Proceedings of the Third

Symposium on Programming Languages and Software Tools

Mati Tombak (Ed.)

Kääriku, Estonia
August 23-24 1993

Univesity of Tartu
Department of Computer Science
August 1993

Proceedings of the Third

Symposium on Programming Languages and Software Tools

Mati Tombak (Ed.)

Kääriku, Estonia
August 23-24 1993

Univesity of Tartu
Department of Computer Science
August 1993

CONTENTS

Grammars for structured documents by generalizing examples [*]

Helena Ahonen
University of Helsinki

Heikki Mannila
University of Helsinki

Erja Nikunen
Research Centre for Domestic Languages

April 1993

Abstract

Examples of structured documents include dictionaries, user manuals, etc. Structured documents have an internal organization that can be used, for instance, to help in retrieving information from the documents and in transforming documents into another form. The document structure is typically represented by a context-free or regular grammar. Many structured documents, however, lack the grammar: the structures of individual documents are known but the general structure of the document class is not available.

In this paper we describe a technique for forming the grammar describing the structure of a structured document. The user describes the structure of some example documents, and from these the system infers a small general description. The technique is based on ideas from machine learning. It forms first finite-state automata describing the examples completely. These automata are modified by considering certain context conditions; the modifications correspond to generalizing the underlaying language. Finally, the automata are converted into regular expressions, which are then used to construct the grammar.

[*]This work was partially supported by TEKES and the Academy of Finland. Authors' addresses: Helena Ahonen, Heikki Mannila, Department of Computer Science, University of Helsinki, P.O. Box 26 (Teollisuuskatu 23), SF-00014 University of Helsinki, Finland. Erja Nikunen, Research Centre for Domestic Languages, Sörnäisten rantatie 25, SF-00500 Helsinki, Finland. e-mail: `{hahonen,mannila}@cs.Helsinki.FI`, `enikunen@domlang.FI`

1 Introduction

Text with structure is quite common: dictionaries, reference manuals, and annual reports are typical examples. In recent years, research on systems for writing structured documents has been very intensive. One of the recent surveys of the field is [2]. The interest in the area has led to the creation of several document standards, of which the best known are ODA and SGML [5, 7]. The common way to describe the structure of a document is to use context-free grammars [6, 13]. Thus, in database terminology, grammars correspond to schemas, and parse trees to instances.

It is typical to use regular expressions in the right-hand sides of the productions of the grammar. For example, the following might describe the simplified structure of a dictionary entry:

$$\text{Entry} \rightarrow \text{Headword Sense}^*.$$

The meaning of this production is that an entry consists of a headword followed by zero or more senses. A more complicated example is

$$\begin{aligned}\text{Entry} \rightarrow {} & \text{Headword [Inflection]}\\ & (\text{Sense\_Number Description}\\ & \quad [\text{Parallel\_form} \mid \text{Preferred\_form}]\ \text{Example}^*)^*,\end{aligned}$$

which states that an entry consists of a headword followed by an optional inflection part and zero or more groups, each group consisting of a sense number, a description, a further optional part which is either a parallel form or a preferred form, and a sequence of zero or more examples

The structure of a document can be used to facilitate transformations and queries which have structural conditions. The structure also provides general knowledge of the text. It can be fairly complicated, however, to find the grammar that describes the structure of a given large text. (See for example [4].) The user might, for example, be experimenting with a totally new text, or the text might be already available, and the user wants to transform it into a structured form. Typically, forming the structure of an existing large text seems to be difficult without any tools.

In this paper we describe a method that can be used to form a context-free grammar for a structured text semi-automatically. The method is based on the idea that the user marks and names some example components and regions of the text using a pointing device. The marking produces example productions. However, since these productions are based on some specific parts of the text, they are overly restrictive and hence, they cannot be used as the grammar describing the structure of the text. Thus, one should be able to generalize the productions in some meaningful way.

The generalization is done by assuming that a sufficiently long common part in two productions for the same nonterminal means that also the parts following the common part should be interchangeable.

For the generalization, we use techniques from machine learning [11, 12]. Learning context-free and regular grammars from examples has been studied in, e.g.,[3, 9, 11, 14, 15]. However, these results are not directly applicable to our setting, either because they assume that positive and negative examples are available or because they make other assumptions about the data that are not valid in our case. The method we have developed proceeds as follows.

1. The example productions are transformed to a set of finite automata, one for each nonterminal. These automata accept exactly the right-hand sides of the example productions for the corresponding nonterminal.
2. Each automaton is modified in isolation, so that it accepts a larger language. This language is the smallest one that includes the original right-hand sides and has an additional property called (k, h)-contextuality. This property states roughly that in the structure of the document what can follow a certain component is completely determined by the k preceding components at the same level. Steps 1 and 2 are based on the synthesis of finite automata presented in [3, 11], specifically (k, h)-contextuality is a modification of k-reversibility [3] and k-contextuality [11].
3. The resulting automata are transformed to regular expressions, which form the right-hand sides of the productions for the corresponding nonterminals.

We have implemented our method in connection with the structured text database system HST [10]. Our preliminary empirical evidence indicates that the method is a useful tool for transforming existing texts to structured form.

The rest of this paper is organized as follows. As a running example we use entries from a Finnish dictionary [1]. Section 2 describes the construction of the initial automaton. In Section 3 we describe the general method for generalizing the productions, and the particular inductive biases, k-contextuality and (k, h)-contextuality, we use in generalizing the examples. Section 4 describes the conversion into regular expressions. Empirical results are discussed in Section 5. Section 6 contains some concluding remarks.

2 Prefix-tree automaton

The right-hand sides of productions obtained from the user's examples are represented by an automaton called a prefix-tree automaton. To construct a prefix-tree automaton we first take the set of sample productions which have the same left-hand side. The right-hand sides of these productions are added to the prefix-tree automaton one by one.

For example, if the following productions are added into a prefix-tree automaton, the result is the automaton shown in Figure 1.

Entry → Headword Inflection Sense Sense
Entry → Headword Inflection Parallel_form Sense Sense Sense
Entry → Headword Parallel_form Sense Sense
Entry → Headword Preferred_form Sense
Entry → Headword Inflection Preferred_form Sense Sense

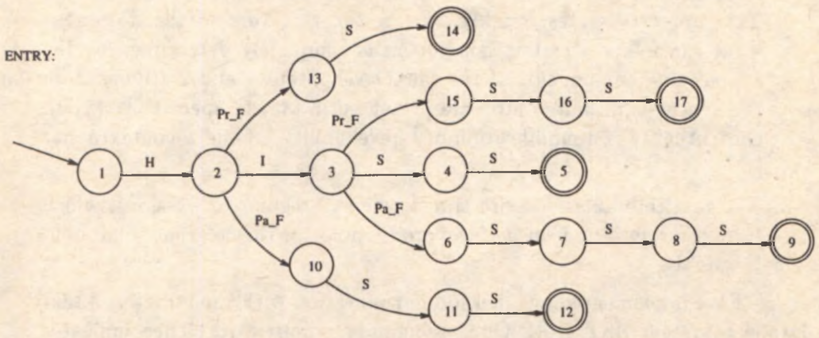


Figure 1: Prefix-tree automaton containing all the examples.

3 (k,h)-contextual languages

A prefix tree automaton accepts only the right-hand sides of the examples. To obtain useful grammars, we need some way of generalizing the examples, and the automaton describing them, in a meaningful way.

In machine learning terms, the examples of productions are all *positive examples*. That is, the user gives no examples of illegal structures. To learn from positive examples, one needs some restrictions on the allowed result of the generalization. Namely, a consistent generalization of a set of positive examples would be an automaton accepting all strings! Thus we have to define a class of automata that are allowed as results of the generalization.

By merging some of the states we get an automaton which accepts more strings, i.e., this automaton generalizes the examples. To merge states s_i and s_j we first choose one of them to represent the new state, say s_i. All the incoming arcs of s_j are then added to the set of incoming arcs of s_i , and all the outgoing arcs of s_j are added to the set of outgoing arcs of s_i. There are many possibilities of generalizing an automaton by merging states.

The generic algorithm is the following:

Algorithm 1 Generalizing a set of productions using some criterion for merging states.
Input: A criterion for merging states and a sample

$$I = \{A \rightarrow \alpha \mid A \in N,\ \alpha \in (N \cup T)^*\}$$

consisting of productions for some nonterminals.
Output: A set

$$O = \{A \rightarrow \alpha' \mid A \in N,\ \alpha' \textit{is a regular expression over the alphabet } (N \cup T)\}$$

of generalized productions such that for all $A \rightarrow \alpha \in I$ there is a production $A \rightarrow \alpha' \in O$ such that α is an instance of α'.
Method:

```
1. for each nonterminal $A$
2.     Construct a prefix-tree automaton $M_A$ from
       the productions of I with left-hand side $A$
3.     repeat
4.         for each pair $p, q$ of states of $M_A$
               if $p$ and $q$ fulfill the generalization condition
               then modify $M_A$ by merging $p$ and $q$
5.     until no more states can be merged
6.     Convert $M_A$ to an equivalent regular expression $E_A$
7.     Output the production $A \rightarrow E_A$
```

How do we choose the generalization condition? Our assumption is that the grammars used in structured documents have only limited context in the following sense. If a sufficiently long sequence of nonterminals occurs in two places in the examples, the components that can follow this sequence are independent of the position of the sequence in the document structure.

A language satisfying this condition is called *k-contextual* [11]. The property of k-contextuality can be described simply in terms of automata.

Lemma 2 A regular language L is k-contextual if and only if there is a finite automaton A such that $L = L(A)$, and for any two states p_k and q_k of A and all input symbols $a_1a_2 \ldots a_k$ we have: if there are states p_0 and q_0 of A such that $\delta(p_0, a_1a_2 \ldots a_k) = p_k$ and $\delta(q_0, a_1a_2 \ldots a_k) = q_k$, then $p_k = q_k$.

For a set of strings H, a k-contextual language L such that

1. $H \subseteq L$ and
2. for all k-contextual languages M such that $H \subseteq M$ we have $L \subseteq M$

is called a *minimal k-contextual language including H*.

It can be shown that there exists a unique minimal, i.e. the smallest, k-contextual language containing a given set of strings. If A is an automaton such that $L(A)$ is k-contextual, we say that A is a *k-contextual automaton*. Lemma 2 and Algorithm 1 give a way of constructing a k-contextual automaton which accepts the smallest k-contextual language containing $L(C)$ for an automaton C. States of C satisfying the conditions in the implication of the lemma are merged until no such states remain.

Finally the 2-contextual automaton looks like the one in Figure 2. We can see that it generalizes the examples quite well. The automaton, however, accepts only entries which have two or more *Sense* nonterminals in the end. This is overly cautious, and therefore we need a looser generalization condition. In Figure 2, for example the states s_4 and s_5 could be merged.

The intuition in using k-contextuality is that two occurrences of a sequence of components of length k implies that the subsequent components can be the same in both cases. We relax this condition and generalize the k-contextual languages further to (k, h)-contextual languages. In these languages two occurrences of a sequence of length k implies that the subsequent components are the same *already after h characters.* As for k-contextuality, we obtain an easy characterization in terms of automata.

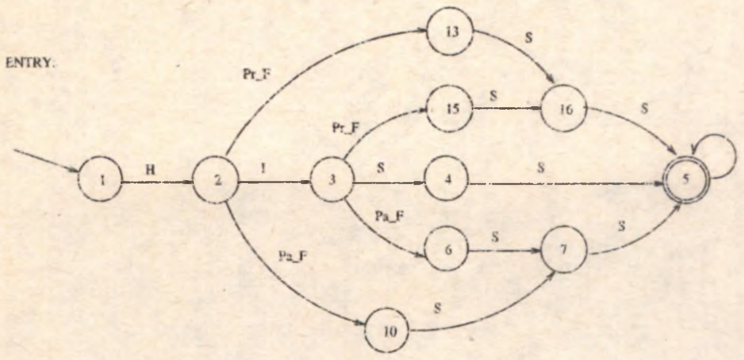


Figure 2: 2-contextual automaton.

Lemma 3 A regular language L is (k,h)-contextual if and only if there is a finite automaton A such that $L = L(A)$, and for any two states p_k and q_k of A, and all input symbols $a_1 a_2 \ldots a_k$ we have: if there are states p_0 and q_0 such that $\delta(p_0, a_1) = p_1, \delta(p_1, a_2) = p_2, \ldots, \delta(p_{k-1}, a_k) = p_k$ and $\delta(q_0, a_1) = q_1, \delta(q_1, a_2) = q_2, \ldots, \delta(q_{k-1}, a_k) = q_k$, then $p_i = q_i$, for every i, where $0 \leq h \leq i \leq k$.

The algorithm for producing the automaton that accepts a (k,h)-contextual automaton is similar to the previous algorithm: one looks for states satisfying the conditions of the above lemma, and then merges states. If similar paths of length k are found, not only the last states but also some of the respective states along the paths are merged. If $h = k$ only the last states are merged. If $h < k$ the paths have a similar prefix of length h before they are joined, i.e., $k - h + 1$ states are merged. In Figure 3 we can see the final (2,1)-contextual automaton.

4 Conversion into a regular expression

After the generalization steps presented in the previous sections have been performed, we have a collection of (k,h)-contextual automata. To obtain a useful description of the structure of the document, we still have to produce a grammar from these. An automaton can be converted into a regular

2

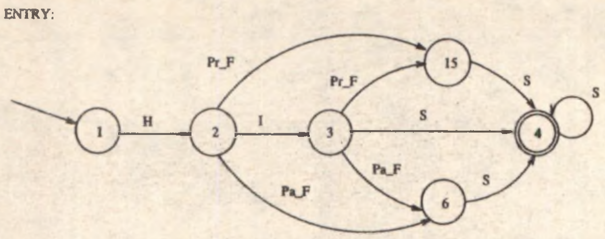


Figure 3: (2,1)-contextual automaton.

expression by using standard dynamic programming methods [8].

One of our goals was to obtain a readable grammar. The regular expressions produced by the standard method are not always so short as they could be, and therefore they have to be simplified. The simplified regular expressions form the right-hand sides of the productions for the corresponding nonterminals.

Sample productions in Section 2 generate the production:

Entry → Headword
(Inflection [Preferred_form | Parallel_form] |
Parallel_form | Preferred_form)
Sense$^+$

5 Experimental results

We have implemented the method described above in connection with the HST structured text database system [10]. We have experimented with several different document types, and the results are encouraging.

In our first test situation a user looked at some bibliographical entries, and quite mechanically marked and named all the parts of them. The program built the productions shown in Figure 4, and then generalized them. The result is shown in Figure 5.

Some remarks can be made. First, the interaction between nonterminals should be taken into account. Then the author list *Author (, Author)** would be replaced by *Authors*, and *Bpage - Epage* would be replaced by *Pages* in

```
Entry        → Key Confpaper
Pages        → Bpage - Epage
Journalpaper → Author , Author , Author , Author , Title .
               Journal , Number '(' Year ')' , Pages
Editors      → Editor and Editor eds
Confpaper    → Author , Author , Author , Author , Title .
               Booktitle , Editors , Publisher , Year , Bpage - Epage
Entry        → Key Confpaper
Entry        → Key Journalpaper
Journalpaper → Author , Author , Author , Author , Title.
               Journal , Number '(' Year ')' , Pages
Editors      → Editor and Editor eds.
Confpaper    → Author , Author , Author , Author , Title .
               Booktitle , Editors , Publisher , Year , Bpage - Epage
Authors      → Author , Author
Entry        → Key Confpaper
Confpaper    → Author , Author , Author , Title . Confname
Authors      → Author , Author , Author
Journalpaper → Authors , Title . Journal Volume
Entry        → Key Journalpaper
```

Figure 4: Sample bibliographical productions

the productions for *Confpaper* and *Journalpaper*. Second, the user sometimes gives inconsistent names, or punctuation varies in similar situations. Most of these cases can be found easily: see for instance the alternatives *eds* and *eds.* in the production for *Editors*.

Another kind of test was made with a Finnish dictionary [1]. The marking with a pointing device is inappropriate when the text considered is large and has a complicated structure. If this kind of text has been prepared for printing it is usually typographically tagged, i.e., parts of the text are circled by begin and end marks (e.g. begin bold – end bold). Since typographical means are used to make the structure clear to the reader, they can be used to make the structure explicit: tags can be changed to structural tags (e.g. begin headword – end headword).

```
Authors       → Author (, Author)*
Confpaper     → Author (, Author)* . Title .
                (Confname | Booktitle, Editors , Publisher , Year , Bpage - Epage)
Editors       → Editor and Editor (eds | eds.)
Journalpaper  → (Author (, Author)* | Authors) . Title .
                Journal (Volume | , Number '(' Year ')' , Pages)
Pages         → Bpage - Epage
Entry         → Key (Confpaper | Journalpaper)
```

Figure 5: Generalized bibliographical productions

We converted our data, which consist of 15970 dictionary entries, in the above way, removed the end tags and the text, and built the sample productions. The total number of different entry structures was about 1300 but only 82 of them covered more than 10 entries. We chose 20 of the most common structures (Fig. 6), which together covered 13313 entries. In the following the tags have been changed into whole words to facilitate understanding.

As a result we got the following production:

```
Entry → Headword
        [ Example |
        Inflection [Example | Reference] |
        [Inflection [Consonant_gradation]]
              ([Technical_field] Sense | Technical_field) [Example] |
        Reference |
        Preferred_form ]
```

This example shows that creating a grammar is not a trivial task: the structure of a dictionary entry is very flexible. The result production may look somewhat complicated itself but in any case it is a good basis for manual improvement.

If we go further with this material and take into account more entry structures, it is not feasible to produce only one production. Therefore we have studied possibilities of adding frequency information into our method (see Section 6). The goal is to separate the most common structures from the rare cases.

Entry → Headword Sense
Entry → Headword Example
Entry → Headword
Entry → Headword Inflection Sense
Entry → Headword Sense Example
Entry → Headword Inflection Sense Example
Entry → Headword Technical_field Sense
Entry → Headword Inflection Consonant_gradation Sense Example
Entry → Headword Inflection Technical_field Sense
Entry → Headword Inflection Example
Entry → Headword Inflection Consonant_gradation Sense
Entry → Headword Reference
Entry → Headword Inflection Technical_field Sense Example
Entry → Headword Technical_field Sense Example
Entry → Headword Technical_field
Entry → Headword Inflection Reference
Entry → Headword Inflection Consonant_gradation Technical_field Sense
Entry → Headword Inflection
Entry → Headword Technical_field Example
Entry → Headword Preferred_form

Figure 6: Sample dictionary productions

6 Conclusion and further work

In this paper we have presented a method for generating a context-free grammar from the user's examples. The user gives names to the parts of existing texts. These names are used to form simple productions, which are then generalized and combined to form a grammar.

In the generalization of the examples we have first applied the idea of k-contextual languages and further extended them to (k,h)-contextual languages. These conditions seem to describe quite natural constraints in text structures.

We have implemented this method and tested it with several document structures. The results are encouraging but also show some possibilities of improvement and extension. The method described here constructs only one production for every nonterminal. This is inadequate when the structure varies a lot or there are many rare or erroneous cases. It is desirable to get one or a few productions which cover most of the examples, and then several productions which correspond to the exceptions.

We have started to implement this idea in the following way. In our dictionary data each sample production has a weight which is the number of entries this production covers. When an example is added into a prefix tree automaton, all the weights of the arcs visited are increased by the weight of the new production. When the automata are generalized, the weight of a merged arc is the sum of the weights of the two arcs that are merged.

The user gives a bound b which means that the program constructs a production which covers at least all the structures that appear b times in data. In addition to this production several exception productions are constructed as well.

It would be reasonable to increase the interactivity with the user. One possibility is to apply the method incrementally: the user adds examples one by one, and the program builds a grammar. The grammar could be shown simultaneously in a different window, which makes it easier for the user to use consistent names for different structures.

If the examples are properly punctuated, it is possible to add a parser to the system. Then the user could have a large collection of existing texts. He/she could choose some examples and analyze them for the learning program and let the rest of the example texts be parsed by the program. If an example cannot be parsed, either the grammar is modified or the user changes the example. The latter gives the user a possibility to correct errors.

Generally, to be a useful tool, this method should be implemented in a flexible way. There should be a possibility of applying different kinds of document structures, both new and existing ones, easily. The user should be able to add new examples, remove old ones, and correct errors at any time. The program should also, at the user's request, offer alternative solutions.

References

[1] *Suomen kielen perussanakirja. Ensimmäinen osa (A–K).* Valtion painatuskeskus, Helsinki, 1990.

[2] J. André, R. Furuta, and V. Quint, editors. *Structured Documents.* The Cambridge Series on Electronic Publishing. Cambridge University Press, 1989.

[3] Dana Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.

[4] G. E. Blake, T. Bray, and F. Wm. Tompa. Shortening the OED: Experience with a grammar-defined database. *ACM Transations on Information Systems*, 10(3):213–232, July 1992.

[5] Heather Brown. Standards for structured documents. *The Computer Journal*, 32(6):505–514, December 1989.

[6] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing*, 1(1):19–44, 1988.

[7] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[8] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, MA, 1979.

[9] Oscar H. Ibarra and Tao Jiang. Learning regular languages from counterexamples. *Journal of Computer and System Sciences*, 43(2):299–316, 1991.

[10] Pekka Kilpeläinen, Greger Lindén, Heikki Mannila, and Erja Nikunen. A structured document database system. In Richard Furuta, editor, *EP90 – Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, The Cambridge Series on Electronic Publishing, pages 139–151. Cambridge University Press, 1990.

[11] Stephen Muggleton. *Inductive Acquisition of Expert Knowledge*. Addison Wesley, Reading, MA, 1990.

[12] Balas K. Natarajan. *Machine Learning: A Theoretical Approach*. Morgan Kaufmann Publishers, May 1991.

[13] V. Quint and I. Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 200–213. Cambridge University Press, 1986.

