is-}r/b\text{-program}\ (p) = T.$$

A linkage editor model of the system in which the segments may refer to each other by entry names different from the segment name, can be defined by a generalization of the VDL-graph and the graph walk algorithm.

6. An inverse assembler model

Semantics of a class of inverse assemblers can also be defined by the graph walk algorithm.

First of all, let us define the machine code program. A machine code program is an ordered set of codes, where a code according to its function, may be an instruction or a data. That is, those programs are considered where the instructions and the data are not separated.

Definition 6.1. The set of machine code program is given by

$$\{p \mid \text{is-code-list}\ (p)\}$$

where

$$\text{is-code} = \text{is-data} \vee \text{is-}\textit{stmt}.$$

It is assumed that the program does not alter the instruction code at all and each instruction code contains the address of the next instruction explicitly that should be executed.

The instruction code part of a machine code program is called an actual program. It is assumed, that an actual program has a finite set of entries, and for any instruction at least one entry can be found where starting the program results in the execution of the instruction, that is the flow graph of an actual program is a VDL-graph:

Definition 6.2. The set of actual program is given by

$$\{t \mid \text{is-instr-graph}\ (t)\}$$

that is

$$\text{is-instr-graph} = (\{\langle s: \text{is-}\textit{stmt}\rangle \mid \text{is-select}\ (s)\}),$$
$$\text{is-}\textit{stmt} = (\langle s\text{-value}: \text{is-instr}\rangle, \langle s\text{-desc}: \text{is-select-list}\rangle)$$

where the predicate "is-*stmt*" is used instead of "is-node" in the definition of the VDL-graph.

Definition 6.3. Let

$$\text{is-code-list}\ (p) = T$$

and

$$\text{is-instr-graph}\ (t) = T.$$

The actual program t is a part of the program p if and only if

$$(\forall s,\ s(t) \neq \Omega)((\exists i)(\text{elem}\,(i)(p) = s(t)).$$

Definition 6.4. Let

$$\{a \mid \text{is-ass-instr}\,(a)\}$$

be the set of assembly form of instructions be considered. Let the function

$$\text{translator: } \{v \mid \text{is-instr}\,(v)\} \rightarrow \{a \mid \text{is-ass-instr}\,(a)\}$$

be given. Then the abstract inverse assembler is specified by the initial state of abstract machine

$$\xi_0 = \mu_0(\langle s\text{-input: } p\rangle, \langle s\text{-table: } t_0\rangle, \langle s\text{-control: } \textit{walk}\,(;\ \text{next-selector, translator})\rangle)$$

where

$$\text{is-code-list}\,(p).$$

7. Conclusions and remarks

This paper can be viewed as a contribution towards the solution of some actual problem of program synthesis. The specifications used in this paper, describe the invariable properties of states of an abstract machine rather than the input-output relationship which is expected to be realized by the desired program. The same techniques can be applied to specify programs, which are never intended to terminate.

Our example demonstrates the application of deductive techniques for deriving program that manipulates the structure of complex data structures like list and graphs.

We have concerned with some aspect of transformation rules for achieving more than one goal simultaneously by checking the protection condition of interference-free.

Abstract

Our purpose in this paper is to illustrate a deductive technique for developing abstract programs systematically from given specifications using the Vienna Definition Language.

The role and the importance of program synthesis within the scope of programming methodology is emphasized. The basic principles and the main steps of a deductive technique for deriving programs systematically from their specifications is summarized.

Programming strategies are formulated for attempting to transform the specifications into a desired VDL-program and the technique is illustrated by the example of an abstract graph walk algorithm. The example includes the definition of an abstract data graph too.

An abstract linkage editor and a general inverse assembler model are given by specifying the graph walk.

KEYWORDS: Abstract data structures, derivation of programs, program verification, program synthesis, programming methodology, Vienna Definition Language.

DEPT. OF MATHEMATICS
L. EÖTVÖS UNIVERSITY
MÚZEUM KRT. 6—8.
BUDAPEST, HUNGARY
H—1088

References

[1] DIJKSTRA, E. W., *A discipline of programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.
[2] DERSHOVITZ, N. and Z. MANNA, On automating structured programming, *Proc. IRIA Symp. Proving and Improving Programs*, Arc. et. Senans, France, July 1975, pp. 167—193.
[3] HOARE, C. A. R., An axiomatic basis of computer programming, *Comm. ACM*, v. 12, 1969, pp.576—583.
[4] HOARE, C. A. R., Parallel programming: An axiomatic approach, *Computer Languages*, v. 1, No. 2, 1975, pp. 151—160.
[5] LONDON, R. L., A view of program verification, *Proc. International Conference on Reliable Software*, April 1975, pp. 534—545.
[6] MANNA, Z., *Mathematical theory of computation*, New York, McGraw-Hill, 1974.
[7] MANNA, Z. and R. WALDINGER, The logic of computer programming, *IEEE Trans. Software Engrg.*, v. 4, 1978, pp. 199—229.
[8] MANNA, Z. and R. WALDINGER, The automatic synthesis of recursive programs, *SIGPLAN Notices*, v. 12, No. 8, 1977, pp. 29—36.
[9] MANNA, Z. and R. WALDINGER, The synthesis of structure-changing program, *Proc. 3rd International Conference on Software Engineering*, Atlanta, Georgia, USA, May 10—12, 1978, pp. 175—187.
[10] MILLS, H. D., How to write correct programs and know it, *Proc. International Conference of Reliable Software*, Los Angeles, Calif., April 1975, pp. 363—370.
[11] OWICKI, S. and D. GRIES, Verifying properties of parallel programs: An axiomatic approach, *Comm. ACM*, v. 19, 1976, pp. 279—285.
[12] RAMAMOORTHY, C. V. and S. B. F. Ho, Testing large software with automated software evaluation systems, *IEEE Trans. Software Engrg.*, v. 1, 1975, pp. 46—58.
[13] RAYMOND T. YEH (Ed.), *Current trends in programming methodology*, Vol. 2. Prentice Hall, Inc., Englewood Cliffs, New Yersey, 1977.
[14] WEGNER, P., Research directions in software technology, *Proc. 3rd International Software Engineering Conference*, Atlanta, Georgia, USA, May 10—12, 1978.
[15] WEGNER, P. (Ed.), *Research directions in software technology*, MIT Press, 1979.
[16] WEGNER, P., The Vienna Definition Language, *Comput. Surveys*, v. 4, 1972, pp. 5—63.
[17] WIRTH, N., On the composition of well-structured programs, *Comput. Surveys*, v. 6, 1974, pp. 247—259.

(Received Oct. 24, 1979)