[14] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. In D. Haussler and L. Pitt, editors, *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 330–344, 1988.

[15] Kurt Vanlehn and William Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2(1):39–74, 1987.

ON THE COMPLEXITY OF OBJECT-ORIENTED PROGRAMS*

ÁKOS FÓTHI, JUDIT NYÉKY-GAIZLER

Dept. of General Computer Science
Eötvös Loránd University, Budapest
H-1117 Budapest, Bogdánfy u. 10/b.
E-mail: nyeky@comput.elte.hu
HUNGARY

Abstract: Object-oriented programs are constructed with the help of the same control structures as traditional ones. At first sight, therefore, their complexity can be measured the same way as the complexity of the traditional programs. In this case the complexity depends on the nesting level of the control structures, as it has been shown by Piwowarski, Harrison, Magel, Howatt, Baker etc.[HB89,HM181,HM281,PIW89]. Why do we still have the feeling that object-oriented programs are more simple than the traditional ones? To answer this, we have to introduce a new measure of complexity. The measures mentioned above have a common problem: each of them evaluates the complexity of a program only from the point of view of its control structure. Our opinion discussed here is that the complexity of a program is a sum of three components:
(1) the complexity of its control structure,
(2) the complexity of data types used,
(3) the complexity of the data handling (ie. the complexity of the connection between the control structure and the data types).

We give a suggestion for the measure of complexity of a program. This new measure of complexity is used to argue why good object-oriented programs could seem more simple.

1.Introduction

There are several methods of measuring program-complexity. The complexity of programs depends on the number of operators and operands (the software science measure); on the number of predicates (cyclomatic complexity); but these measures do not characterize sufficiently the nature of complexity, since n nested loops or n nested if statements are undoubtedly more complex than the sequence of n loops, or the sequence of n decisions. As far as we found in the literature [HB89,HM181,HM281,PIW89,McC76,Va92] the complexity of programs was so far measured only on the basis of its control structure.

Nowadays one of the most frequently read notion in the literature of programming methodology is the 'object-oriented' one. While constructing great systems the questions of reusability and extendibility became of key importance. The more simple a program is the easier it is to understand, later to modify or reuse some parts of it in the case of the construction of other, similar programs.

Followers of object-oriented methodology state that professional software production becomes notably simplified using this new technique, which results in enormous cost decrease.

Object-oriented programs contain the same control structures (sequence, if- and loop statements) as the traditional ones, thus there seems to be no difference in their complexity.

What is the greatest novelty of this design approach? It draws the attention to the importance of precise definition and consistent use of abstract data types. Actually if we inspect starting from this concept the program complexity measures so far, it will immediately strike us, that none of them takes into account neither the influence of the technique of **hiding** (e.g. use of procedures) on the complexity of programs nor the complexity of **data** used in the program, respectively the complexity of references to objects of different types or the consequences of hiding the representation and implementation of abstract data types. We also have parallelly with the control structures to examine the structure of data with the help of an appropriate measure to their complexity.

Our main proposal is, that when counting the complexity of a program, we should take the complexity of the data used and the complexity of data handling into consideration, we should see the decreasing of complexity through hiding techniques.

2. Preliminary definitions and notions

We shall define the new measure on the basis of the definitions given to the complexity of nested control structures. The definitons connected to this come from the excellently "rigorous" description of J.Howatt and A. Baker [HB89].

Definition 2.1. A *direct graph* $G = (N, E)$ consists of a set of nodes N and a set of edges E. An edge is an ordered pair of nodes (x,y). If (x,y) is an edge then node x is an *immediate predecessor* of node y and y is an *immediate successor* of node x. The set of all immediate predecessors of a node y is denoted **IP(y)** and the set of all immediate successors of a node x is denoted **IS(x)**. A node has *indegree n* if E contains exactly n edges of the form (w,z), similarly a node has *outdegree m* if E contains exactly m edges of the form (z,w).

Definition 2.2. A *path* P in a directed graph $G = (N, E)$ is a sequence of edges $(x_1, x_2), (x_2, x_3), \dots (x_{k-2}, x_{k-1}), (x_{k-1}, x_k)$, where $\forall i[1 \leq i < k] \Rightarrow (x_i, x_{i+1}) \in E$. In this case P is a *path* from x_1 to x_k.

Definition 2.3. A *flowgraph* $G = (N, E, s, t)$ is a directed graph with a finite, nonempty set of nodes N, a finite, nonempty set of edges E, $s \in N$ is the start node, $t \in N$ is the terminal node. For any flowgraph G, the s start node is the unique node with indegree zero; the t terminal node is the unique node with outdegree zero, and each node $x \in N$ lies on some path in G from s to t. Let N' denote the set $N - \{s, t\}$.

J.W. Howatt and A.L.Baker define the notion of the basic block for modeling control flow as follows:

Definition 2.4. A *basic block* is a sequential block of code with maximal length, where a sequential block of code in a source program P is a sequence of tokens in P that is executed strating only with the first token in the sequence, all the tokens in the sequence are always executed sequentially, and the sequence is always exited at the end. Namely, it doesn't contain any loops or if statements.

Definition 2.5. Every node $n \in N$ of a flowgraph $G = (N, E, s, t)$ which has outdegree greater than one is a ***predicate node***. Let Q denote the set of predicate nodes in G.

The well-known measure of McCabe (cyclomatic complexity) is based only on the number of predicates in a program:$V(G) = p + 1$. The inadequacy of the measure becomes clear, if we realize that the complexity depends basically on the nesting level of the predicate nodes. The measures proposed by Harrison and Magel [HM181,HM281] and Piwowarski [Piw82] proven to be equivalent in principle by Howatt and Baker [HB89] take this lack into account.

Definition 2.6. Given a flowgraph $G = (N, E, s, t)$, and $p, q \in N$, node p ***dominates*** node q in G if p lies on every path from s to q. Node p ***properly dominates*** node q in G if p dominates g and $p \neq q$. Let $r \in N$, node p is the ***immediate dominator*** of node q if (i) p properly dominates q and (ii)if r properly dominates q then r dominates p.

The formal definition of the scope number is based on the work of Harrison and Magel.

Definition 2.7. Given a flowgraph $G = (N, E, s, t)$, and $p, q \in N,$, the set of ***first occurence paths*** from p to q, FOP(p,q) is the set of all paths from p to q such that node q occurs exactly once on each path.

Definition 2.8. Given a flowgraph $G = (N, E, s, t)$, and nodes $p, q \in N,$, the set of nodes that are on any path in FOP(p,q) is denoted by MP(p,q):

$$MP(p,q) = \{v \mid \exists P\ [\ P \in FOP(p.q) \wedge v \in P\ ]\ \}$$

Definition 2.9. In a flowgraph $G = (N, E, s, t)$, the set of ***lower bounds*** of a predicate node $p \in N$ is

$$LB(p) = \{v \mid \forall r \forall P\ [\ r \in IS(p) \wedge P \in FOP(r,t)\ \Rightarrow\ v \in P\ ]\}$$

Definition 2.10. Given a flowgraph $G = (N, E, s, t)$, and a predicate node $p \in N$, the ***greatest lower bound*** of p in G is

$$GLB(p) = \{q \mid\ q \in LB(p) \wedge \forall r\ [\ r \in (LB(p) \setminus \{q\}) \Rightarrow r \in LB(q)]\}$$

Definition 2.11. Given a flowgraph $G = (N, E, s, t)$, and a predicate node $p \in N$, the set of nodes predicated by node p is

$$Scope(p) = \{n \mid \exists q\ [\ q \in IS(p) \wedge n \in MP(q, GLB(p))\ ]\ \} \setminus \{\ GLB(p)\ \}$$

Definition 2.12. Given a flowgraph $G = (N, E, s, t)$, the set of nodes that ***predicate*** a node $x \in N$, is

$$Pred(x) = \{p \mid\ x \in Scope(p)\}$$

Definition 2.13. The *nesting depth* of a node $x \in N$, in a flowgraph $G = (N, E, s, t)$ is

$$nd(x) = \mid Pred(x) \mid$$

Thus, the total nesting depth of a flowgraph G was counted as

$$ND(G) = \sum_{n \in N'} nd(n)$$

The measure of program complexity given by Harrison and Magel is the sum of the adjusted complexity values of the nodes. This value can be given - as proved by Howatt - as the scope number of a flowgraph:

Definition 2.14. The *scope number*, SN(G) of a flowgraph $G = (N, E, s, t)$ is

$$SN(G) = \mid N' \mid + ND(G)$$

The main concept behind this definition is, that the complexity of understanding a node depends on its nesting depth, on the number of predicates dominating it.

This measure was proved by J.W. Howatt and A.L.Baker to be equivalent to the ones proposed by Piwowarski or Dunsmore and Gannon, that is why we shall refer to this in the following.

3. Proposal for a new measure

As we can see from the above, the software complexity measures did not so far take the role of procedures into consideration, while the complexity of data used was completely out of the question.

Our first suggestion is directed towards the introduction of the notion of **procedure** . The complexity of programs, decomposed to suitable procedures, is decreasing. We need a measure which expresses this observation.

Let us represent a program consisting of procedures not with a flowgraph, but with the help of a **set of flowgraphs**. Let us define the complexity of a program as the sum of the complexities of its component flowgraphs!

Definition 3.1. A *programgraph* $\mathcal{P} = \{G \mid G = (N, E, s, t) \ flowgraph\}$ is a set of flowgraphs, in which each start node is labeled with the name of the flowgraph. These labels are unique. There is a marked flowgraph in the set, called the 'main' flowgraph, and there is at least one flowgraph in the set which contains a reference to each label except the 'main' one.

Definition 3.2. The *complexity of a programgraph* will be measured by the sum of the scope numbers of its subgraphs

$$\mathcal{C}(\mathcal{P}) = \sum_{G \in \mathcal{P}} SN(G)$$

This definition shall reflect properly our experience that if we e.g. take a component out of the graph which does **not** contain a predicate node to form a procedure - i.e. a basic block, or a part of it (this means a single node), then we increase the complexity of the whole program according to our definition. This is a direct consequence of the fact that in our measures so far we contracted the statement-sequences what is reasonable according to this view of complexity. If we create procedures from sequences the program becomes more difficult to follow, since we can not read the program linearly, we have to "jump" from the procedures back and forth. The reason for this is that a sequence of statements can always be wieved as a single transformation. This could of course be refined by counting the different transformations being of different weight, but this approach would transgress the competence of the model used. The model mirrors these considerations since if we form a procedure from a subgraph containing no predicate nodes, then the complexity increases according to the complexity of the new procedure subgraph, i.e. by 1.

On the other hand, if the procedure does contain predicate node(s), then by the modularization we decrease the complexity of the whole program depend ing from the nesting level of the outlifted procedure. If we take a procedure out of the flowgraph, creating a new subgraph out of it, the measure of its complexity becomes independent from its nesting level. On the place of the call we may consider it as an elementary statement (as a basic block, or part of it).

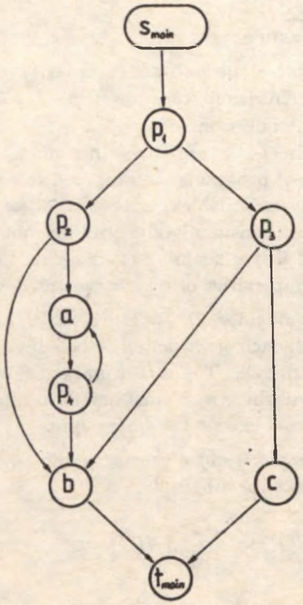


Fig.1.

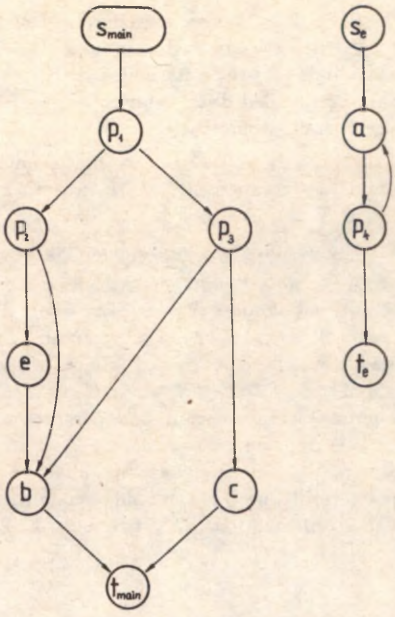


Fig.2.

See Fig. 1. and Fig. 2. as an example. It is visible, that even in such a simple case the complexity of the whole program decreases if we take an embedded part of the program out as a procedure. One can simply control that the complexity of the program shown on Fig.1. $SN(G) = 19$, while the complexity of the second version shown on Fig. 2. $\mathcal{C}(\mathcal{P}) = \sum_{G\in\mathcal{P}} SN(G) = 18$.

This model reflects well the experience of programmers, that the complexity of a program can be decreased by the help of modularization not only when the procedure is called from several points of the program, but a well developped procedure alone, in the case of a single call can decrease the complexity of the whole program.

It is also trivial, that if we form a procedure from the whole program, than we also increase the complexity.

Now we are reaching the point where it is inevitable, not only from the point of wiev of handling procedure calls but also in connection with the whole program, to deal with the question of data. The complexity of a program depends not only on the complexity of the transformation but also on the subject of this transformation. What are the data to be processed.

We extend the definitions that we have used so far: Let the set of nodes of our flowgraphs be widened by a new kind of node to denote the data! Let us denote by a small triangle ($\triangle$) the data nodes in the program! Let us draw to these nodes special edges, called **data reference edge**, which surely return to their origin from each node, where there is a reference to that data!

Definition 3.3. Let N and D be two finite, nonempty sets of control structure and data nodes respectively. A **data reference edge** is a triple (x_1,x_2,x_1) where $x_1 \in N$ and $x_2 \in D$.

Let us redefine the notion of a flowgraph as follows:

Definition 3.4. A *data-flowgraph* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ is a directed graph with a finite, nonempty set of nodes $\mathcal{N} = N \bigcup D$, where N represents the nodes belonging to the control structure of the program and D represents the nodes belonging to the data used in the program, (both of them are nonempty), with a finite, nonempty set of edges $\mathcal{E} = E \bigcup R$, where E represents the edges belonging to the control structure of the program, and R represents the set of its **data reference edges**. $s \in N$ is the start node, $t \in N$ is the terminal node. The s start node is always the unique node with indegree zero for all the data-flowgraphs $\mathcal{G}$,; the t terminal node is the unique node with outdegree zero, and each node $x \in \mathcal{N}$ lies on some path in $\mathcal{G}$ from s to t. Let $\mathcal{N}'$ denote the set $\mathcal{N} - \{s,t\}$.

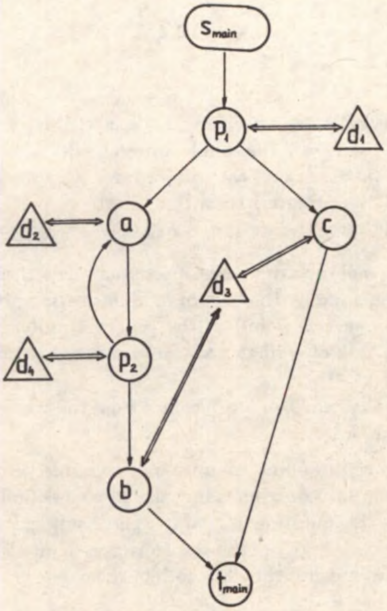


Fig.3.

The complexity of the program will be computed from the set of graphs obtained this way in accordance with the previous definitions - depending from the number of nodes and predicates dominating them. We call the attention to the fact, that if we take the role of data in the program into consideration, then the number of those nodes, which have outdegree greater than one, increases, and we have to determine the Scope also for those nodes, where there is a reference to a data.

As an example let us have a look at the program represented by the graph on Fig.3. The complexity counted this way can be obtained:
$Scope(p_1) = \{d_1, a, d_2, p_2, d_4, c, d_3, b\}$
$Scope(p_2) = \{a, d_2, p_2, d_4\}$
$Scope(a) = \{d_2\}$
$Scope(b) = \{d_3\}$
$Scope(c) = \{d_3\}$
$Pred(p_1) = 0$
$Pred(p_2) = \{p_1, p_2\}$
$Pred(a) = \{p_1, p_2\}$
$Pred(b) = \{p_1\}$
$Pred(c) = \{p_1\}$
$Pred(d_1) = \{p_1\}$
$Pred(d_2) = \{p_1, p_2, a\}$
$Pred(d_3) = \{p_1, b, c\}$
$Pred(d_4) = \{p_1, p_2\}$
Thus $ND(\mathcal{G}) = 15$ and $SN(\mathcal{G}) = 24$.

This way the complexity will also be influenced by the data and this is just as well as at the transformations, since that to what extent a data makes a program more complicated is determined by the decisions preceding the reference to it. This graph and its complexity measure defined this way express that the complexity of a program depends also on the data used, and on the references to these data.

As we have seen so far the complexity may be decreased the by the appropriate modularization of the program. Similarly, if we take out a subgraph which contains one or more data with all of the data reference edges leading to this data, we will decrease the complexity. E.g. if there is a single reference to a data at some transformation, and we take this transformation in order to create a procedure, where this data will be a **local** variable - the complexity of program decreases. The substantial moment in this activity is, that we **hide** a variable from the view of the whole program, we make it invisible (local), and thus essentially decrease the additive factor to the complexity at this point.

As an example see Fig. 4. constructed from the graph shown on Fig.3.. As one can easily control, the complexity of this program will be 18 opposed to the value 24 obtained for the program on Fig. 3.

The occurences are of course, in general, not so simple because there can be several references to the same data. How could we decrease the complexity of the program in addition to this? One fundamental tool is the decreasing of

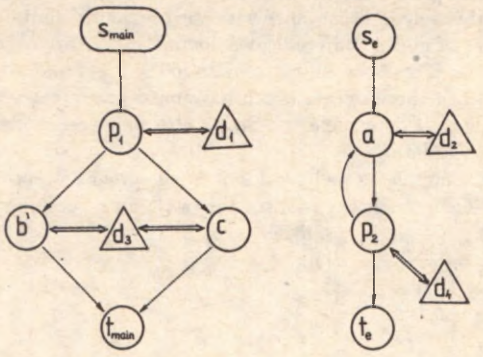


Fig.4.

the number of small triangles, the number of data used. One possibility for this is that we draw certain data into one structure, creating data structures from our data. E.g. if we have to work with a complex number, then we decrease the complexity of the progrma if instead of storing its real and imaginary part separetely in the variables a and b , we draw these to a complex number x which has operations treating the real and imaginary part appropriately. The reduction(the decreasing of data nodes) occurs of course only when we **hide the components** in the following from the outerworld, since if we do not this, this would mean, on the level of the program graph, that we did not merge two data nodes into one, but created a third one to the previous two.

As a matter of fact we can decrease the complexity of program in connection with data if and only if we build abstract data types **hiding the representation**. In this case the references to data elements will always be references to data since a data can only be handled through its operations. While computing the complexity of the whole program we have to take into account not only the decreasing of the complexity, but also the increasing by the added complexity of determined by the implementation of the abstract data type. Nevertheless this will only be an additiv factor instead of the previous nested factor.

That is the most important complexity-decreasing consequence of the object oriented view of programming: **the object hides the type** from the predicates (decisions) supervising the use of the object.

The complexity measure studied here expresses the structural complexity of the program.

The notion of inheritance allows actually to **hide a class of types**, further decreasing the sum of complexity, of course adding the complexity of the inheritance graph. To compute the complexity of an inheritance graph we have to use the graphrepresentation suggested by Meyer [Me88], namely using edges from the descendants to their ancestors, since the complexity of a class depends on their ancestor(s), not on their descendant(s). The complexity of an object-oriented program will thus be determined by the sum of the complexity of the inheritance graph and the complexity of objects used.

Conclusions

We investigated the given complexity measures, and found them suffering from a common problem, that they, while computing the complexity of a given program, did not take the role of neither the modularization nor the data used into account. On the basis of the previous efforts of J.W.Howatt and A.L.Baker we suggested a new measure of program complexity, which reflects our psychological feeling that the main concepts of object-oriented programming methodology help us to decrease the total complexity of a program.

References:

[Dij76] Dijkstra,E.W.: A Discipline of Programming, Prentice-Hall, Engelewood Cliffs, N.Y.,1976.

[FN91] Fóthi,Á. and Nyéky-Gaizler,J. : A Theoretical Approach of Objects and Types, in: Kai Koskimies and Kari-Jouko Raiha (eds.): Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala, Finland, August 21-23,1991, Report A-1991-5,August,1991.

[HM181] Harrison,W.A. and Magel,K.I. : A Complexity Measure Based on Nesting Level, ACM Sigplan Notices,16(3),63-74 (1981).

[HM281] Harrison,W.A. and Magel,K.I. : A Topological Analysis of the Complexity of Computer Programs with Less Than Three Binary Branches, ACM Sigplan Notices,16(4), 51-63 (1981).

[HB89] Howatt,J.W. and Baker,A.L. : Rigorous Definition and Analysis of Program Complexity Measures : An Example Using Nesting, The Journal of Systems and Sofware 10,139-150 (1989).

[McC76] McCabe, T.J. A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4),308-320 (1976).

[Me88] Meyer,B. : Object-Oriented Software Construction, Prentice Hall, New York, 1988

[Piw82] Piwowarski,P. : A Nesting Level Complexity Measure, ACM Sigplan Notices ,17(9),44-50 (1982).

[Va92] Varga, L.: A new approach to defining software design complexity. In: R.Mittermeier (ed.): Shifting Paradigms in Software Engineering. Springer Verlag, Wien, New York, 198-204.(1992)

* Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr 2045

THE WEAKEST PRECONDITION AND THE SPECIFICATION OF PARALLEL PROGRAMS*

ZOLTÁN HORVÁTH

Dept. of General Computer Science
Eötvös Loránd University, Budapest
H-1117 Budapest, Bogdánfy u. 10/b.
E-mail: hz@comput.elte.hu
HUNGARY

Abstract

We propose a method to express safety and progress properties of parallel programs based on the well-known concept of the weakest precondition [Dij76, FH91] and related predicate transformers.

We give new definitions for the operations of Unity [CHM88,Kna92], i.e. for unless, ensures and leads-to. Postulating fairness conditions [Mor90] we investigate the relationship of the old and new operations to the commonly used operations of linear and branching time temporal logics [ES88] and to the concept of the weakest and strongest invariant [Lam90].

Introduction

We take the specification as the starting point for program design. We are looking for a model of programming which supports the top-down refinement of specifications [Var81, FH91, CHM88]. The proof of the correctness of the solution is developed parallel to the refinement of the specification of the problem. However we do not aspire to synthetise programs automatically [Lav78, ESS88/4.1.3] or to verify ready algorithms [ESS88/4.2]. In the present paper we are especially interested in building tools for specification of parallel programs.

The UNITY model [CHM88] of programing seeems to be an appropriate choice. We describe the main concepts of UNITY in section 2. We give a short overwiew of semantic models and temporal logics in section 3. Three basic operators are used for specification of parallel programs in UNITY, i.e.: unless, ensures and leads-to. We propose new definitions for the operators based on the well-known concept of the weakest precondition [Dij76, FH91] and related predicate transformes in section 4. We justify the correctness of the new definitions and investigate the relationship of the old and the new operations to the commonly used operations of linear and branding time temporal logics [ES88]. We show, that the new logic is more expressive than the old one.

A similiar approach to define progress properties is taken by Lukkien and Snepscheut in [LS92]. They give a new definition for leads-to for a language dealing with sequential composition but in absence of parallelism and fairness.

* Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. 2045

1. Preliminary notions and definitions

In the following we use the terminology used in [Par79, Fót83, Fót88, Hor90, FH91], (To avoid confusion, we use the word statement instead of program, and effect relation instead of program function.) $R^n(A)$ denotes the set of n-ary relations on A, otherwise relation means binary relation in the following.

Def. 1.1. The relation $R \subseteq A \times B$ is a *function*, if $\forall a \in A : |R(a)| = 1$.

Def. 1.2. $f \subseteq A \times \mathcal{L}$ is a logical function, if it is a function, where $\mathcal{L} ::= \{\uparrow, \downarrow\}$.

Remark: We use the words predicate and condition as synonyms for logical function. If P and Q are logical functions, then we use the $\wedge, \vee, \rightarrow$ operations for function composition on the usual way.

Def. 1.3. $TS[f] ::= \{a \in A | f(a) = \{\uparrow\}\}$ is called the truth-set of the logical function f. The operations $\cup, \cap, \subseteq$ correspond to the function conmpositons $\wedge, \vee, \rightarrow$.

Def. 1.4. $I \subset \mathcal{N}$. $\forall i_j \in I : A_{i_j}$ is a finite or numerable set. The set $A ::= \underset{i_j \in I}{\times} A_{i_j}$ is called state space, the sets A_{i_j} are called *type value sets* .

Def. 1.5. The elements of the state space, the points $a = (a_{i_1}, ..., a_{i_n}) \in A$, are called *states* .

We can imagine a statement (a sequential program) as a relation, which associates a sequence of the points of the state space to the points of the state space.

Def. 1.6. The relation S is called a **statement**, if

i) $S \subseteq A \times A^{**}$,

ii) $\mathcal{D}_S = A$,

iii) $(a \in A \wedge \alpha \in S(a)) \Rightarrow \alpha_1 = a$,

iv) $(\alpha \in \mathcal{R}_S \wedge \alpha \in A^*) \Rightarrow (\forall i \, (1 \leq i < |\alpha|) : \alpha_i \neq \alpha_{i+1})$,

v) $(\alpha \in \mathcal{R}_S \wedge \alpha \in A^\infty) \Rightarrow$
$$(\forall i \in N \, (\alpha_i = \alpha_{i+1} \rightarrow (\forall k (k > 0) : \alpha_i = \alpha_{i+k}))).$$

where A^* is the set of the finite sequences of the points of the state space, and A^∞ the set of the infinite ones. Let $A^{**} = A^* \cup A^\infty$.

Def. 1.7. The **effect relation** of the statement S is the relation $p(S) \subseteq A \times A$, if

i) $\mathcal{D}_{p(S)} = \{a \in A \mid S(a) \subseteq A^*\}$

ii) $\forall a \in \mathcal{D}_{p(S)} : p(S)(a) = \{b \in A \mid \exists \alpha \in S(a) : \tau(\alpha) = b\}$,

where $\tau : A^* \rightarrow A$ is a function, which associates its last element to the sequence $\alpha = (\alpha_1, ..., \alpha_n)$, i.e. $\tau(\alpha) = \alpha_n$.

Def. 1.8. The statement $wp(S, R)$ is called the **weakest precondition** of the postcondition R in respect of the statement S , if $TS[wp(S, R)] = \{a \in \mathcal{D}_{p(S)} \mid p(S)(a) \subseteq TS[R]\}$

Def. 1.9. A statement over the state space A is called *empty* and denoted by $SKIP$, if $\forall a \in A : \; SKIP(a) = \{(a)\}$.

Def. 1.10. Let $A = A_1 \times ... \times A_n$, $F = (F_1, ..., F_n)$, where $F_i \subseteq A \times A_i$. The statement $S \subseteq A \times A^{**}$ is a *general assignment* , if

$S \ = \ \{(a, red(a,b)) \mid a,b \in A \wedge a \in \bigcap_{i\in[1,n]} \mathcal{D}_{F_i} \wedge b \in F(a)\} \cup$

$\{(a, (aaa....)) \mid a \in A \wedge a \notin \bigcap_{i\in[1,n]} \mathcal{D}_{F_i}\}$

Remark: $\mathcal{D}_F = \bigcap_{i\in[1,n]} \mathcal{D}_{F_i}, F(a) = F_1(a) \times F_2(a) \times ... \times F_n(a)$

The assignment S is denoted by $a := F(a)$ and called deterministic, if $(\forall a \in A : |p(S)(a)| = 1)$. If we use the notation $(a := F(a),\text{ if } \pi)$, then $((\mathcal{D}_S = A) \wedge (\forall a \in TS[\neg\pi] : p(S)(a) = \{a\}))$. This kind of assignment is called *conditional.*

Def. 1.11. A function $F : R^n(A) \rightarrow R^m(B)$ is monotone if $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$.

As is well known every monotone function has a minimal (least) and a maximal (greatest) fixpoint.

Lemma 1.12. Let function $F : R^n(A) \rightarrow R^m(B)$ be monotone.

a) The minimal fixpoint of F: $\mu F = \bigcap\{X | F(X) \subseteq X\}$,
b) fixpoint induction for minimal fixpoint: if $F(Z) \subseteq Z$ then $\mu F \subseteq Z$,
c) $F(\mu F) = \mu F$,
d) the maximal fixpoint of: $\eta F = \bigcup\{X | X \subseteq F(X)\}$,
e) fixpoint induction for maximal fixpoint: if $Z \subseteq F(Z)$ then $Z \subseteq \eta F$,
f) $F(\eta F) = \eta F$.

2. The main concepts of UNITY

The first specification of the problem is short, only the most important aspects are formulated at the begining. The specification and its solution, the abstract program is independent of architectures, of scheduling and of programing language. The implementation of the abstract program is defined by the help of standard methods, so called mappings.

The structure of the abstract program should not imply to encode unnecessary synchronisation points between the processes. CSP [Hoa78] or Ada like programs are built up from sequential components which define explicit control flow over large subsets of statements. Therefore the abstract program is regarded as a set of deterministic (simultaneous) conditional assignments (cf. Def. 1.10.). The condotions of the assignments encode the necesary synchronisation restrictions explicitly. In each step of executon of the abstract program some assignment is selected nondeterministically and executed. Every statement is executed infinitely often, i.e. an unconditionally fair scheduling is postulated. If the condition of an assignment is false, then the effect is equivalent to SKIP. If more than one processor selects statements for execution, then the executions of different processors are fairly interleaved.

The abstract program terminates never. A fixed point said to be reached in a state, if any statement in that state leaves the state unchanged.

3. Semantic models

If we want to reason about absence of undesirable side-effects, then we have to use rich mathematical model which is appropriate to reflect all the symptoms caused by the interaction of processes. We have to deal with synchronous and asynchronous, distributed and shared-memory architectures. Events an different processors take place simultaneously, processes on the same physical processor may interfer with each other. We would like to incorporate into our model the concept of true parallelism and true nondeterminism [BW91, MV91]. On the other hand we have to choose a model which complexity can be managed.

By the help of different semantics we formalize the meaning of our abstract programs. A semantics is said to be more abstract, if it regards more syntactically different abstract programs to be equivalent. The method used for the definition of equivalence may be denotational, operational or axiomatic, etc. In denotational semantics elements of domains, i.e. a set of mathematical entities, are associated to abstract programs. The function from the set of abstract programs to the domain is compositional, i.e. the element of the domain associated to a compoud program is defined in terms of the elements associated to the component programs. Operational semantics is often based on labelled transition systems reflecting the behavior of processes. In the axiomatic semantics equivalence is expressed by a set of axioms and dervation rules. The same abstraction level can be achived using any of the three style of semantic definition [Hen88]. The advantage of denotational semantics is the ability to reason about the correctness of programs on a static way, i.e. by comparison of the elements of the semantic domain.

A semantics is true parallel, if it do not identifies a parallel program with set of the interleavings of its elementary components, i.e. $(a \parallel b \neq ab + ba)$.

We are speaking about linear time semantics, if the nondeterministic behaviour of the program is restricted to the initial states of its execution. Branching time semantics reflects the true nondeterminism of programs, i.e. $(a(b + c) \neq ab + ac)$.

We give two examples for semantic models:

Model 3.1.1.: The semantic domain is a set of binary relations which associate a sequence of the points of the state space to the points of the state space [Fót88] A sequence denotes one possible execution of the statement (program). This is a linear time denotational semantics. Operations over the domain are defined as compositions of relations [Fót83,Hor90].

Model 3.1.2.: Sequences, built up from ordered pairs are associated to initial states in the semantic model of UNITY. The ordered pair consist of the state and the label of the program component (i.e. the label of the assignment), which is selected for execution at this state. Labels are important to identify the model and/or the process, which is responsible for the given state transition [Bes83, CHM88]. This is an interleaved, linear time semantics, which is relatively easy to use from mathematical point of view. The linear time nature of the semantics is reflected in the definitions of operators used in specifications. The concept of unless, ensures and leads-to is based on the concepts of linear time temporal logics [ES88].

3.2. Temporal logics

3.2.1 Branching time temporal logic

Using branching time temporal logics for describing properties of nondeterministic programs we can associate to a program a directed tree. A node of the tree corresponds to a point of the state space, an edge represents a state transition. Labels associated to edges identify program components, which are responsible for the state transition. Assertions can be formulated to characterize the nodes and the paths [ES88]. We denote a node by e, a path by t.

$AP(e)$, if for all path t leading from node $e : P(t)$,
$EP(e)$, if exists a path t leading from node $e : P(t)$,
$GP(t)$, if for every point e of path $t : P(e)$,
$FP(t)$, if exists a point e of path $t : P(e)$.

We denote by an abstract program given in UNITY by S. We are interested in the case, when the paths corresponding to nonfair execution sequences are excluded. In this case we use the operators A_ϕ and E_ϕ:

$\phi = \forall s \in S : GF$ exec(s), where exec(s) holds at a node e, if the edge leading to e is labelled by S.
$A_\phi P = A(\phi \rightarrow P)$
$E_\phi P = E(\phi \wedge P)$

Several notion of fairness can be formulated using branching time logic [ES88]. We may associate guards (boolean expressions) to atomic actions. An atomic action is enabled to be selected for execution, if its guard is true. *enabl*(s) holds at a node, if in the state corresponding to the node the guard associated to s is true.

- if ($\forall s \in S : GF$ exec(s)), then the scheduling is unconditionally fair.
- if ($\forall s \in S : FG$ enabl(s) $\rightarrow GF$ exec(s)), then the scheduling is weakly fair.
- if ($\forall s \in S : GF$ enabl(s) $\rightarrow GF$ exec(s)), then the scheduling is strongly fair.

We define some commonly used operators of branching time temporal logic:

$\Box P ::= AG(P)$ (always)
$\Diamond P ::= \neg\Box\neg P = EF$ (not never, potentially)
$\sim P ::= AF(P)$ (eventually, ineviability)

3.2.2 Linear time temporal logic

The nondeterministic behaviour of programs is restricted for the initial state of the execution in the case of linear time temporal logics. The program is represented as we have seen in Model 3.1.1. or 3.1.2. We are not allowed to use operators like A or E. We can characterize one single execution sequence of the program by the help of the G, F and related operators. A specification given in linear time temporal logic satisfied by a nondeterministic program, if it is satisfied by every possible execution sequence of the program.

The linear time specification P can be translated to branching time logic as AP. Since the operator A is not distributive [ES88], the whole specification has to be translated at once. The well-none example for the faliure of distribution of A is: $A(FP \vee G\neg P) \not\equiv (AFP \vee AG\neg P)$.

The operators $\Box$, $\sim$, $\Diamond$, are used in linear time temporal logic too. The first two operators are defined to correspond to their branching time version (by the translation method).

$\Box P ::= G(P)$ (always)
$\Diamond P ::= \neg\Box\neg P =\sim P ::= F(P)$ (eventually, ineviability)

Remark: Potentiality can not be expressed using linear time temporal logic.

An other way to define the semantics of a temporal logic is the use of Kripke structures [Krö87] or to refer the execution sequences explicitly:

- $(P \text{ atnext } Q)(t_i) ::= ((\forall j > i : \neg Q(t_i)) \vee (Q(t_k) \wedge P(t_k) \wedge \forall j \in (i,k) : \neg Q(t_j)))$.
- $P\, \mathcal{U}_w\, Q ::= Q \text{ atnext } (P \rightarrow Q)$ (weak until).

4. The original and the new definition of the operators: unless, ensures, leads-to

4.1. The original definitions

The operators used in [CHM88] for the specification of UNITY programs were based on the linear time semantics of the abstract programs. Hoare triples ($\{P\}s\{Q\}$) were used as short hand form: t denotes a sequence representing a possible program execution (cf. 3.2.), $p[t_i]$ holds, if P holds at the appropriate state. Ror program S and $s \in S : \{P\}S\{Q\}$ holds, if for all sequences and for all $i \geq 0 : (P[t_i] \wedge t_i.label = s) \Rightarrow q[t_{i+1}]$.

Def.4.1.1. (unless)
p unless $q \equiv (\forall s : s \in S :: \{p \wedge \neg q\}s\{p \vee q\})$

Remark: P unless $Q = (\Box(P \Rightarrow (P \bigcup_w Q)))$ [Sin91].

Def. 4.1.2. P stabil ::= P unless $\downarrow$.

Remark: The weakest and strongest invariant of S [La90] is closely related to unless. $win(S,R)$ is the weakest stabil (in respect of S) condition, which implies R. $sin(S,R)$ is the strongest stabil condition, which is implied by R.
Let denote FP the condition which truth set is the set of fixed points of S.
$winp(S,R) ::= win(S,(FP \rightarrow R))$
$sinp(S,R) ::= sin(S,(FP \rightarrow R))$

Progress properties:

Def. 4.1.3. (ensures)
p ensures $q ::= (P$ unless $q) \wedge \exists s \in S :: \{p \wedge \neg q\}s\{q\}$ [CHM88]

Remark: the definition of ensures based on the unconditional scheduling.

Def. 4.1.4. (p leads-to q, inevitability)
A given program has the property p leads-to q if and only if this property can be derived by a finite number of applications of the following inference rules:
p ensures $q \Rightarrow p$ leads-to q,
(transitivity) p leads-to q, q leads-to $r \;\Rightarrow\; p$ leads-to r,
(disjunction) For any set W,
$[\forall m : m \in W :: p(m)$ leads-to $q] \;\Rightarrow\; [\exists m : m \in W :: p(m)]$ leads-to q.

Remark: Since leads-to is defined by inference rules, the statement not (P leads-to Q) has the meaning that P leads-to Q can not be derived. We are not able to express using leads-to, that Q is not inevitable from P.

Remark: If (P leads-to Q), then ($\Box(P \rightsquigarrow Q)$) [Sin91].

4.2. The redefinition of unless

We use in our definitions the concept of the weakest precondition, fixpont of functions, least and greatest fixpoints of equations, and fixpont induction as we defined them in section 1.

We generalize the concept of UNITY program on the following way. Let denote with I a nonempty, finite subset of the natural numbers. Let be S a nonempty, finite set of conditional assignments, such that

$S = \{s_i \mid i \in I \ \wedge \ \mathcal{D}_{p(s_i)} = A \ \wedge \forall a \in A : (|s_i(a)| < \omega)\}$

From now on we use the branching time semantic model 3.2.1., which reflects the definition of the weakest precondition.

Def. 4.2.1. Let S be an abstract program. $wp(S, R) ::= \forall s \in S : wp(s, R)$.

Def. 4.2.2. Let S be an abstract program. $wpa(S, R) ::= \exists s \in S : wp(s, R)$ (the angelic weakest precondition [Mor90]).

Def. 4.2.3 (unless)

$P \triangleright Q ::= (P \wedge \neg Q \rightarrow wp(S, P \vee Q))$.

Lemma 4.2.1. $\triangleright$ is the branching time version of UNITY unless. Proof left to the reader.

Remark: The presence of fairness does not affect the definition of $\triangleright$.

4.3. The new definition of progress properties

The new definition of leads-to (4.3.5.) is based on Park's paper [Par79]. Park recognised, that the weakest precondition of iterative programs can not be expressed by the help of the minimal or by the help of the maximal fixpoint in the case of presence of fairness. He used the combination of the least and greatest fixpoint operator to define a fixpoint different from both. By the help of this he determined the set of sequences given by the fair merge of sequence X and sequence Y.

The weakest precondition of recursive procedures are given by the help of predicate transformers and fixpoint oprators in case of presence of fairness in [Mor90]. We use the set-theoretic approach of [Park79] and investigate iterative program structures instead of recursive procedures.

Def.4.3.1. $G(P, Y, X) = P \vee ((wpa(S, Y) \wedge wp(S, X))$

Lemma 4.3.1. G is monotone in P, Y, X.

Proof: Since wp and wpa are monotone, 4.3.1 can be proved by predicate calculus. □

Consequense: $\forall P, Y : \eta X : G(P, Y, X)$ exists.

Def.4.3.2. $F(P, Y) = \eta X : G(P, Y, X)$

Lemma 4.3.2. $F(P, Y)$ is monotone in P, Y.

Proof: Let suppose $Y \rightarrow R$ and $P \rightarrow S$.

$F(P, Y) = G(P, Y, F(P, Y))$ { fixpoint (cf. 1.12/e) }.

$G(P, Y, F(P, Y)) \rightarrow G(S, R, F(P, Y))$ {G monotone }.

By fixpoint induction 1.12/f: $F(P, Y) \rightarrow \eta X : G(S, R, X)$,

$\eta X : G(S, R, X) = F(S, R)$, by definition. □

Def.4.3.3. eventually P (inevitability)

$\sim P = \mu Y : F(P, Y)$

Lemma 4.3.3: $\sim P$ is monotone in P.

Proof: Suppose, that $P \to Q$. $F(P, \sim Q) \to F(Q, \sim Q)$ {F is monotone } $F(Q, \sim Q) = \sim Q$ { fixpoint 1.12/c}.
By fixpoint induction: $\mu Y : F(P, A) \to (\sim Q)$ { by def. } $[\sim P \to \sim Q]$ □

Lemma 4.3.4.: $P \to (\sim P)$.

Proof: $P \vee (wpa(S, \sim P) \wedge wp(S, \sim P)) = \sim P$ { fixpoint }. By strenghtening the left hand side: $P \to (\sim P)$ □

Collorary 4.3.5.: By 4.3.4 and 4.3.12. $(\sim P) \leftrightarrow (\sim (\sim P))$ □

Def.4.3.5.: (ensures)

$(Q \mapsto P) ::= (Q \to (P \vee ((wpa(S, P) \wedge wp(S, Q)))$
$i.e.(Q \mapsto P) ::= (Q \to G(P, P, Q))$

Lemma: $\mapsto$ is the branching time version of UNITY ensures.
Proof left to the reader.

Def.4.3.6.: (leads-to)

$(Q \hookrightarrow P) ::= (Q \to (\sim P))$

Lemma 4.3.6.: If $Q \mapsto P$, then $Q \hookrightarrow P$.

Proof: $[Q \mapsto P] = [Q \to G(P, P, Q)]$ { def }. By fixpoint induction 1.12./e:
(if $[Q \to G(P, P, Q)]$ then $[Q \to \eta X : G(P, P, X))]$ i.e.
$Q \to F(P, P)$. $F(P, P) \to F(P, \sim P)$ {$P \to (\sim P)$ (Lemma4.), F is monotone }.
$F(P, \sim P) = (\sim P)$ { fixpoint }. i.e. $Q \to (\sim P)$ { by def.} $Q \hookrightarrow P$ □

Lemma 4.3.7.: If $P \hookrightarrow Q$ and $Q \hookrightarrow R$, then $P \hookrightarrow R$.

Proof: $[Q \hookrightarrow R] = [Q \to (\sim R)]$ { def }.
Since $\sim$ is monotone, $[(\sim Q) \to (\sim (\sim R))]$. By the application of Collorary 5. $[(\sim Q) \to (\sim R)]$. On the other hand by definition: $[P \to (\sim Q)]$. By transitivity of implication: $[P \to (\sim R)]$ □

Lemma 4.3.8.: Let be I an arbitrary set. If $\forall i \in I : (P_i \hookrightarrow Q)$ then $(\exists i : P_i) \hookrightarrow Q$.

Proof: $[\forall i \in I : (P_i \hookrightarrow Q)] =$ { definition }
$[\forall i \in I : P_i \to (\sim Q)] \Rightarrow [P_{i_1} \vee \ldots \vee P_i \to (\sim Q)] = [(\exists i : P_i) \to (\sim Q)] \Rightarrow$ { def }
$[(\exists i \in P_i) \hookrightarrow Q]$ □

Collorary: By Lemmas 6., 7., 8.:
If P leads-to Q, then $P \hookrightarrow Q$ □.

Def.4.3.8.:

$$FF^0(P) = \downarrow$$

$$FF^i(P) = \begin{cases} F(P, FF^{i-1}(P)), & \text{if } i \text{ is not a limit ordinal} \\ \bigvee_{j=0..i} FF^j(P), & \text{if } i \text{ is a limit ordinal.} \end{cases}$$

Lemma 4.3.10.:

a) $(i \leq j) \rightarrow [FF(P)^i \rightarrow FF(P)^j]$

b) $(\mu Y : F(P,Y)) = FF(P)^i$ for some i i.e. $(\sim P) = FF(P)^i$ for some i.

c) $FF(P)^i \rightarrow wpa(S, FF(P)^{i-1}) \vee P$

d) $FF(P)^i \rightarrow wp(S, F(P)^i) \vee P$

Proof:

Since F is monotone, the proof of a) and b) is a special case of Lemma 3. of [Mor90].

Proof of c) and d): $FF(P)^i = \{$ def $\} = F(P, FF(P)^{i-1}) = \{$ def $\} = \eta X : G(P, FF(P)^{i-1}, X) = \{$ def $\} = \eta X : (P \vee ((wpa(S, FF(P)^{i-1}) \wedge wp(S, X))) = \{$fixpoint$\} = (P \vee ((wpa(S, FF(P)^{i-1})) \wedge wp(S, FF(P)^i))) = (P \vee ((wpa(S, FF(P)^{i-1})) \wedge (P \vee wp(S, FF(P)^i)))$ i.e. by weakening by right hand side $FF(P)^i \Rightarrow P \vee wpa(S, FF(P)^{i-1}))$ and $F(P)^i \Rightarrow P \vee wp(S, FF(P)^i))$ □

Theorem 4.3.11. If $(\sim P)$ holds for $a \in A$, the scheduling is unconditionally fair and the UNITY program S is in the state a, then S inevitable reach a state, for which P holds (i.e. $A_\phi F(P)$).

Proof: The proof is similiar to the proof given in [Mor90]. Let choose a sequence t starting from the truth set of $(\sim P)$. We show, if P never holds along t, then the selection of the statements of t is not fair. Let associate an ordinal $i > 0$ to the point t_j of t, such that $FF(P)^i(t_j)$ and $\neg FF(P)^{i-1}(t_j)$. By Lemma 10/d and by the definition of $\sim P$ such an i exists and it is not an limit ordinal. Since the sequence of associated ordinals is monotonically decreasing (Lemma 10/d), there exists a k and h, such that: $\forall j > h : FF(P)^k(t_j) \wedge \neg FF(P)^{k-1}(t_j)$. By lemma 10/c the sequence is not fair, i.e. never selects the existing direction s for which $wp(s, FF(P)^{k-1})$. □

Collorary 4.3.12. $(\sim (\sim P)) \rightarrow (\sim P)$ □

Conclusion

By the help of the new definitions we increased the effectiveness of the specification language. The old properties of the operators remain valid. Finally we show an example for expressing potentiality:

$x :: \mathbb{Z}$.

$P \equiv (x > 5) \quad Q \equiv (x < 15) \quad R \equiv (x = 5)$

$R \mapsto ((\neg Q \mapsto Q) \vee (P\ stabil))) \wedge (\neg(R \hookrightarrow Q) \wedge \neg(R \hookrightarrow P))$

INIT ::= $(x = 5)$.

$$\left\{ \begin{matrix} x := x + 1, \text{ ha } x \geq 5 \ ; \\ x := x - 1, \text{ ha } x \leq 5 \end{matrix} \right\}$$

Remark: The specification corresponds to $E_\phi GP \wedge E_\phi FQ \wedge A_\phi(GP \vee FQ)$ [ES88].

References

[Bes83] Best,E.: Relational Semantics of Concurrent Programs, in *Formal Desc. of Programming Concepts II.*, 1983.

[BW91] de Bakker,J.W.-Warmerdam,J.H.A.: Four domains for concurrency, Theoretical Computer Science, Vol. 90 (1991), 127-149.

[CHM88] Chandy,K.M.-Misra,J.: *Parallel program design: a foundation*, Addison-Wesley, 1988, (1989).

[Dij76] Dijkstra,E.W.: *A Discipline of Programming*, Prentice-Hall, 1976.

[ES88] Emerson,E.A.-Srinivasan,J.: Branching Time Temporal Logic, in *Linear Time, Branching Time and Partial Order in Logics an Models for Concurrency, LNCS 354*. Springer-Verlag 1989. 123.-172

[Fót83] Fóthi Á.: *Introduction into Programming (Bevezetés a programozáshoz)*, ELTE TTK egyetemi jegyzet, Budapest, 1983.

[Fót88] Fóthi Á.: A Mathematical Approach to Programming, Annales Uni. Sci. Budapest. de R. Eötvös Nom. Sectio Computatorica, Tom. IX. (1988), 105-114.

[FH91] Fóthi Á.- Horváth Z.: The Weakest Precondition and the Theorem of the Specification, in *Proceedings of the Second Symposium on Programming Languages and Software Tools*, Pirkkala, Finland, August 21-23, 1991,ed.: Kai Koskimies and Kari-Jouko Räihä, Uni. of Tampere, Dep. of Comp. Sci. Report A-1991-5, August, 1991.

[Hen88] Hennessy,M.: *Algebraic Theory of Processes*, The MIT Press, 1988.

[Hoa78] Hoare,C.A.R.: Communicating Sequential Processes, CACM 8/21. 1978.

[Hor90] Horváth Z.: Fundamental relation operations in the mathematical models of programming, Annales Uni. Sci. Budapest. de R. Eötvös Nom. Sectio Computatorica, Tom. X. (1990), 277-298.

[Kna92] Knapp, E.: Derivation of concurrent programs: two examples, Sci. of Comp. Progr. Vol. 19., 1-23., (Oct. 1992)

[Krö87] Kröger, F.: *Temporal Logic of Programs*, Spriger-Verlag, 1987.

[La90] Lamport,L. : win and sin Predicate Transformers for Concurrency, ATPL, Vol.12. No.3. July 1990, 396-428

[Lav78] Laventhal,M.: *Synthesis of Synchronization Code for Data Abstractions*, Ph.D. Thesis, MIT, 1978

[LS92] Lukkien, J., van de Snepscheut J.,L.,A.: Weakest Preconditions for Progress, Formal Aspects of Computing (1992) 4: 195-236.

[Mor90] Morris,J.,M.: *Temporal Predicate Transformers and Fair Termination* Acta Informatica Vol. 26, 287-313, 1990.

[MV91] Mak,R.H.-Verhoeff,T.: Classification of Models, Lecture Notes on Process Models, TU Eindhoven, manuscript, 1991.

[Par79] Park, D.: *On the semantics of fair parallelism* In LNCS 86, pp 504-526. Springer 1980.

[Sin91] Singh, A.,K.: Specification of concurrent objects using auxiliary variables, Science of Computer Programming 16, 49-88, 1991.

[Var81] Varga L.: *Programok analízise és szintézise*, Akadémiai Kiadó, Bp., 1981.

Interactive Diagnosis and Testing of Logic Programs

Tamás Horváth (e-mail: h158hor@ella.hu)
Tibor Gyimóthy (e-mail: h42gyi@ella.hu)
Zoltán Alexin (e-mail: h157ale@ella.hu)
Ferenc Kocsis (e-mail: h155koc@ella.hu)

Research Group on the Theory of Automata
Hungarian Academy of Sciences, H-6720 Szeged
Aradi vértanúk tere 1., Hungary

Abstract: In this paper a method (called IDT) is presented which combines Shapiro's Interactive Diagnosis Algorithm with the Category Partition Testing Method. This method can be used both in the debugging and in the testing of Prolog programs. The basic idea of IDT is that the test database prepared by the Category Partition Testing can be used to reduce the amount of user interaction during the debugging. In the IDT method the diagnosis process effects the testing hence IDT can be considered as an integrated debugging and testing method.

Keywords: logic programs, category partition testing method, algorithmic debugging

1 Introduction

The automatic program debugging technique introduced by Shapiro [SHA83] can isolate an erroneous procedure, given a program and an input on which it behaves incorrectly. Shapiro's model has been applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output, and nontermination. A major drawback of this debugging method is the great number of queries to the user for the correctness of intermediate results of procedure calls. Thus, an important improvement would be to minimize the amount of information the user needs to supply for algorithm to diagnose the error.

An algorithmic debugging method GADT (Generalized Algorithmic Debugging and Testing) for imperative languages is discussed in [FRI91]. A major improvement in bug-localization process is demonstrated in GADT by combining the Category Partition Testing Method (CPM) [OST88] with the algorithm introduced in [SHA83]. The main concept of the improvement is the following: during the debugging of a program the user has to answer many "difficult" questions. If this program has been already tested by CPM, the test results for the procedures of the program can be used in the debugging process.

Of course, during the testing the programs cannot be tested with all possible properties and values of the input parameters. Hence, the first task of the tester in CPM is to define the critical properties of parameters (called *categories*) of the procedures. These categories can be divided into classes (called *choices*) presuming that the behavior of the elements of one choice is identical from the point of view of the test process. From the specification of choices *test frames* are generated. A test frame contains exactly one choice from each category. The behavior of a test frame from the point of view of the testing process can be represented by an arbitrary element of it (called *test case*). Running the given procedures on the

* This work is supported by OTKA-501 and ESPRIT BRA 6020.

corresponding test cases a *test database* can be created. An item of this test database contains the identifier of the test frame, the concrete input parameters (test case), the output values and the evaluation result of this test frame (*yes/no/undefined*).

In the debugging process the concrete values of the parameters of a procedure are given. By determining the corresponding test frame to a given input the test database can be checked with the selected frame. In the case of 'good' tests report the debugger skips to the next procedure without query.

In this paper we present a method (called *IDT*) which combines Shapiro's Interactive Diagnosis Algorithm with CPM. This method can be used both in the debugging and testing of Prolog programs. The basic idea of the IDT system is similar to GADT e.g., by using the test database prepared by the CPM the number of user interaction can be reduced during the debugging. Shapiro's diagnosis algorithm presented in [SHA83] traverses the refutation tree of a program and asks the user about the expected behavior of each clause (which can be considered as a procedure). The user has to give yes or no answer and the algorithm isolates an error inside a certain procedure body.

Shapiro suggests that the number of queries can be reduced by storing the answers to previous queries. However, the probability that the same query for a procedure is used several times during the debugging is very small. In the IDT method we also store the queries, but each query represents a (equivalence) class of the possible inputs. These classes are defined during the CPM testing process. A query can be represented by a triple <p, x, y>, where p is a procedure and x and y vectors over some domain D, such that p on x returns y. For a query <p, x, y> a test frame t is identified on the basis of the CPM test specification of procedure p. This identification can be done automatically by using predefined *searching functions* in the test specification for the inputs and classes. When these searching functions are not given new queries are generated to the user to determine the correct test frame. Usually it is much easier to answer these queries than to the original ones. After the test frame t is identified to a query <p, x, y> the system is looking for a stored query <p, x', y'> which represents the test frame t. If the procedure p was previously tested by such query <p, x', y'> then the result of this query is used to answer the query <p, x, y>. Of course sometimes the assumption that queries <p, x', y'>, <p, x, y> give the same result may be incorrect. The 'visible' effect of such a false assumption may be that a bug could not be localized using IDT method. In this case we use Shapiro's original algorithm to identify the incorrect procedure. Assume that the query <p, x, y> was used to isolate the incorrect procedure p and this procedure was tested with the query <p, x', y'>. Suppose that <p, x, y> and <p, x', y'> belong to the same test frame t and the result of the query <p, x', y'> is *yes* but the result of the query <p, x, y> is *no*. In this case the partition specification for the procedure p during CPM testing was not correct. Therefore the IDT method prepares a new partition for the inputs of the procedure p. In this new partition the inputs x and x' represent different test frames. Hence, the IDT method can be considered as an improvement of CPM algorithm presented in [OST88].

The main differences between the GADT and the IDT methods are that the GADT has been applied to imperative languages while IDT to Prolog and in the IDT the diagnosis process effects testing. Hence IDT can be considered as an integrated debugger and testing method.

In the rest of this paper we firsts give a brief overview of Shapiro's algorithm in Section 2. Section 3 contains a formal description of the Category Partition Method. In Section 4 we describe the IDT method and an example for this method is presented in Section 5. Finally in Section 6 the conclusion and some remarks for future works are described.

2 Shapiro's single-stepping diagnosis algorithm

In this section we give a short overview of an algorithm presented in [SHA83] and recall definitions related to this algorithm. Shapiro's single-stepping algorithm can isolate an erroneous procedure (clause), given a program and an input on which it behaves incorrectly. This algorithm traverses the refutation tree of a program and asks the user about the expected behavior of each clause. The user has to give a *yes* or *no* answer and the bug inside a certain procedure is identified.

2.1 Definition: A *logic program* P is a finite set of definite clauses (the clause $A \leftarrow B_1, \ldots, B_n$ is definite iff all B's are atoms, $n \geq \theta$).

2.2 Definition: Let C denote the clause $A \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$. Then *head*(C) denotes A and *body*(C) is the set $\{B_1, \ldots, B_n\}$.

2.3 Definition: Let P be a logic program, M an interpretation of P, A' a ground atom and $A \leftarrow B_1, \ldots, B_n$ an arbitrary clause in P. We say that $A \leftarrow B_1, \ldots, B_n$ *covers* A' in M iff there is a substitution θ such that $A\theta = A'$ and for all i $(1 \leq i \leq n)$ $B_i\theta \in M$.

2.4 Definition: An arbitrary clause $p \in P$ is *correct* in M iff all ground atoms covered by p are in M. Otherwise we say that p is incorrect in M.

2.5 Definition: Let p be an arbitrary clause in P that terminates on some input x and returns y as output. Then the *top level trace* of the triple <p, x, y> is a finite (possibly empty) ordered set $\{<p_1, x_1, y_1>, <p_2, x_2, y_2>, \ldots, <p_n, x_n, y_n>\}$. Where p on input x calls first p_1 with input x_1, that returns y_1 as output then p_2 with x_2, ... , and so on. Finally p call p_n on input x_n which returns y_n and p returns y.

2.6 Definition: A *partial computation tree* of P is an ordered tree. Every node in this tree is labeled with some triple <q, u, v>. The set of the direct descendants of an inner node is a legal top level trace of this node. A T tree is a *complete computation tree* of P if it is a partial computation tree and all leaves in T are empty sets.

In the following we suppose that the program P is free of side-effects. Let p be an arbitrary clause in P that terminates on input x and returns y as output such that $<p, x, y> \notin M$. It means that P has at least one incorrect clause. Then for finding the incorrect clause that causes the error the computation tree rooted by <p, x, y> is traversed in a postorder manner. During the traversing of the tree at each <q, u, v> node a membership question is issued. Let us suppose that the first *false* answer is received at the node <q, u, v>. Let the direct descendants of <q, u, v> be $<q_1, u_1, v_1>, \ldots, <q_m, u_m, v_m>$. Since we used postorder strategy for all i $(1 \leq i \leq m)$ it holds that $<q_i, u_i, v_i> \in M$. From this it follows that the clause $q \leftarrow q_1, \ldots, q_m$ covers the triple <q, u, v> which is not in M. The algorithm stops at the node <q, u, v> and returns the clause instance $<q, u, v> \leftarrow <q_1, u_1, v_1>, \ldots, <q_m, u_m, v_m>$. By this method the erroneous clause can always be identified assuming that the answers to the membership questions are correct.

A query-optimal modified version of this method is called divide-and-query. We demonstrate the behavior of the single-stepping method through a small example [SHA83].

2.7 Example: a simple Prolog program for sorting a list.

```
insert(X,[Y|U],[Y|V])   :- Y < X , insert(X,U,V).
insert(X,[Y|U],[X,Y|U]) :- X ≤ Y.
insert(X,[],[X]).

sort([],[])   :- .
sort([X|Y],Z) :- sort(Y,U),insert(X,U,Z).
```

The above Prolog program contains two procedures (*insert and sort*). The procedure *insert* puts a new element into the list — given as the second argument — and returns the result in the third argument. The procedure *sort* sorts the tail of the input list and then inserts the head of the input list into this list.

This program works as follows:

?: sort([-2,-4,3,-6,2,-3],X).

X = [-6,-4,-3,-2,2,3].

For the purposes to demonstrate Shapiro's algorithm let us make a bug in the above program: change in the first line the call **insert(X,U,V)** to **insert(Y,U,V)**. It is a case of simple mistyping.

```
insert(X,[Y|U],[Y|V]) :- Y < X , insert(Y,U,V).
```

Calling the sort with the argument [6,-2,4] it returns X = [-2,-2,4] as a result which is obviously wrong. Let us try to find the wrong procedure using the single-stepping algorithm:

This algorithm works as follows: recursively interprets the call structure of Prolog. It performs the Prolog evaluation from bottom up and each step it asks the user whether the newly determined goal is *true* or *false*. The call tree looks as follows:

```
sort([6,-2,4],[-2,-2,4])
        sort([-2,4],[-2,4])
                sort([4],[4])
                        sort([],[])
                        insert(4,[],[4])
                insert(-2,[4],[-2,4])
                        -2 ≤ 4
        insert(6,[-2,4],[-2,-2,4])
                6 > -2
                insert(-2,[4],[-2,4])
                        -2 ≤ 4
```

The questions, the user should answer are the following:

Query: Is it ok?	*sort([],[])*	yes
Query: Is it ok?	*insert(4,[],[4])*	yes
Query: Is it ok?	*sort([4],[4])*	yes
Query: Is it ok?	*insert(-2,[4],[-2,4])*	yes
Query: Is it ok?	*sort([-2,4],[-2,4])*	yes
Query: Is it ok?	*insert(-2,[4],[-2,4])*	yes
Query: Is it ok?	*insert(6,[-2,4],[-2,-2,4])*	no

After seven questions the false procedure call has been identified. The wrong clause instance is:
insert(6,[-2,4],[-2,-2,4]) ← -2 < 6, insert(-2,[-2,4],[-2,-2,4]).

3. Category Partition Testing Method

An informal description of the Category Partition Testing Method can be found in [OST88]. In this section we give a formalization of this functional testing method. During the process of functional testing, the programs (procedures) cannot be tested with all possible properties of the input parameters. Hence, the tester's first task is to define the critical properties of parameters. These critical properties - called categories - are investigated in the testing process.

The categories can be divided into classes - called choices - presuming that the behavior of the elements of one choice is identical from the point of view of the testing.

If the categories and choices for a program have been defined, then all the possible test frames can be generated. A test frame contains exactly one choice from each category.

In general, there are many superfluous frames among the generated test frames. These frames can be eliminated by associating selector expressions with the choices. A choice can be made in a test frame if the selector expression associated with the choice is true. The selector expressions contain property names. A property name is also associated with a choice and can be considered as a logical variable. The value of this variable is true if the given frame contains that choice. In example 3.1 we give simple CPM specification for the clause 'insert' and the generated test frames.

3.1 Example: The category-partition specification for 'insert':

```
Test specification: insert
Category: number_of_elements

    Choice: d0: {(X,[]) | X ∈ D}
        property zero

    Choice: d1: {(X,[Y]) | X, Y ∈ D}

    Choice: d2: {(X,[Y|Z]) | X, Y ∈ D, Z ∈ D+}

Category: first_element

    Choice: e0: {(X,[]) | X ∈ D}
        if zero

    Choice: e1: {(X,[Y|Z]) | X, Y ∈ D, Z ∈ D* and X ≤ Y}
        if not zero

    Choice: e2: {(X,[Y|Z]) | X, Y ∈ D, Z ∈ D* and X > Y}
        if not zero

End of specification
```

The generated test frames are: $\{(d_0,e_0), (d_1,e_1), (d_1,e_2), (d_2,e_1), (d_2,e_2)\}$

In the following we give a short formal description of the Category Partition Method in logic programming environment. Let $\mathbf{P} = \{p_1, \dots , p_n\}$ denote a logic program, where p_i $(1 \le i \le n)$ is a clause (procedure) in the program. We assume, that the domain of the interpretation is the set D. Then every clause p in **P** can be considered as a mapping from D^n into D^m, where n and m are the input and the output arities of p, respectively.

3.1 Definition: Let p an element of **P** and n the input arity of p. An equivalence relation over D^n is called a *category* of p.

3.2 Definition: Let p an element of **P** and c a category of p. An equivalence class of c is called *choice of c*.

Every choice of a category determines a class of D^n. In the CPM the user can define a choice using an expression (searching function) that depends on a subset of the input variables. By these expressions the choice an arbitrary element of D^n belongs to can be automatically determined. We assign to every procedure a finite set of categories and each category can be considered as a finite set of choices.

3.3 Definition: Let p be an arbitrary clause in **P** and let $\Sigma_p = \{\sigma_1, \dots, \sigma_{pk}\}$ denote the set of the categories of the clause p where $\sigma_i = \{c_1, \dots, c_{in}\}$ (c's are the choices). Σ_p is a *test specification of the clause* p. A *test specification of the program* **P** is the set $\Sigma_{\mathbf{P}} = \{ \Sigma_p \mid p \in \mathbf{P}\}$.

3.4 Definition: Let p be a clause in **P**, and $\Sigma_p = \{\sigma_1, \dots, \sigma_{pk}\}$ a test specification of p. Let $F(\Sigma_p)$ denote a subset of the Cartesian-product $\sigma_1 \times \dots \times \sigma_{pk}$. $F(\Sigma_p)$ is called the *set of test frames* of p and we suppose that $F(\Sigma_p)$ covers the total D^n. The set $F(\Sigma_p)$ is a classification of D^n therefore $F(\Sigma_p)$ can be considered as an equivalence relation over D^n. An element in $F(\Sigma_p)$ is called a *test frame* of p. The set of test frames of the program **P** is the set $F(\Sigma_{\mathbf{P}}) = \{ F(\Sigma_p) \mid p \in \mathbf{P} \}$.

3.5 Definition: Let p be a clause in **P**, $\Sigma_p = \{\sigma_1, \dots, \sigma_{pk}\}$ a test specification of p, and $f = (t_1, \dots, t_{pk})$ a test frame of p. A test case of f is an element $(d_1, \dots, d_{pk})$, where d_i is in the class (choice) t_i $(1 \le i \le p_k)$. (Of course if f denotes an empty subset of D^n then such a test case does not exist.)

A test frame can be considered as an element of the products of equivalence relations over D^n. The Category Partition Method checks the behavior of p for a given test frame by investigating the result of p on a representative element (test case) of this frame. We assume that the set of test frames is partially evaluated i.e. there are test frames which are not evaluated yet (this is the usual situation in practice). Therefore we have to order an evaluation function to the set of the frames which maps the test frames into the set {*true*, *false*, *undefined*}. The T_GEN [SZU91] system based on the category partition method works as follows: the user has to define for every procedure a set of categories. Every category has to be an equivalence relation. The user can mark the important test frames, which he wants to test. Then the system generates test cases automatically to these test frames, and calls the procedure with the test cases as input. The user as the ground oracle has to decide the truth of an output. The test frame, its test case, the output values and the evaluation is stored into a database. The evaluation of a test case is a function, see below. We assume, that every test frame has at most one test case.

3.6 Definition: Let p be a procedure of **P**, and Σ_p a test specification of p. Then a *CPM-testing of p* by Σ_p is a mapping φ_p: $F(\Sigma_p) \rightarrow$ {*true*, *false*, *undefined*}. Let f be an arbitrary test frame of p, and t a test case of f. Then $\varphi_p(f)$ is *true/false* iff we applied p to t and the output was *true/false* by the ground oracle. Otherwise $\varphi_p(f)$ is *undefined*. (if f denotes an empty set then $\varphi_p(f)$ = *true*).

3.7 Definition: Let f be an arbitrary test frame of p over Σ_p. We say, that the CPM-testing of f is *consistent* iff $\varphi_p(f)$ is independent of the choice of its test case.

3.8 Definition: Let p be an arbitrary procedure in **P** and Σ_p a test specification of p. The CPM-testing of p is *consistent* iff it is consistent for every test frame of p over Σ_p. The CPM-testing of **P** is *consistent* iff every p in **P** is consistent.

3.9 Definition: Let p an arbitrary procedure (clause) in **P** and let Σ_p an arbitrary test specification of **P**. Let $\chi_p : D^n \rightarrow F(\Sigma_p)$ mapping, such that $\chi_p(x) = f$ iff f is the corresponding test frame of x. From the previous definitions it follows that χ_p is a function. (i.e. χ_p is the *searching function*.)

3.10 Definition: A *CPM test configuration* of the program **P** over the set of test frames $F(\Sigma_p)$ is a finite set of five-tuples $T(F(\Sigma_p)) = \{ (p, f, i, o, e) \mid p \in \mathbf{P}, f \in F(\Sigma_p)$, i is a test case of f, o is the output of clause p on input i and $e \in \{$*true, false, undefined*$\}$ is the evaluation of the test frame f (e.g. $e = \varphi_p(f)$) $\}$. A CPM test configuration can be considered as a test database. For a clause $p \in \mathbf{P}$ the CPM test configuration of the clause p contains all elements of $T(F(\Sigma_p))$ such that the first component is p.

3.11. Definition: Let σ be an equivalence relation over D^n and let x be an arbitrary element of D^n. Then $\sigma[x]$ denotes the class of σ containing element x. Let σ_1 and σ_2 be two equivalence relations over D^n. We say that $\sigma_2 \geq \sigma_1$ iff from $\sigma_2[a] = \sigma_2[b]$ it follows that $\sigma_1[a] = \sigma_1[b]$ for any $a, b \in D^n$. Let $F(\Sigma_p)$ and $F(\Sigma'_p)$ be two sets of test frames of an arbitrary clause p in **P**. We say that the CPM test configuration of clause p $T(F(\Sigma'_p))$ is a *refinement of $T(F(\Sigma p))$* iff $F(\Sigma'_p) \geq F(\Sigma_p)$ (i.e. $\forall \pi \in F(\Sigma_p)\ \exists \rho \in F(\Sigma'_p)$ such that $\rho \geq \pi$). For a program **P** a CPM test configuration of **P** $T(F(\Sigma'_p))$ is a *refinement of $T(F(\Sigma_p))$* iff $T(F(\Sigma'_p))$ is a refinement of $T(F(\Sigma_p))$ for all $p \in \mathbf{P}$.

If T_1 and T_2 are two CPM test configurations of the same program **P** and T_2 is a refinement of T_1 then the cardinality of the inconsistent test frames of T_2 is less or equal than of T_1.

4 The Interactive Diagnosis and Testing Algorithm

In Section 2 the single-stepping method for algorithmic debugging of logic programs has been briefly summarized. This algorithm can be applied only for those clauses that terminate on some input and result in an incorrect output. For these inputs the algorithm traverses the computation tree rooted by <p, x, y> in postorder manner. At each node it generates a membership question. It stops at the first node <q, u, v> which is not in the interpretation M. The number of the queries depends on the position of the incorrect clause instance inside the computation tree. In order to reduce the number of the queries an extended method called divide-and-query has been proposed. We present our algorithm as a modification of the single-stepping method but it can be used with the divide-and-query method as well.

We assume that there exists a CPM test configuration for the program P i.e. for all $p \in P$ a set of test frames is defined and each test frame may have an evaluation. In the process of algorithmic debugging we use the assumption that the given CPM test configuration is consistent. From this assumption it follows that if a test frame is already evaluated then this evaluation is independent of the test case chosen. This is only a hypothesis of the user which may be false. Therefore we should take into account that there are inconsistent test frames in the CPM test configuration of **P**. If a referred test frame is not evaluated (i.e. φ is *undefined* for this test frame) we automatically generate a membership question.

Let $p \in \mathbf{P}$ be a clause and let us suppose that $<p, x, y> \notin M$ for some x and y and the complete computation tree rooted by <p, x, y> is finite. Similarly to the single-stepping method our algorithm walks this tree in postorder manner. Let us suppose that we have to answer the query <q, u, v> is in M or not, where <q, u, v> is a node in the tree. We examine whether the test frame belonging to q and u is already evaluated (i.e. $\varphi_q(\chi_q(u))$ is not *undefined*). If this test frame has already been evaluated then the result of the evaluation is used to answer the membership query.

Since our hypothesis for the consistency of the test frames is not proved therefore the result given by our algorithm should be verified. On one hand it is possible that we will not find the false clause. On the other hand, if a false clause has been identified by the IDT algorithm it may occur that the test frame belonging to the head of the clause is inconsistent or in the body of the clause instance there is an atom whose test frame is inconsistent. This latter case means that the clause we found is not the deepest incorrect clause in the tree. For resolving this inconsistency the original single-stepping method is invoked. We store all questions and answers in a test database (CPM test configuration) therefore the number of queries the user has to answer is less or equal (in the worst case) than in the original single-stepping method. In the case of consistent test frames the number of queries may be significantly decreased. If we find an inconsistent test frame during the debugging then the user has to modify the CPM test configuration. More exactly the user has to define a refinement for the CPM test configuration such that the new configuration will be consistent regarding the known test cases.

4.1 Algorithm for scanning the computation tree by IDT:

Input: <p, x, y> not in M,
Output: q false clause instance and f Boolean.

```
procedure IDT_Debug (<p,x,y>,q,f) ;
begin
  let T := { $<p_1,x_1,y_1>$, $<p_2,x_2,y_2>$, ... , $<p_n,x_n,y_n>$}
                     be the top level trace of <p,x,y>.
  f := false ;
  i := 1 ;
  while (i $\leq$ |T|) and (not f) do
  begin
    IDT_Debug ( $<p_i,x_i,y_i>$,q,f ) ;
    inc(i) ;
  end ;
  if not f then
  begin
    if $\varphi_p(\chi_p(x))$ = undefined then ans := Query (<p,x,y>) ;
    f := $\varphi_p(\chi_p(x))$ ;
    if not f then
    begin
      f := true ;
      q := <p,x,y> $\leftarrow$ $<p_1,x_1,y_1>$,$<p_2,x_2,y_2>$,...,$<p_n,x_n,y_n>$;
    end ;
  end ;
end ;
```

Let us suppose that <p, x, y> $\notin$ M for some p $\in$ **P**, x $\in$ D^n and y $\in$ D^m. The Algorithm 4.1 gets the triple <p, x, y> as an input and returns in f a Boolean value which shows whether the procedure found the incorrect clause instance. If this value is *true* then q will contain this false clause instance. Let us suppose that {$<p_1, x_1, y_1>$, $<p_2, x_2, y_2>$,..., $<p_n, x_n, y_n>$} is the top level trace of <p, x, y>. The algorithm recursively calls itself on the elements of the top level trace until it finds a false clause (i.e. f = *false*). If for all i
($1 \leq i \leq n$) the evaluation of the test frame belonging to $<p_i, x_i, y_i>$ is *true* then let the test frame belonging to <p, x, y> be examined. If this test frame is not evaluated yet (i.e. $\varphi_p(\chi_p(x))$ = *undefined*) then the `Query(<p,x,y>)` is asked (see later). Otherwise (i.e. $\varphi_p(\chi_p(x)) \neq$ *undefined*) the membership question is not printed out but it is answered with the evaluation of the test frame. If $\varphi_p(\chi_p(x))$ = *false* then the algorithm stops and returns the corresponding instance of the clause p in variable q.

4.2 Algorithm for finding false procedure by IDT:

Input: <p, x, y> not in M,
Output: r false clause instance.

```
procedure Main_IDT_Debug (<p,x,y>,r) ;
begin
  IDT_Debug (<p,x,y>,r,f) ;
  if f then
  begin
    let r = <q,u,v> ← <q_1,u_1,v_1>,<q_2,u_2,v_2>,...,<q_m,u_m,v_m>
    ans := Query (<q,u,v>) ;
    if ans = false then
    begin
      ans := Query_Top_Level_Trace (r);
      if ans = true then
        return (r)
      else
      begin
        Modified_Single_Step (<q,u,v>,r,f);
        return (r);
      end ;
    end ;
  end ;
  Modified_Single_Step (<p,x,y>,r,f);
  return (r) ;
end ;
```

Let us suppose that the Algorithm 4.1 has found the clause instance <q, u, v> ← <q_1, u_1, v_1>, <q_2, u_2, v_2> ,..., <q_m, u_m, v_m>. Then we should examine the following cases.

1. A membership question for <q, u, v> should be asked. If this concrete question has already occurred then the result is retrieved from the test database. For this purpose the function Used_Test_Case is invoked. If *true* was answered then the test frame belonging to p and x is inconsistent so a call of Modified_Single_Step is necessary.

2. If in 1 there was no inconsistency then for all elements of the top level trace of <q, u, v> the Query(<q_i, u_i, v_i>) should be issued. If all elements in the top level trace are in the interpretation M then the false clause instance is <q, u, v> ← <q_1, u_1, v_1>, <q_2, u_2, v_2> ,..., <q_m, u_m, v_m>. Otherwise if there is an element <q_i, u_i, v_i> in the top level trace whose test frame is inconsistent then the Modified_Single_Step should be called on the triple <q, u, v> since we know that the error is in the computation tree rooted by <q, u, v>.

If the algorithm did not find a false clause then there is at least one node (the root surely) in the computation tree of <p, x, y> whose test frame is inconsistent.

The above Algorithm 4.2 handles these cases.

4.3 Utilities for finding a false procedure by IDT:

Input: clause instance <q, u, v> ← <q_1, u_1, v_1>, <q_2, u_2, v_2> ,..., <q_m, u_m, v_m>,
Output: a Boolean value.

```
function Query_Top_Level_Trace (c) : boolean ;
begin
  Let c := <q,u,v> ← <q_1,u_1,v_1>,<q_2,u_2,v_2>,...,<q_m,u_m,v_m>;
  f := true ;
  i := 1 ;
```

```
      while (i ≤ |body(c)|) and (f) do
      begin
        f := Query(<q_i,u_i,v_i>) ;
        inc(i) ;
      end ;
      return (f) ;
    end ;
```

Query_Top_Level_Trace gets a clause instance as input. It returns *true* if for all i ($1 \leq i \leq m$) $<q_i, u_i, v_i>$ is in M.

Input: the triple <p, x, y>,
Output: a Boolean value.

```
    function Query (<p,x,y>) : boolean ;
    begin
      if Used_Test_Case(p,x) then  return( $\varphi_p(\chi_p(x))$ ) ;
      f := $\varphi_p(\chi_p(x))$ ;
      ans := Is_In_Interpretation (<p,x,y>) ;
      if (f ≠ undefined) and (f ≠ ans) then
        divide test frame $\chi_p(x)$ into two parts consistently.
      Store (<p,x,y>,ans) ;
    end ;
```

The function Query receives a triple <p, x, y> as input and returns a logical value. If there is a test case in the database which equals <p, x, y> then it returns the stored answer. Otherwise the question <p, x, y> ∈ M is asked from the user. If the test frame belonging to p and x is not evaluated yet then the answer is stored into this frame. In the other case we compare the answer with the evaluation of the frame. If these values are different then the test frame is inconsistent therefore the user is requested to refine the CPM test configuration consistently with the known set of test cases. Naturally all questions and answers are stored.

4.4 The modified single-stepping algorithm:

Input: <p, x, y> not in M,
Output: false clause instance q and Boolean f.

```
    procedure Modified_Single_Step (<p,x,y>,q,f);
    begin
      let T := ( <p_1,x_1,y_1>, <p_2,x_2,y_2>, ... , <p_n,x_n,y_n>)
          be the top level trace of <p,x,y>.
      f := false ;
      i := 1 ;
      while (i ≤ |T|) and (not f) do
      begin
        Modified_Single_Step (<p_i,x_i,y_i>,q,f);
        inc(i) ;
      end ;
      if not f then
      begin
        f := not Query (<p,x,y>) ;
        if f = true then  q := <p,x,y> ← <p_1,x_1,y_1>,<p_2,x_2,y_2>,...,<p_n,x_n,y_n> ;
      end ;
    end ;
```

Algorithm 4.4 differs from the original single-stepping method [SHA83] in that it uses the procedure Query which loops back to the CPM test configuration.

Let us suppose that <p, x, y> is not in the interpretation M. Let |N(<p, x, y>)| denote the cardinality of the nodes of the complete computation tree rooted at <p, x, y> (it is supposed to be finite). Then $0 \le n \le |N(<p, x, y>)|$ where n is the number of queries the user should answer. The best case is the following: the Algorithm 4.1 found exactly the false clause. Let $<q, u, v> \leftarrow <q_1, u_1, v_1>, <q_2, u_2, v_2>, \ldots, <q_m, u_m, v_m>$ be this clause. In Algorithm 4.2 we have to call the function Query m+1 times. If all of these queries occurred in the database then it terminates without any questions. The worst cases are: either the Algorithm 4.1 did not find the false clause or it finds one and the test frame belonging to its head is inconsistent. Both cases are caused by inconsistent test frames. In these cases we should call the original single-stepping algorithm.

5 Example

In this section we propose a detailed example on testing a sorting program. Let the program P be the following[SHA83]:

```
insert(X,[Y|U],[Y|V])   :- Y < X , insert(Y,U,V).
insert(X,[Y|U],[X,Y|U]) :- X ≤ Y.
insert(X,[],[X]).

sort([],[])   :- .
sort([X|Y],Z) :- sort(Y,U),insert(X,U,Z).
```

$\Sigma_{sort} = \{\sigma_{sort}\}$.
$\sigma_{sort} = \{c_0, c_1, c_2\}$.
c_0 = {the empty list}
c_1 = {all lists that contain exactly one element}
c_2 = {all lists that contain more than one element}

$\Sigma_{insert} = \{\sigma_{insert1}, \sigma_{insert2}\}$.
$\sigma_{insert1} = \{d_0, d_1, d_2\}$.
$d_0 = \{(X,[]) \mid X \in D\}$
$d_1 = \{(X,[Y]) \mid X, Y \in D\}$
$d_2 = \{(X,[Y|Z]) \mid X, Y \in D, Z \in D^+\}$

$\sigma_{insert2} = \{e_0, e_1, e_2\}$.
$e_0 = \{(X,[]) \mid X \in D\}$,
$e_1 = \{(X,[Y|Z]) \mid X, Y \in D, Z \in D^* \text{ and } X \le Y\}$
$e_2 = \{(X,[Y|Z]) \mid X, Y \in D, Z \in D^* \text{ and } X > Y\}$

It is obvious that the category σ_{sort} is an equivalence relation over the domain D^*, and similarly $\sigma_{insert1}$, $\sigma_{insert2}$ are also equivalence relations on $D \times D^*$. The set of test frames $F(\Sigma_P) = F(\Sigma_{sort}) \cup F(\Sigma_{insert})$ are defined as follows:

$F(\Sigma_{sort}) = \{c_0, c_1, c_2\}$
$F(\Sigma_{insert}) = \{(d_0, e_0), (d_1, e_1), (d_1, e_2), (d_2, e_1), (d_2, e_2)\}$

For these frames the following test cases are generated:

c_0	[]
c_1	[6]
c_2	[5,4,7]

(d_0, e_0)	(6,[])
(d_1, e_1)	(4,[20])
(d_1, e_2)	(10,[6])
(d_2, e_1)	(7,[10,5,1])
(d_2, e_2)	(15,[9,8])

Here is the content of the test database:

$T(F(\Sigma_P))$ =
{
(sort, c_0, [], [], *true*),
(**sort**, c_1, [6], [6], *true*),
(**sort**, c_2, [1,4,7], [1,4,7], *false*),
(**insert**, (d_0, e_0), (6,[]), [6], *true*),
(**insert**, (d_1, e_1), (4,[20]), [4,20], *true*),
(**insert**, (d_1, e_2), (10,[6]), [6,6], *false*)
(**insert**, (d_2, e_1), (7,[10,5,1]), [7,10,5,1], *true*),
(**insert**, (d_2, e_2), (15,[9,8]), [9,8,8], *false*)
}

The sort program terminates with an incorrect output on the list [6,-2,4]. We call the Algorithm 4.2 on *<sort,[6,-2,4],[-2,-2,4]>*. The first step is to call the *IDT_Debug (<sort,[6,-2,4],[-2,-2,4]>,r,f)*. IDT_Debug works on this input as follows:

It calls the function query with the following arguments but the answers are taken from the CPM test configuration.

<sort, [], []>	$\chi_{sort}([]) = c_0$	answer = $\varphi_{sort}(c_0)$ = *true*
<insert, (4, []), [4]>	$\chi_{insert}((4, [])) = (d_0, e_0)$	answer = $\varphi_{insert}((d_0, e_0))$ = *true*
<sort, [4], [4]>	$\chi_{sort}([4]) = c_1$	answer = $\varphi_{sort}(c_1)$ = *true*
<insert, (-2, [4]), [-2, 4]>	$\chi_{insert}((-2, [4])) = (d_1, e_1)$	answer = $\varphi_{insert}((d_1, e_1))$ = *true*
<sort, [-2, 4], [-2, 4]>	$\chi_{sort}([-2, 4]) = c_2$	answer = $\varphi_{sort}(c_2)$ = *true*
<insert, (-2, [4]), [-2, 4]>	$\chi_{insert}((-2, [4])) = (d_1, e_1)$	answer = $\varphi_{insert}((d_1, e_1))$ = *true*
<insert,(6, [-2, 4]), [-2, -2, 4]>	$\chi_{insert}((6, [-2, 4])) = (d_2, e_2)$	answer = $\varphi_{insert}((d_2, e_2))$ = *false*

The returned clause instance is *insert(6,[-2,4],[-2,-2,4])* $\leftarrow$ *-2 < 6, insert(-2,[-2,4],[-2,-2,4])*. First we should ask the membership question:

? *insert(6,[-2,4],[-2,-2,4])* $\in$ M *no.*

Since the answer was *no* we inspect all atoms in the body:

? *insert(-2,[-2,4],[-2,-2,4])* $\in$ M *true.*

The Algorithm 4.2 terminates successfully and returns in r the clause instance **insert(6,[-2,4],[-2,-2,4])** $\leftarrow$ **-2 < 6, insert**(-2,[-2,4],[-2,-2,4]) which is exactly the false clause.

6 Summary

In this paper we have presented an overview of an Interactive Diagnosis and Testing method for logic programs. The basic idea of this approach is that we try to give an integration of the debugging and testing phases of the software development process. It means that the IDT algorithm introduced in this paper uses the test results of a program during the bug localization process to reduce the number of user interaction. In addition, if during the IDT algorithm an inconsistent test frame is identified then a refinement of the given test specification is prepared.

On the basis of the IDT method a prototype system (called *IDTS*) is under development. A small prototype for CPM test generator in SB-Prolog environment is implemented. The Shapiro's algorithms for automatic program debugging are under implementation. After the complete implementation we will integrate the testing and debugging programs and we will prepare the system IDTS.

We believe that the IDT method can be applied in the process of Interactive Inductive Logic Programming. We try to investigate this topic in the future.

7 References

[FRI91] Peter Fritzson, Tibor Gyimóthy, Mariam Kamkar, Nahid Shahmeri Generalized Algorithmic Debugging and Testing. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada June 26-28, 1991. pp. 317-326.

[OST88] Thomas J. Ostrand, Marc J. Balker: The Category-Partition Method for Specifying and Generating Functional Tests. CACM 31:6, June 1988. pp. 676-686.

[SHA83] E. Y. Shapiro: Algorithmic Program Debugging MIT Press 1983.

[SHA90] N. Shahmeri, M. Kamkar and P. Fritzson: Semi-automatic Bug Localization in Software Maintenance. San Diego, Nov. 26-29, 1990.

[SZU91] Róbert Szűcs, Tibor Gyimóthy: T-GEN: An Extended Version of the Category Partition Testing Method. Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala Finland August 21-23, 1991. pp. 70-77.

An Overview of the TaLE Language Editor

Esa Järnvall and Kai Koskimies
Department of Computer Science, University of Tampere,
P.O. Box 607, SF 33101 Tampere, Finland

email: ejj@cs.uta.fi, koskimie@cs.uta.fi

Abstract

TaLE is a specialized editor for developing language implementations in an object-oriented (Eiffel) programming environment. In contrast to conventional language implementation systems, there is no formal metalanguage for specifying a language; instead, the user edits the classes taking part in the implementation under the control of a specialized editor offering high-level, partly graphical views of those classes. The system supports the reuse and refinement of the language implementation classes, incremental implementation development, integration of syntactic and name analysis, and special views for classes representing standard language features. The expected main advantages of the system are high usability (due to the metalanguageless approach) and fast development cycle due to the high-level facilities and reusing capabilities). The basic features of the system are presented using a small example task.

1 Introduction

TaLE (Tampere Language Editor) is a new tool supporting the development of language implementation software in an object-oriented programming environment. The design of TaLE is unconventional in the sense that TaLE emphasizes software engineering qualities rather than contributions in formal language specification; this makes the system in many ways different from more traditional language implementation systems. In fact, we feel justified to say that TaLE employs a different paradigm: the user is not expected to write a language specification, but to edit the software units (classes) taking part in the implementation under the control of a specialized editor. Currently the language of the produced software is Eiffel [Mey88], but in principle this language could be any class-based object-oriented language. The system is implemented in Eiffel 2.3 under Sun3/UNIX.

TaLE is particularly intended for the rapid implementation of application-oriented languages, i.e. for various nontrivial textual representations of data, specifications, algorithms etc.; typically these are "little languages" as proposed by Bentley [Ben86]. Since such a language is often only a small aspect in a large software project, we cannot expect that the user of a language implementation system would be willing to learn a new formal metalanguage based on a theoretically oriented specification paradigm. The simpler and the more self-explaining the user interface of a language implementation system is, the smaller tasks will be included in the potential applications of the system. Our intention is that an average programmer will decide to use our system for virtually all nontrivial processing of structured textual data, in the same way as GUI editors are currently used for implementing graphical user interfaces. This puts high demands on the user-friendliness and simplicity of the system. In TaLE, these requirements are hoped to be satisfied through the metalanguageless, editor-based approach, and through high-level, intuitive views of the classes taking part in the implementation.

Conventional language implementation systems are typically closed: they provide a mapping from an abstract specification into an executable language processor written in a target language, and the user is not expected to understand the target language, far less modify the resulting processor. However, this idealistic view is rarely fully possible in practice because the abstract metalanguage is not general or efficient enough, or because the produced language implementation is part of a larger system, and the resulting code must be patched to integrate it with the rest of the software. The editor-based approach of TaLE leads to an open system in which the use of the base language is natural and in which the user essentially gets what s/he sees. A necessary requirement for this

approach is the object-oriented programming paradigm which allows a close relationshi
concepts of the implemented language and the software units (classes) of the implementatio.

In the long run, the crucial factor of the productivity in software development is reuse. In languag
implementation this aspect has been mostly ignored (at least in the usual sense of software reuse), because programming languages have been regarded as indivisible entities that give little opportunities for sharing common code. However, actually this is not true: there is a lot of almost identical features in different languages, and it seems reasonable to assume that this could be reflected in the implementation as reusable units. For example, practically every language has a notion of an arithmetic expression, with minor variations. This implies that essentially the same concept is implemented over and over again, and that very similar code is repeated in numerous language processors. The same holds for concepts like standard constant denotations, control structures, subprogram mechanisms, type systems etc. The possibility to reuse code is particularly obvious for special-purpose languages that are currently under design: in many cases it would be sufficient to simply pick up a suitable standard form of, say, arithmetic expressions from a library of standard language features, in the same way one employs a standard data structure from a general library. Even if a direct adoption of a library feature is not appropriate, it should be possible to easily modify and extend a library unit according to the needs of a particular language.

In fact, certain kinds of reuse are fairly common, although perhaps not identified as such: special-purpose languages are often developed by extending a general-purpose language with application-specific features, or by embedding a particular structure from a general-purpose base language within a special-purpose language. In both cases one actually reuses all or some of the structures of a general-purpose language, in the hope that they need not be reimplemented. Here we want to generalize this kind of language development, and regard languages as collections of relatively independent, replacable units.

TaLE supports the reuse of language concepts and structures in three ways: first, by employing a distributed implementation model in which language structures are implemented by highly independent classes TaLE allows a language to adopt structures (and their implementations) from other languages; second, general language-independent concepts can be implemented on an abstract level and refined for individual languages (making use of the subclassing mechanism); third, standard language concepts and their implementations are built into the system, so that the language implementer can adopt these concepts into his/her language through special interfaces. Together with the high-level views provided by the metalanguageless user interface, the facilities supporting reuse are expected to speed up the language development process in most cases by an order of magnitude when compared to traditional systems like LEX/YACC.

Although the overall character of TaLE is unique (as far as we know), there are similarities with existing systems. Object-oriented language implementation techniques have been applied in Muir [Win87], TOOLS [KoP87], Orm ([Hed89], [Mag90]), Ag [Gro90], OOAG [ShK90], SmallYacc [AMH90], and Smalltalk [Gra92]. Object-oriented context-free grammars have been introduced already in [LMN88], [Ten88] and [Kos88]. Incremental language implementation (especially parsing) has been investigated in [GHK88], [Hor89], [HKR89] and [Kos90]; the latter method has been used as a starting point in TaLE, too.

In the following we present the central features of TaLE using a small example. Although the example language is by no means a realistic one, we have tried to make it sufficiently interesting for demonstrating purposes. The language does not represent a typical case for TaLE; for example, the language is processed entirely statically (i.e. the semantic processing is merged with analysis) which is clearly a special case. Our example language is a simple desk calculator language: we want to be able to express sequences consisting of assignments to named variables and output instructions, e.g.:

```
X:= 55; Y:= (X+24)*X; Z:= X*Y; OUT Z+220
```

Since we assume that a variable can be referenced only if its value has been defined before, each assignment and output instruction can be "executed" immediately after it has been analyzed.

2 Object-oriented context-free grammars

A nonterminal of a context-free grammar is a structural specification of an object (an instance of the nonterminal) appearing during the analysis of a source. Hence there is a direct analogy between nonterminals and classes in the object-oriented sense: a nonterminal can be viewed as a class specifying the node objects in the internal representation tree of the source. On the other hand, the natural object-oriented interpretation of syntactic alternation is subclassing: production rules A -> B C and A -> D E (or A -> B C | D E) imply that an instance of A may take two different, alternative forms, that is, class A has two subclasses. The problem in this case is that the subclasses have no names, and unnamed classes usually cannot be allowed. We could solve the problem by rewriting the rules in the form A -> A_1 | A_2, A_1 -> B E, A_2 -> D E, introducing two new nonterminals as the names of the alternative forms of A. However, we should clearly understand that we are now using the production symbol ("->") for two essentially different purposes: A -> A_1 | A_2 means subclass relation while A_1 -> B E and A_2 -> D E mean structural specifications. We make this distinction more explicit by using a different symbol for the former purpose: A > A_1 | A_2.

In an object-oriented context-free grammar (OO-CFG) each nonterminal symbol (or class) A has exactly one rule, either of the form A > B_1 | B_2 | ... | B_k or of the form A -> B_1 B_2 ... B_k ($k \geq 0$). In the former case A is called a *conceptual class*; in the latter case A is called a *structural class*. We assume that ">" does not imply a cyclic class hierarchy for the conceptual classes. We allow, however, multiple inheritance, i.e. rules of the form A > B, C > B.

Our example language can be presented as an OO-CFG as follows (we use extended syntactic notation with iterations):

```
Program -> Instruction (";" Instruction)*
Instruction > Variable | Output
Variable -> id "=" My_expression
Output -> "OUT" My_expression
My_expression -> Term (AddOp Term)*
Term -> Factor (MulOp Factor)*
Factor > Variable_access | Constant
Variable_access -> id
Constant -> number
```

Note that without a different arrow symbol the last two rules could be interpreted also as subclass relations: two different symbols are indeed necessary to make the interpretation unambiguous.

An OO-CFG implies an internal representation of the source program as a collection of objects, deviating slightly from the conventional one. Note that the subclass relation ">" does not give rise to a separate node in the object representation; it merely indicates a class layer in a node object. Hence the object representation of the source program is more abstract than a conventional syntax tree: in conventional terminology, typical chain productions are eliminated from the representation.

3 Using the built-in high-level concepts

The first task of the language implementer is to consider the existing concepts in TaLE: is there something we could directly apply in our language? The TaLE class browser (Fig. 1) shows e.g. classes Std_Expression (standard arithmetic expressions) and Std_List (a standard list structure) which seem to be useful to us. Further, Pascal_id and Pascal_int (subclasses of Identifer and Std_integer) could be candidates for expressing identifiers and integer constants, and Simple_output seems a promising class for implementing the output instruction. We can hope to be able to use these classes either directly or as superclasses in our language.

We use the following textual notation: if a class (say A) is defined as a subclass of another class (say B), we write A = B(...), where the parenthesized part contains the subclass parameters. We do not here give any actual form for the subclass parameters.

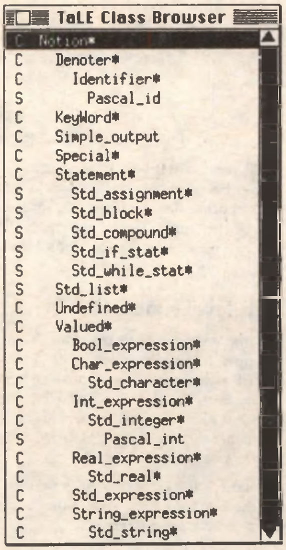
Our "plan" to implement the example language could be now presented as follows:

```
Program = Std_list(...)
Instruction > Variable | Output
Variable -> Pascal_id "=" My_expression
Output = Simple_output(...)
My_expression = Std_expression(...)
```

Let us first define a class for the expression structure we need in our example language. For this purpose we use the standard facilities provided by predefined class Std_expression: we give a "create subclass" menu command for class Std_expression, and the special view for expressions is displayed to allow the user to construct the subclass with the given name (here My_expression). This is an example where a standard language notion is reused through a special interface allowing the fine-tuning of the concept.

The expression view is shown in Fig. 2 after completion: the user has selected the operator symbols s/he wants, their precedences and associativeness, the allowed type combinations (shown in a separate window), the representations for constants (as classes; here we need only integer constants given as class Pascal_int), the name of the class giving the other primitive constituents of expressions (here we give a new, so far undefined class Variable_access for this purpose), and the parenthesis convention. Further, the user can specify whether s/he wants static type cheking and/or static evaluation of the expression; in our simple example language we can decide to use static evaluation. If only standard operations +, -, *, /, are needed, this is all that has to be done; in the case of non-standard operators the user must give the Eiffel-statements that implement the operator in a separate text window.

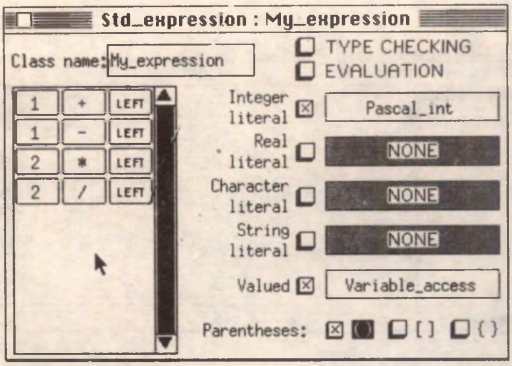
Note that we applied here also another type of reuse: we adopted Pascal's integer representation (Pascal_int) directly as such in our language, with its full implementation. In this case the reused structure is a simple token, but in principle all structures (classes) are reusable in this way: due to our incremental approach a structure is not tied to a particular language but a more or less independent unit.



```
C  Notion*
C    Denoter*
C      Identifier*
S        Pascal_id
C    KeyWord*
C    Simple_output
C    Special*
C    Statement*
S      Std_assignment*
S      Std_block*
S      Std_compound*
S      Std_if_stat*
S      Std_while_stat*
S    Std_list*
C    Undefined*
C    Valued*
C      Bool_expression*
C      Char_expression*
C        Std_character*
C      Int_expression*
C        Std_integer*
S          Pascal_int
C      Real_expression*
C        Std_real*
C      Std_expression*
C      String_expression*
C        Std_string*
```

Fig.1. The class browser. "C" denotes "conceptual" and "S" denotes "structural" (see text).

The view of the subclasses of Std_expression shown in Fig. 2 is an example of a *special view*, tailored for the particular properties of the class. Currently there are other special views for instance for the standard lexical structures (Identifier, Std_character, Std_integer, Std_real, Std_string) allowing certain structural properties to be individually selected. In principle special views can be designed for all sufficiently standard language features.



Class name: My_expression

☐ TYPE CHECKING
☐ EVALUATION

1	+	LEFT
1	-	LEFT
2	*	LEFT
2	/	LEFT

Integer literal ☒ Pascal_int
Real literal ☐ NONE
Character literal ☐ NONE
String literal ☐ NONE
Valued ☒ Variable_access
Parentheses: ☒ ( ) ☐ [ ] ☐ { }

Fig. 2. The view for expression class My_expression.

3 Graphical class views

Let us next construct a class for the variable assignment structure, called Variable. This class we will construct from scratch, rather than building on an existing reusable class. We ask the system to create a new structural subclass for the root class Notion; the system the displays a general structural class view for constructing the new class. This view is shown in Fig. 3 after some editing actions.

The upper section of the view contains the feature list, the check list, and buttons for giving the class certain special properties. The feature list contains all the (visible) attributes and operations of the class, including the inherited ones (the latter are associated with the defining class in brackets). The user may introduce new features; the Eiffel code of the features is given in a separate text window appearing when a feature is added or edited. In this class we need no user-given features.

The check list contains the semantic checks carried out during analysis (e.g. type checking). Each check is denoted in the list by a string which serves as the message emitted when this check fails; the check itself is given as a Boolean expression over the attributes of the class and the attributes of the components in the pattern. Here we need no checks, either.

The property buttons are actually short-cuts for inheriting certain predefined classes, but they also cause some additional actions to be carried out automatically by the system. Button "SCOPE" makes the class a static visibility region, i.e. the language structure it represents will be associated with a set of named objects. Button "NAMED" associates the class with the properties of named objects that can be stored in a built-in object base; each object with this property is automatically inserted into the object base as a member of the set associated with the smallest enclosing "SCOPE" structure. Button "VALUED" associates the class with a special value attribute of a predefined class; currently this class is capable of representing all the scalar values of Eiffel. Finally, button "TOKEN" turns the structure into a syntactic token in the sense that no spaces are allowed between the different parts of the structure, and that the parser uses this structure as a whole for syntactic look-ahead. This is the way arbitrary user-defined token categories can be introduced in TaLE, in addition to the (fairly covering) standard ones provided by predefined classes and their special views.

Since each variable assignment can be seen as a declaration of the left-hand side variable in our example language, we tick the NAMED button; consequently, the class inherits a string-valued attribute *key* containing the identifying name of the object. Likewise, we tick the VALUED button since we want that each assignment is associated with a value; as a result, the general *value* attribute is inherited from a system class.

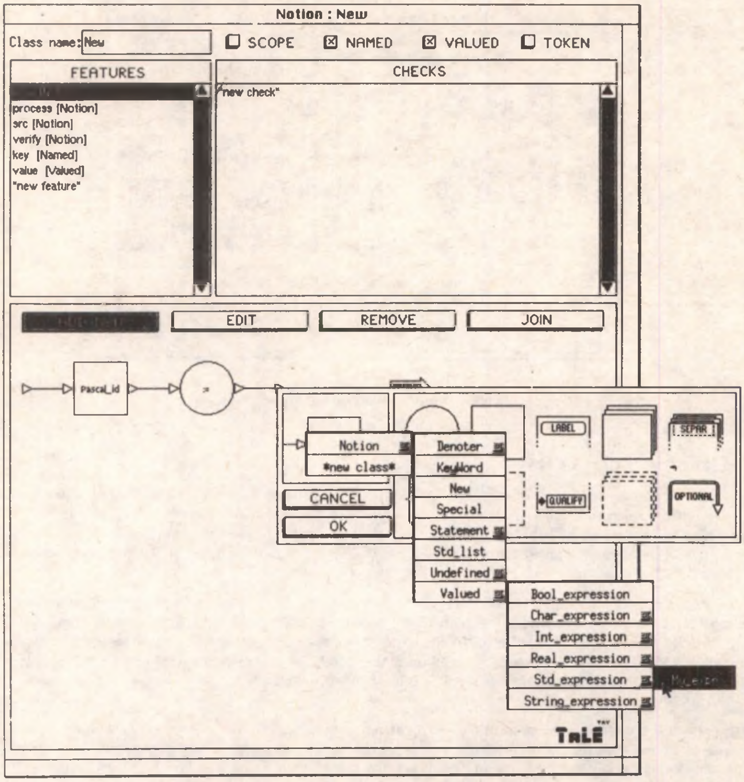


Fig. 3. The editing window for the assignment structure.

In the lower section of the view the syntactic structure is given graphically as a "railyard" syntax; we call this the *pattern* of the class. The pattern is constructed and edited directly using mouse-driven commands and the mode buttons appearing in the upper part of the pattern section. The icons denoting components of a pattern are selected from a palette appearing when a component is added or edited. The view of Fig. 3 is shown in a situation where the user is constructing the pattern: the icons representing the left-hand side and the assignment symbol have been inserted, and the user has indicated s/he wants a new component (for the right-hand side expression) in the pattern; as a result, the palette is shown allowing the user to select the kind of the component and the class it represents. The class can be selected from a hierarchical, dynamic menu showing all the existing classes; here the user has selected class My_expression. The icons in the palette represent a single keyword, a single substructure, a list structure, a list separator, a set of alternative keywords, a secondary structure (a named substructure without a class of its own), a secondary list, and a passing arc (for making a component an optional one).

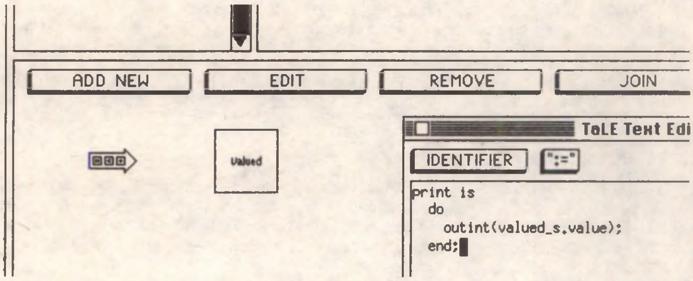
Each arrow head in the pattern represents a code location: the user may insert arbitrary Eiffel code into the pattern by clicking on an arrow head. This results in the opening of an Eiffel window in which the user may type any sequence of Eiffel statements. These statements will be executed during the analysis phase at the corresponding point. The system assists the user in the writing of the statements: the features of the component structures need not be explicitly written but they can be selected from a menu displayed when the corresponding pattern icon is pointed by the mouse. In this case we must define the values of the key and value attributes: we click on the last arrow head (since we want that these actions are carried out after the analysis of the whole assignment), and write the following text in the opened text window:

```
key := Pascal_id_s.src;
value := My_expression_s.value;
```

That is, the key attribute gets its value from the source string (src) corresponding to the Pascal_id component, and the value is taken directly from the value of the My_expression component. Finally, we give this class name Variable, and exit the class window.

4 Reusing abstract general-purpose classes

Let us next create a class for output statements. For output structures TaLE offers no built-in facilities, but what a lucky coincidence: somebody has previously constructed class Simple_output which we can now reuse. Simple_output is obviously intended to be reused through subclassing; this class is conceptual. Part of the existing specification of Simple_output is shown in Fig. 4.



```
print is
  do
    outint(valued_s.value);
  end;
```

Fig. 4. Part of the class window for Simple_output.

Since Simple_output is a conceptual class, it does not have a pattern. Nevertheless, even a conceptual class can have components which exist independently of a pattern; we call them abstract components. In this case there is a single abstract component of class Valued. In addition, the class defines an operation called print; this operation simply prints out the value attribute of the abstract component. The code for the print operation is shown in a separate text window.

We can now construct the class Output as a structural subclass of Simple_output. The view shown for this subclass contains initially the inherited abstract components, located in sequence after the thick arrow symbol. The user must "consume" all the inherited abstract components in the pattern of the structural subclass, by dragging them into their place in the pattern. In this way the user associates each abstract component with some concrete component position in the pattern. When an abstract component has been inserted into a pattern, it can be further refined, i.e. its class can be narrowed from the original one. Fig. 5 is shown in a situation where the user has already inserted the keyword icon into the pattern, and is next going to drag the abstract Valued-component into its place. This component must be further refined to class My_expression. Note that the inherited components are displayed with thick border, indicating that they cannot be edited in this view.

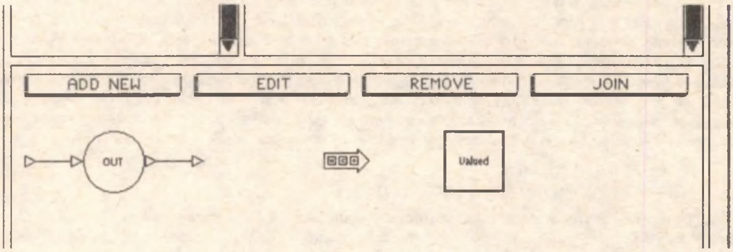


Fig. 5. Constructing the Output class.

The actual effect of an output instruction can be easily realized using the inherited print-operation: all we have to do is to insert the call of print into the last arrow head of the pattern of Output.

TaLE is an incremental system: each constructed class is a full-fledged Eiffel class after its editing has been completed. Usually a class need not even be recompiled when some other classes are modified, although the class makes use of the modified classes; this is due to the distributed implementation strategy of TaLE (for details, see [JäK93]). Since each undefined class is given a default implementation (an undefined class X is assumed to have a pattern with a single keyword item "X"), it is possible to test a class at any point, even if it makes use of undefined classes.

We can test class Output by activating a special Test-command in the main manu. The opened tester window asks for the class to be tested. We select the class Output and write in the input pane:

OUT 2+3*4

We click on an activation button, and observe the result "14" in the output pane.

At this point we could construct class Instruction as well. Since the only purpose of this class in our language is to collect classes Variable and Output under a common name, class Instruction needs no features itself: everything will be specified in its subclasses. Recall that alternation is described in OO-CFGs via subclassing; hence it is sufficient to specify that Variable and Output are subclasses of Instruction.

Since we have no particular requirements for class Instruction, we create it as a conceptual subclass of the root class, Notion. Immediately after giving its name we exit the Instruction window, and return to the class browser level. There we use a special multiple inheritance command, forcing Variable and Output to inherit Instruction. This command is used for making inheritance relations required for syntactic reasons, and it affects only the system-dependent parts of the Eiffel classes. Nevertheless, the affected classes have to be recompiled.

5 Automated name analysis

Let us concentrate on the so far undefined class Variable_access, representing the variable references in an expression. This is a structural class: it has a particular syntactic form consisting of an identifier. We define it as a subclass of Notion and tick the VALUED button in the appearing view. We insert a single component to the pattern, and select the class Pascal_id for the component (Fig. 6). Since this component must be associated with an existing variable, we *qualify* it with class Variable: in this way we make sure that the identifier indeed is the name of an existing Variable object. Using qualification, the association of named entities and their references in the source is carried out automatically by the system. An additional advantage of using qualification is that the parsing process can make use of the qualification information, and avoid LL(1) look-ahead conflicts that would otherwise arise (although in this case there is no fear of that).

We must further specify that the value of a Variable_access object is obtained from the value of the Variable object denoted by this object. We click on the arrow head at the end of the rail-yard syntax and write the following text in the opened text window:

```
    v ?= denoted;
    value:= v.value
```

where denoted is a predefined attribute that automatically refers to the Named (Variable) object associated with this object. Attribute v is a new feature that must be given to class Variable_access due to the type rules of Eiffel: since the static class of denoted is Named, it is not guaranteed that the denoted object would inherit Valued; therefore we must introduce an additional attribute v of type Valued, and apply so-called reverse assignment attempt ("?=") which checks the dynamic class of the right-hand side object (in this case the check never fails).

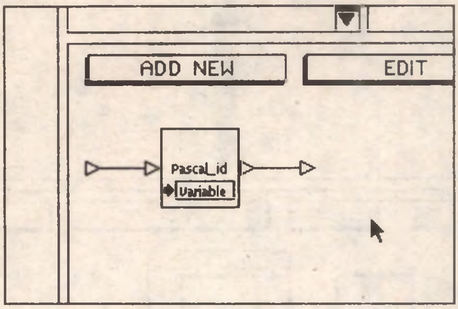


Fig. 6. The pattern for Variable_access, with a qualified component.

6 Reusing structures: refinement of components

Since the entire "program" in our example language is a list of something, we define it as a subclass (named Program) of Std_list. When the "create subclass" command is issued for Std_list, the view in Fig. 7 is shown. This view is a so-called refinement view[1] in which the original, inherited syntactic pattern is shown with thick border lines, indicating that it is not modifiable. What the user can do is to refine the components in the pattern, i.e. to narrow their classes. This is done by specifying the classs in each refinement icon located under the actual component icon. In the inherited pattern of Std_list, the list element is specified to have class Notion, implying that any class will do here. The concept of Std_list also includes a separator, which is specified to be of class Key_word; this is a special built-in class whose subclasses are implicitly all individual key strings or sets of such strings (a key string is a class only in a technical sense, allowing conceptually unified handling of keywords). It is also possible to give new arrow head actions and new features and checks in the refinement view.

In our example language the list element class is Instruction, and the separator symbol is semicolon. Fig. 7 is shown in a situation where the user has already selected Instruction as the refinemnent class of the Notion component, and is currently refining the separator component. Here these refinements are sufficient for the complete specification of class Program, and this concludes the implementation of our example language. Note that in some other application it might be sensible e.g. to add some semantic processing in the arrow head actions, and obtain thus specialized list classes which could be further refined for individual languages.

[1]Strictly speaking this is a special view for lists: it contains a "backward" arc which is not allowed in the usual patterns; hence this kind of pattern could not have been created by the user. However, since the subclass view for Std_list otherwise looks like a refinement view, we use it here as an example of that.

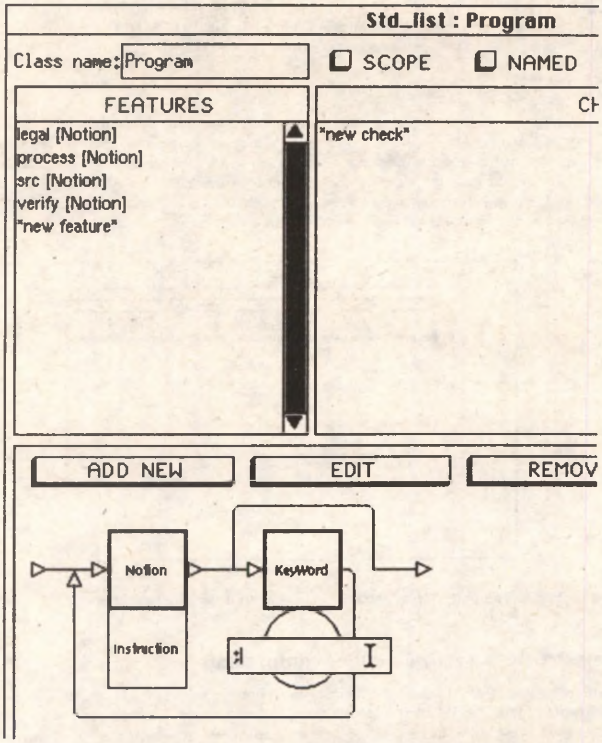


Fig.7. Part of the view shown for giving a subclass of Std_list. The user has just typed in the separator symbol (";").

7 Conclusions

This example demonstrates some of the advantages of TaLE, when compared to more conventional language implementation systems. The user need not know any special language implementation paradigm or formalism: we only expect that s/he understands object-oriented programming and the underlying OO language (here Eiffel). Most of the work is done using generally understandable graphical facilities in an interactive environment. The user has the feeling of editing rather than that of writing a formal specification.

If the language to be implemented is reasonably compliant with the existing classes of TaLE, the implementation can be carried out with very little work. The system supports the implementation development on several levels. The strongest support is achieved if an existing class can be used as such. In the TaLE implementation model, the classes implementing different language structures are independent units, and they can be freely combined to form new languages (e.g. Pascal_int and Pascal_id in the example). The second level of support is offered through the high-level built-in facilities for standard language features: these allow the construction of new language specific classes through a specialized interface (e.g. My_expression in the example). The third level is obtained through the general specialization mechanism of all classes, allowing the user to add abstract components (Simple_output in the example), to associate the abstract components with concrete ones (Output), to refine the classes of components (Program), and to add new features and analysis-time actions (arrow-head actions). Finally, the fourth level comprises of the basic mechanisms of TaLE like the graphical syntactic specification, support for writing feature references, automated name analysis, etc. All this has the effect that the need to write Eiffel code is minimized.

The system is open. Except for the fact that the implementation classes are viewed through a specialized editor, there is nothing special in those classes. The interfaces of the classes are visible and understandable for the user, and s/he may use them freely in other software. The internal representation of the source is a normal collection of objects that can be associated with arbitrary processing. There are no hidden interpreted system files.

The system is incremental. As long as the interface of a class is not changed (e.g. by removing or adding a user-given operation), the editing of a class does not necessitate the reproduction and/or reprocessing of all the other classes, or even the client classes. For instance, syntactic changes can be freely made in the patterns without affecting the other classes. There is no global information about a language that should be updated after each modification. Individual language structures can be tested independently.

The success of the TaLE approach clearly depends on the extent predefined class libraries can be used to support the implementation of new languages. We expect that different TaLE class libraries will be developed for different application domains, so that the language development can be done (re)using concepts that are already near to the language. Trivially, each language implementation carried out in TaLE is in fact a specialized class library that can be utilized e.g. in the development of the next generation of the language; hence this approach seems to be especially useful to maintain the implementation of a relatively non-stable language in a specialized environment.

Acknowledgements

This work has been financed by the Academy of Finland through grant 1061120, and by the University of Tampere. We have received significant support from our fellow researchers Jukka Paakki and Juha Vihavainen. We wish to thank the following students who have programmed parts of the system: Maarit Niittymäki, Annika Knuuttila, Janne Lahti, Marianne Setälä, Jari Torkkola, Hannu Vainio-Mattila, and Toivo Venäläinen.

References

[AMH90] Aksit M., Mostert R., Haverkort B.: Compiler Generation Based on Grammar Inheritance. Mem Informatica 90-07, Department of Computer Science, University of Twente, February 1990.

[Ben86] Bentley J.: Little Languages. Programming Pearls, CACM 29,8 (August 1986), 711-721.

[Gra92] Graver J.O.: The Evolution of an Object-Oriented Compiler Framework. Software Practice & Experience 22, 7 (July 1992), 519-535.

[Gro90] Grosch J.: Object-Oriented Attribute Grammars. In: Proc. 5th International Symposium on Computer and Information Sciences (ISCIS V), A.E. Harmanci, E. Gelenbe (eds.), Cappadocia, Nevsehir, Turkey, 1990, 807-816.

[GHK88] Gyimóthy T., Horváth T., Kocsis F., Toczki J.: Incremental Algorithms in PROF-LP. In: Proc. of Workshop on Compiler-Compilers, Lecture Notes in Computer Science 371, Springer-Verlag 1989, 93-102.

[Hed89] Hedin G.: An Object-Oriented Notation for Attribute Grammars. In: Proc. of the European Conference on Object-Oriented Programming (ECOOP '89), Nottingham, 1989. The British Computer Society Workshop Series, Cambridge University Press 1989, 329-345.

[HKR89]	Heering J., Klint P., Rekers J.: Incremental Generation of Parsers. In: Proc. of ACM Sigplan '89 Conference on Programming Language Design and Implementation, Portland, Oregon. Sigplan Notices 24, 7 (1989), 179-191.
[Hor90]	Horspool R.N.: Incremental Generation of LR Parsers. Journal of Computer Languages, 15, 4 (1990), 205-223.
[JäK93]	Järnvall E., Koskimies K.: Language Implementation Model in TaLE. Report A-1993-1, Department of Computer Science, University of Tampere, 1993.
[KoP87]	Koskimies K., Paakki J.: TOOLS - A Unifying Approach to Object-Oriented Language Implementation. In: Proc. of ACM Sigplan '87 Symposium on Interpreters and Interpretive Techniques, Sigplan Notices 22,7 (July 1987), 153-164.
[Kos88]	Koskimies K.: Software Engineering Aspects in Language Implementation. In: Proc. of Workshop on Compiler Compilers and High-Speed Compilation, D. Hammer (ed.), LNCS 371, Springer-Verlag 1988, 39-51.
[Kos90]	Koskimies K.: Lazy Recursive Descent Parsing for Modular Language Implementation. Software Practice & Experience 20, 8 (1990), 749-772.
[LMN88]	Lehrmann Madsen O., Nørgaard C.: An Object-Oriented Metaprogramming System. In: Proc. 21st Annual Hawaii International Conference on System Science (B.D. Shriver ed.), 1988, 406-415.
[Mag90]	Magnusson B., Bengtsson M., Dahlin L.-O., Fries G., Gustavsson A., Hedin G., Minör S., Oscarsson D., Taube M.: An Overview of the Mjølner/Orm Environment: Incremental Language and Software Development. Report LU-CS-TR:90:57, Department of Computer Science, Lund University, 1990. Also in Proc. of TOOLS '90, Paris 1990.
[Mey88]	Meyer B.: Object-Oriented Software Construction. Prentice-Hall 1988.
[ShK90]	Shinoda Y., Katayama T.: Object-Oriented Extension of Attribute Grammars and its Implementation. In: Workshop on Attribute Grammars and Their Applications, P. Deransart, M. Jourdan (eds.), Paris, LNCS 461, Springer-Verlag, 177-191.
[Ten88]	Tenma T., Tsubotani H., Tanaka M., Ichikawa T.: A System for Generating Language-Oriented Editors. IEEE Trans. on Software Engineering 14,8 (August 1988), 1098-1109.
[Win87]	Winograd T. A.: Muir: A Tool for Language Design. Report STAN-CS-87-1159, Department of Computer Science, Stanford University, May 1987.

ON USING TYPE INFORMATION IN SYNTACTICAL DATA COMPRESSION

Jyrki Katajainen
University of Copenhagen
Department of Computer Science
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
email: jyrki@diku.dk

Erkki Mäkinen
University of Tampere
Department of Computer Science
P.O. Box 607
SF-33101 Tampere, Finland
email: em@cs.uta.fi

Abstract. This note proposes a new method for compressing program files. In syntactical compression of program files linearization of parse trees are used. A linearization of a parse tree contains production labels and symbol table references to user terminals. By storing type information concerning user terminals, it is possible to leave out some production labels from the linearization and to decrease the number of bits needed for some labels.

Key Words and Phrases: syntax-directed compression, semantic compression.

1. Introduction

The effectiveness of text compression depends on how much information is available about the structure of the text. When compressing program files, the syntax of a program constitutes a basis for efficient compression methods; for a survey on syntactical compression see chapter 6 in [5]. The information we have about the structure of the input text, i.e. about the program, is given in the form of a context-free grammar defining syntactically correct programs. Syntactic approaches are known to give better compression results than conventional lexical approaches.

The compression process in syntactic methods can be described as follows. The input program is first scanned and the tokens found are classified into *syntactic terminals* (keywords, operators, punctuation symbols) and *user terminals* (identifiers, constants). The user terminals are gathered to the symbol table. Then the parse tree is generated. Each internal node in the parse tree is associated with the label of the production applied at the node. Each leaf contains a pointer to a symbol table entry. *Linearization* of the parse tree is obtained by traversing the tree in preorder; it contains both the labels stored in the internal nodes and the pointers stored in the leaves. The original program can be reproduced from the linearization of the parse tree .

As noted by several authors [3,9,11], global production labels contain redundant information. Namely, when reproducing the program from a label sequence, we always know the leftmost nonterminal of the current sentential form. So, we do not need global production labels but labels which distinguish between productions having the same left hand side. Arithmetic coding can be used when coding the production labels.

The context-free grammar defining syntactically correct programs contains all possible syntactic information about the programs to be compressed. To obtain better compression gains one can try to extract other kind of information from the grammar. The purpose of the present paper is to show that better compression results can be obtained if in addition to the syntactic properties also some semantical aspects are taken into considerations as well. We consider the compression of Pascal programs (see [4] for the grammar); similar results can be obtained by using other programming languages, too.

We would like to stress that only correct programs are acceptable as input. It is normally supposed in the literature that programs to be compressed are syntactically correct. This is not a serious restriction, since incorrect programs are rarely stored for a long time or transmitted via a computer network. Moreover, if the input program is produced by a syntax-directed editor, it cannot be incorrect. It is also normally supposed that all comments and formating features are omitted. A pretty printer program is supposed to be available to achieve readable program from the decoded output.

According to Peltola and Tarhio [9, Table I] the compressed version of a typical program file occupies storage as follows: about one third of the bits are needed for the parse tree (excluding the information concerning user terminals); symbol table references, i.e., pointers indicating the occurrences of user terminals, usually take a majority of the rest space needed. About 10-30 per cent of the space is needed for storing the names of the user terminals. We concentrate our compression efforts to parse trees and to symbol table references.

We organize the symbol table such that type information concerning user terminals can be succinctly stored in it. The advantages gained are twofold. First, some labels of productions become unnecessary in the parse tree, since we can deduce them from the types of associated user terminals. Secondly, this information justifies some grammar transformations which decrease the size of the parse tree. Katajainen et al. [6,7] changed the underlying Pascal grammar in many ways. For example, precedence and associative rules of expressions are left out. This reduces the number of internal nodes of the parse tree. However, as noted by Peltola and Tarhio [9], it has the drawback that the number of production alternatives per a node increases. When reproducing a Pascal program from the left parse and user terminals there are several situations where the number of production alternative can be decreased if the type of user terminals is known. We argue that the type information stored in the symbol table justifies the grammar transformations made by Katajainen et al. [6,7], and in fact, gives us better compression results.

When changes are made to the context-free grammar defining the language, it may become necessary to perform parsing using one grammar and to compress using another grammar. Fortunately, there is a rich theory concerning mappings between context-free grammars. Provided that a grammar transformation obeys same natural conditions, we say that there is a *cover* relation between the grammars. Consult [8] for details concerning grammatical coverings and related topics.

2. Symbol table organization

A compiler uses a symbol table to keep track of scope and binding information about names encountered in the input text (see [1]). In a system for compressing program files a symbol table makes it possible to replace the occurrences of user terminals by symbol table references. An entry in compiler's symbol table usually contains attributes that have no relevance in a compression system. In our system the only symbol table attribute is the *type category* of a user terminal.

During compression and decompression the symbol table is organized into lists maintained by the move-the-front heuristic. Peltola and Tarhio [9] use two lists. One list is for identifiers and the other one is for numerical constants. A reference to a symbol table entry specifies the list in question and the position inside the list. Also Cameron [3] mentions the possibility to slice the symbol table, but without any attempt to make use of the semantic information. We do not know any previous attempts to obtain better compression results by using semantic information.

We suggest the use of a separate move-to-front list for each type category. This means that more bits are needed for specifying the list in question. However, as little as four bits, i.e., 16 type categories, should be enough. On the other hand, more lists means fewer bits for the positions inside lists. The use of the move-to-front heuristic can be motivated by the fact appearances of words are clustered both in natural languages and in programs [2,10]. We argue that this clustering phenomenon can be utilized in a more efficient manner when user terminals of different types are stored in different lists.

In the compressed version the only attribute of symbol table entries is presented implicitly by storing user terminals having the same type in consecutive positions. Hence, in the compressed version the symbol table is divided into slices each of which contains only user terminals belonging to the same type category. Our suggestion of using several type categories and thus, having more slices, increases slightly the number of bit needed for the compressed symbol table. Namely, we have to store information about the boundaries between the type categories. This can be done e.g. by storing the sizes of the slices or by reserving a certain symbol to indicate the boundaries.

The type categories to be considered depend on the programming language in question. Moreover, the use of different grammar transformations may invoke the need of different type categories to be stored in the symbol table. In the following chapter we give a Pascal related example in which syntactic categories label, constant_name, type_name, variable_name, and subprogram_name are used.

3. Eliminating labels from the parse tree

This chapter shows how to use the semantic information stored in the symbol table. We consider the following simple declaration part:

```
program example(input, output, file1);
        label 23, 55;
        const ace = 14;
                header = '52 playing cards';
        type  suit = (diamond, heart, spade, club);
        var card: record x:[2..ace]; y: suit end;
        ...
```

Recall that we store information about the type of user terminals if they belong to the type categories label, constant_name, type_name, variable_name, or subprogram_name. In this example we suppose that user terminals not in the categories mentioned belong to the category 'others'. In the transformed grammar we have the production

<program> → **program** <identifier> '(' <identifiers> ')' <declarations> <subprograms>
begin <statements> **end** .

Since each derivation begins with this production, it is not necessary to store its label. The first user terminal is the name of the program. The following user terminals belonging to the category 'others' are the parameters of the program. It is sufficient to store only pointers to the corresponding symbol table entries. The type category 'others' tells us that the identifier indeed are parameters of the program name. Next we encounter user terminals of type category label. Again no information concerning the productions applied in the left parse are needed. Declarations of constant are also easy to handle (although there are a few alternatives). When an identifier of type category variable_name is encountered, we need syntactic information to handle the various possibilities to define a type. The possibilities are the following: simple type, pointer type, array, packed array, file, set, and record. For each type declaration we have to give a left parse from the nonterminal <type> to a terminal string. Note that this is the first time when labels of productions must be stored in order to be able to reproduce the program. Hence, no syntactic information is needed to handle the nonterminals <identifier>, <identifiers>, and <declarations> in the production above provided that there are no type declarations. If type declarations exist we have to store left parses from nonterminal <type>. If a program has subprograms the above activity can be repeated when handling their declaration parts. If the compression system uses only the type categories mentioned then we have to have some kind of end marker which tells when the parameters of the program name end and the statements begin.

4. Grammar transformations

This chapter discusses grammar transformation that helps us in compression. We consider Pascal expressions as an example. In a normal Pascal grammar there is a large amount of productions providing the proper precedence and associative rules related to these expressions [4]. Katajainen et al.[6,7] replace these productions by the following simple productions

p_1: <expression> → <factor> <operand> <expression>

p_2: <expression> → <factor>

p_3: <operand> → **=** | **≠** | **<** | **≤** | **≥** | **>** | **in** | **+** | **-** | **or** | * | **/** | **div** | **mod** | **and**

p_4: <factor> → <variable> | **not** <factor> | **nil** | unsigned_integer | unsigned_real | string | (<expression>) | identifier (<actual_parameters>) | [<elements>] | [ ].

Some of the nonterminals used (e.g. <variable>) are not explained here, but the reader should obtain a clear overall picture of the nature of changes done. As already mentioned, these changes reduce the size of the parse tree but at the same, they increase the number of production alternatives per a node: the original grammar (see [4]) has separate nonterminals for relational, adding, and multiplying operators but above all operators are gathered to the right hand side of one single production.

Suppose now that we are reproducing the program and we know that p_1 is the next production to be applied. If we in addition know that the next user terminal is of type *boolean*, then we can conclude that the operator must be one of the relational operators (=, ≠, <, ≤, ≥, >, in) or a boolean operator (and, or). Hence, knowing the type of the next user terminal allows us to continue as if the production were

<boolean_expression> → <boolean_factor> <boolean_operand> <boolean_expression>

where only the operators mentioned can be generated from <boolean_operator> and the nonterminals <boolean_factor> and <boolean_expression> are defined correspondingly. Similar case appears if we know that the user terminal is of type real. Then the operators **mod** and **div** are not possible.

The above example shows that type information makes it possible to compress by using virtual productions whose nonterminals have fewer production alternatives than the nonterminals in the original production. This decreases the number of alternatives per a node in the parse tree. Since we use local production labels the increase in the total number of productions has no effect to the compression result; it only makes the compression process somewhat more complicated.

We have already mentioned that it is necessary to parse a program using one grammar and to compress using another grammar. The connection between the two grammars is established by defining a homomorphism from the set of productions of the original grammar to the set productions of the transformed grammar. Type information is also transmitted via this transformation.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.

2. J.L. Bentley, and C.C. McGeoch, Amortized analyses of self-organizing sequential search heuristics. *Comm. ACM* **28**, 4 (1985), 404-411.

3. R.D. Cameron, Source encoding using syntactic information source models. *IEEE Trans. Inf. Theor.* **IT-34**, 4 (1988), 843-850.

4. K. Jensen, and N. Wirth, *Pascal User Manual and Report*. Springer, 1975.

5. J. Katajainen, and E. Mäkinen, Tree compression and optimization with applications. *Intern. J. Found. Comput. Sci.* **1**, 4 (1990), 425-447.

6. J. Katajainen, M. Penttonen, and J. Teuhola, Syntax-directed compression of program files. *Softw. Pract. Exper.* **16**, 3 (1986), 269-276.

7. J. Katajainen, M. Penttonen, and J. Teuhola, A Prolog prototype of a syntax-directed compression system. University of Turku, Dept. of Mathematical Sciences, Report D31, 1988.

8. A. Nijholt, *Context-Free Grammars: Covers, Normal Forms, and Parsing*. Lecture Notes in Computer Science 93, Springer, 1980.

9. H. Peltola, and J. Tarhio, On syntactical data compression. In: Proceedings of the Second Symposium on Programming Languages and Software Tools. University of Tampere, Dept. of Computer Science, Report A-1991-5, August 1991.

10. D.D. Sleator, and R.E. Tarjan, Amortized efficiency of list update and paging rules. *Comm. ACM* **28**, 2 (1985), 202-208.

11. R.G. Stone, On the choice of grammar and parser for the compact analytical encoding of programs. *Comput. J.* **29**, 4 (1986), 307-314.

Psd – a Portable Scheme Debugger

Pertti Kellomäki
Software Systems Lab, Tampere University of Technology
P.O.Box 553, SF-33101 Tampere, Finland
email: pk@cs.tut.fi

April 16, 1993

Abstract

A portable debugger for the Scheme language was implemented. The debugger does not rely on implementation specific features in order to provide debugging capabilities. Instead, the source program is transformed into one that behaves as if run under a conventional debugger.

1 Introduction

The Scheme language, a member of the Lisp family of languages, has been gaining popularity within academia as a teaching vehicle. It is a very simple and compact language, and consequently many free implementations have emerged. However, none of these provide the kind of source level debugging commonly found for the C language, for example.

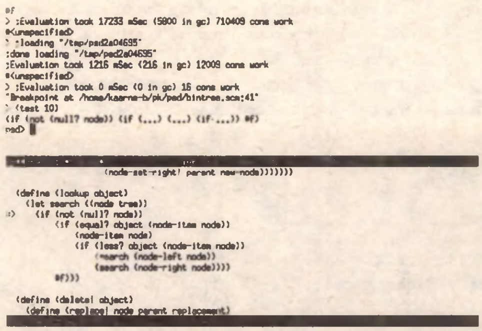
Psd is a portable source level debugger for the Scheme language, that adds debugging capabilities to any implementation conforming to the de facto language standard "Revised^4 Report on the Algorithmic Language Scheme" [1]. Psd provides most of the features commonly found in debuggers. The user can set breakpoints, single step evaluation and examine and change the variables of the debugged program. Psd shows the source line being executed in an editor window. Even programs that cause run time errors can be debugged with Psd, as the debugger intercepts the execution of the debugged program just before a run time error would occur.

Psd works by transforming the original source code into a form that provides the desired debugging capabilities. There is at least one similarly implemented debugger, namely the edebug package for GNU Emacs, written by Daniel LaLiberte.

The pragmatic reason for implementing Psd was to provide better debugging capabilities for students using Scheme in courses taught by the Software Systems Lab.

2 The User Interface

Psd has been designed to be used when running a Scheme interpreter as a subprocess of GNU Emacs (a popular free editor). Psd uses Emacs as a front-end for displaying the current location in source code, as well as for handling details such as temporary file name



```
#f
> ;Evaluation took 17233 mSec (5800 in gc) 710409 cons work
#<unspecified>
> ;loading "/tmp/psd2a04695"
;done loading "/tmp/psd2a04695"
;Evaluation took 1216 mSec (216 in gc) 12009 cons work
#<unspecified>
> ;Evaluation took 0 mSec (0 in gc) 16 cons work
"Breakpoint at /home/kaarne-b/pk/psd/bintree.scm:41"
> (test 10)
(if (not (null? node)) (if (...) (...) (if ...)) #f)
psd> 

                  (node-set-right! parent new-node)))))))

  (define (lookup object)
    (let search ((node tree))
=>    (if (not (null? node))
          (if (equal? object (node-item node))
              (node-item node)
              (if (less? object (node-item node))
                  (search (node-left node))
                  (search (node-right node))))
          #f)))

  (define (delete! object)
    (define (replace! node parent replacement)
```

Figure 1: A Psd Session

generation and issuing commands to the instrumentation code. The user interacts with the debugger and debugged program partly by giving commands via Emacs, partly by typing them directly to the debugger prompt. It is also possible to pick procedures to be debugged directly from an editing window. An example session is shown in figure 1. The debugger communicates with Emacs by printing specially formatted lines containing information about the current source line. Emacs interprets these lines by showing the appropriate file in an editing window, and maintaining an overlay arrow indicating the current line.

GNU Emacs was chosen as a front-end mainly because it is widely used for writing programs, and because a similar interface had already been developed. The Psd interface to Emacs was modified from an existing Emacs interface for gdb (the GNU debugger). It would not be difficult to modify Psd to use a different front-end, though.

3 Debugging by Instrumentation

Debuggers such as dbx on Unix machines usually run the debugged program as a separate process. The debugged program is run just like it would be normally run, and the debugger implements single stepping and breakpoints by using services provided by the operating system. These techniques are somewhat hard to apply in an interactive environment, and they are necessarily implementation dependent.

Many Lisp systems, including most Scheme implementations, implement debugging by stopping the execution of the debugged program, and starting a new read–eval–print loop. The user can then type expressions that the system evaluates, walk up and down stack frames etc. This provides a very powerful debugging environment, as the user can use all the features provided by the underlying language. However, a debugger implemented this way must be built into the implementation. It also seems that while being a powerful tool for the experienced user, this kind of debugger can be very confusing for the novice programmer.

Psd provides debugging capabilities by augmenting the original program with additional code. This is done entirely in source code level, so no special support is needed from the underlying Scheme implementation.

A simple example of a source code transformation providing debugging capabilities is adding code for printing a trace of every procedure invocation. For example, the procedure

```
(define (square x)
  (* x x))
```

could be transformed into

```
(define (square x)
  (display "square called with argument ")
  (display x)
  (newline)
  (* x x))
```

and then loaded into the Scheme environment. If the instrumentation and loading is handled by the programming environment, the user just sees a procedure being traced.

An important point is that the augmented procedure retains its interface and operation, as seen from the calling code. When another procedure calls square, it should not matter, whether the original or the augmented definition was used. Procedures instrumented by Psd retain their interface, which allows Psd to be used as an additional tool in the programming environment, rather than as a separate environment.

4 Transformations Performed by Psd

The transformations Psd performs are similar in spirit to the above example, although more complicated. Extensive instrumentation of the original source code is needed for providing breakpoints and access to variables in the debugged program.

An example of an instrumented procedure is given in figure 2. It is the result of instrumenting the procedure

```
(define (foo x)
  (+ x 42))
```

Psd makes heavy use of the first-classness of Scheme procedures. They are used for accessing local variables, as well as for implementing single stepping and break points.

In order to provide access to the values of local variables from the debugger command loop, two procedures are inserted at the start of each lexical scope in the original program. They provide access to the variables visible within that scope. For example, the procedure `psd-val` in figure 2 returns the value of the variable `x` when it gets the symbol `x` as an argument. The procedure is passed to `psd-debug`, the debugger runtime, as an argument. When the user wants to see the value of the variable `x`, the debugger calls the procedure `psd-val` with the symbol `x`, and gets the value of the variable in the debugged program. Note that since Scheme uses lexical scoping, there is no other portable method for accessing the value of a lexical variable from outside the scope.

```
(define foo
  (let ((psd-context (lambda () (cons (quote foo) (psd-context)))))
    (lambda (x)
      (let ((psd-val (lambda (psd-temp)
                       (case psd-temp
                             ((x) x)
                             (else (psd-val psd-temp)))))
            (psd-set! (lambda (psd-temp psd-temp2)
                        (case psd-temp
                          ((x) (set! x psd-temp2))
                          (else (psd-set! psd-temp psd-temp2))))))
        (psd-debug psd-val psd-set! psd-context
                   (quote (+ x 42)) 1 2 2
                   (lambda ()
                     (psd-apply ((lambda x x)
                                 (psd-debug psd-val psd-set! psd-context
                                            (quote +) 1 2 2
                                            (lambda () +))
                                 (psd-debug psd-val psd-set! psd-context
                                            (quote x) 1 2 2
                                            (lambda () x))
                                 (psd-debug psd-val psd-set! psd-context
                                            (quote 42) 1 2 2
                                            (lambda () 42)))
                                psd-val psd-set! psd-context
                                (quote (+ x 42)) 1 2 2 #f)))))))
```

Figure 2: An Instrumented Procedure

For implementing break points and single stepping, each expression is converted into a *thunk*, a procedure of no arguments. When the value of the expression is needed, the thunk is called, yielding the value of the original expression. This is effectively same as delayed evaluation. Thus, if the expression

```
        (+ x 42)
```

is converted to

```
        (lambda () (+ x 42))
```

it is possible to evaluate its value at a later time. The thunk is passed to the debugger runtime, which can then interact with the user both before and after evaluating the expression.

The same conversion is applied recursively to each of the subexpressions. Instead of using the native procedure calling mechanism, the subexpressions of a procedure call are passed to the procedure psd-apply, which checks that it is safe to call the procedure with the given arguments. This is done in order to catch run time errors. The final result of the conversion is roughly equivalent to

```
        (psd-debug (lambda ()
                     (psd-apply
                      (psd-debug (lambda () +))
                      (psd-debug (lambda () x))
                      (psd-debug (lambda () 42)))))
```

When the debugger decides to run one step of evaluation, it calls the procedure that was passed to it. Calling the thunk invokes each of the calls to psd-debug, yielding the addition procedure, the value of x, and the number 42. The debugger runtime gets control both before and after each subexpression is evaluated. Finally, psd-apply calls the addition procedure with the desired values.

Each time psd-debug is invoked, it checks whether it should continue evaluation immediately, or stop and prompt the user for commands. Single stepping is implemented simply by using a global variable whose value tells whether single stepping is on or not. For break points, a list of source code lines containing the currently active break points is maintained. Each time execution proceeds to a line in the break point list, execution is stopped and the debugger command loop is entered.

As seen in figure 2, the runtime is called with additional parameters besides the debugged expressions. These parameters contain the location in source file, the expression being evaluated and so forth.

5 Run Time Errors

Because Psd relies on correct execution of the instrumented program, it cannot let run time errors occur. For the programmer, though, errors in the program often manifest themselves by causing run time errors. A debugger is commonly used for running the program until a run time error occurs, and examining the state of the program at that point.

Psd catches most run time errors by examining the arguments passed to primitive procedures from the debugged code. If Psd determines that a run time error would result from calling the primitive procedure, the debugger command loop is called instead.

In Scheme, type is associated with a value, not a storage location. All variables, including the formal parameters of a procedure, can hold values of any type. Thus, calling a user defined procedure with arguments of wrong type does not cause run time errors. Type errors can only occur when calling a primitive procedure. Psd detects calling primitive procedures with wrong number or wrong type of arguments, and stops the execution of the debugged program.

User defined procedures can thus be called with arguments of any type. However, trying to call a user defined procedure with wrong number of arguments results in a run time error. In order to prevent that, Psd would have to know the arity of each user defined procedure, which is not possible to determine portably. With some help from the implementation, it would be easy, though.

It would be possible to do at least some static type and arity checking, but most small implementations do not perform it. The flexibility of the language makes static checking difficult, as procedures can be passed around and stored in data structures just as other values.

Other types of run time errors, such as division by zero, are not handled by Psd.

6 Limitations and Further Development

Because the aim of developing Psd was to provide a portable debugger, it cannot use features outside the language standard. Perhaps the most serious single limitation caused by that is that no backtrace information is available in the debugger. It would be possible to pass the backtrace around as an additional parameter, but this would require that all procedures accept the extra parameter. However, backtrace information is very helpful when figuring out what went wrong, so solving the problem is a main aim for further development.

Another concern that becomes apparent when dealing with large applications is space and time efficiency. The instrumented code is much larger than the original source code, expansion factors from 14 to 39 have been observed. Small procedures expand more than larger ones, because there is a fairly constant amount of code associated with each procedure. Thus, the example of figure 2 is a rather extreme case. Instrumentation also slows down the debugged program considerably, observed slowdown has been in the range of 170-200. If an implementation has a compiler, the runtime support code could be compiled. In practice, most of the debugged program is usually run normally, and only a few procedures are being debugged at a time. In this case, the efficiency of a single procedure does not necessarily slow down the whole program significantly.

7 Availability of Psd

Psd is available from the author. It is also placed under anonymous ftp in `cs.tut.fi` in the directory `/pub/src/languages/schemes`, and in the Scheme Repository `nexus.yorku.ca` and its mirroring site `ftp.inria.fr`. Psd is placed under the GNU General Public License, so it can be freely used and distributed.

References

[1] Jonathan A. Rees and William Clinger, editors. The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1992.

Modelling Communicative Strategies

Mare Koit
Tartu University, Department of Computer Science, J. Liivi 2, EE2400 Tartu, Estonia,
e-mail: koit@csd.ut.ee

Haldur Õim
Tartu University, Department of General Linguistics, Ülikooli 18, EE2400 Tartu,
Estonia

Abstract
The paper discusses the problem of how in a dialogue in natural language between two participants one of the participants, using so-called communicative strategies, can influence the other participant to make a certain decision. The model of communicative strategy will be offered which unites the partner model and the reasoning algorithm used in the process of working out a decision.

1. Introduction: a short survey

In order to develop Artificial Intelligence (AI) systems which could adequately understand people and make themselves understood while interacting in natural language it is necessary to interpret human communication and to model it on computers.

Communication is not a simple interaction of speech acts (for example, ask - reply, demand - refuse, agree, etc.) but, as a rule, is an hierarchically organized process. First, certain goals and subgoals will be pursued. Secondly, to achieve the goal certain methods called communicative strategies are applied (e.g. blandishing, frightening, threatening, etc.).

Let us consider interaction between two subjects, $S1$ and $S2$, where the goal of the initiator of the interaction ($S1$) is to achieve a decision by the partner ($S2$) to perform an action (A). $S1$ cannot directly "bring about" this decision, he can only release and influence the decision making process in $S2$.

Therefore, $S1$ must have a general depiction of the partner's reasoning model. The paper presents a formal model which tries to represent the typical trains of thought by the human reasoner. According to this model the reasoning process - whether to perform or not perform an action A - will be released by three kinds of factors, the so-called determinants: 1) wishes of the subject, 2) his considerations of the usefulness and 3) his considerations of the obligatoriness of A. The reasoning process consists of a sequence of steps where the pleasant and unpleasant, useful and harmful, etc. aspects of the action will be weighted. This process is governed by reasoning postulates and principles characteristic of human motivational and reasonig system in general.

On the other hand, we are interested in how the subject $S1$ "can penetrate" the reasoning algorithm of $S2$ so as to direct and influence it. $S1$ can do it only via communication. First of

all *S1* has to choose the starting determinant (wish, needed or must determinant) which he attempts to trigger in *S2*. When it has been triggered, *S1* must see to it that *S2*'s reasoning proceeds through any of those routes which leads to the decision to perform the action A.
Such an influencing process represents the communicative strategy of *S1* with the aim to generate the decision to perform *A* by *S2*. The communicative strategy as well as the reasoning process can be formalised. The paper presents a possible approach to this task.

2. Background and related research

The notion of communicative strategy (CS) is related to the notions of dialogue planning and plan recognition (e.g. [8], [10]). Strategies in our sense represent higher level structures though: a communicative strategy in dialogue is a general line of reasoning, a general basis also for constructing concrete plans.
The notion of CS in discourse is used by such authors as T.A. van Dijk [3] and K.McKeown [9]. T.A. van Dijk defines a strategy in terms of general attitudes which the speaker tries to impose on the recipient.
K. McKeown is interested in so-called discursive strategies the aim of which is to organize the texts in such a way as to make their understanding easier for the receivers.
In conversational analysis and related research the concept of strategy is relatively popular. But as a rule it is related quite straightforwardly to the use of definite categories of verbal expressions and/or to various social aspects of the interaction (see e.g. [4,5] for the illustration). The cognitive aspect, i.e. the attempt to take into account the recipient's processes of reasoning and understanding in which ultimately a strategy should get realised is almost lacking.
Our model of CS differs from these models in some crucial aspects.
First, the aim of introducing this notion is to tie together communicative goals of the speaker, on the one hand, and the partner model structure, on the other. The partner model here is understood as a relatively concrete personality type model.
Second, our model presupposes human reasoning model and takes into account different aspects of personality structure in this respect.
The concept of strategy is also used in the game theory: a strategy is a prescription which determines the behavior of the participant in all situations which may appear during the game. Communication as the interchange of speech acts can be considered as a game in which moves are realised by speech acts of the participants [1]. To a speech act of one of the participants the other participant can react in a finite (and often quite restricted) number of different ways. For instance, an order may be answered by complying, by refusing, by asking for additional information etc. The concept of CS used in the present paper in analogous to the concept of strategy in a game of 2 persons. But the specific characteristics of the human interaction we want to take into account determine the specificity of our concept of CS.

3. Modelling natural reasoning

Let us consider a special case of communication - a dialog between two subjects, *S1* and *S2*. Let the goal of *S1* be that *S2* would agree to perform a certain action *A*. How can *S1* in the course of (verbal) interaction influence *S2* to come to the given decision? Apparently, *S1* must have a depiction of the model of reasoning with which people operate when they are working out their decisions.
We will present here our view of the reasoning model (for the motivation see [7]).

According to our model, the human reasoning concerning an action A may be induced and directed by three kinds of factors, or determinants. These factors are divided into internal and external ones with respect to the reasoning subject S.

The internal determinants of S which may induce the reasoning about A are, first, S's wishes, and second, S's considerations of what needs to be done. We will call the first class of determinants "WISH-determinants" and the second class - "NEEDED-determinants".

The external determinants are the obligations, the norms that force S to perform A without taking into account S's own interests. These are "MUST-determinants".

The WISH-determinants operate when S would find it pleasant to do A for its own sake or for the sake of some of its consequences. This may be characterised as the primary and the most natural motive to do some action: to get satisfaction from it.

The NEEDED-determinants get activated as subgoals of some already accepted goal: in order to reach the goal G the action A as one of its subgoals has first to be realized.

Finally, as MUST-determinants function obligations, norms and also other subjects' orders which hold in the situation where the subject S finds himself.

We are interested in the process of reasoning that leads from these determinants to the decision to perform or not to perform the action A. These processes can be described as proceeding by specific decision steps, but at the same time they follow a certain overall scheme. Nevertheless, in the frames of this scheme the subprocesses are different and differently organised depending on the input determinant type.

Let us briefly discuss the categories that will be used in formulations of reasoning algorithm, and the principles that govern their interactions.

1) PLEASANT/UNPLEASANT: these categories represent the primary (originally emotional) evaluations which are anchored in the sensual system of the subject.

2) USEFUL/HARMFUL: these are prototypical rational evaluations, i.e. they are based on certain beliefs or a certain knowledge of S and there are certain criteria for making the corresponding judgements. These criteria are connected, first of all, with the goals of S: an aspect of A is useful for S if it helps S to reach some goal G; and, correspondingly, an aspect of A is harmful if it prevents S from reaching some goal G.

3) OBLIGATORY/PROHIBITED: these are also rational evaluations but they are based either on the knowledge of certain (social) norms or on some directive communicative act of a person who is in the position (has the power) to exercise his will upon S. Obligations and prohibitions are connected with the concept of punishment, which is an action taken by some other subject as the reaction to S's not following the corresponding obligations or prohibitions. Thus, trhrough this concept, the dimension of obligatory/prohibited is connected with the previous dimensions: a punishment is (is intended to be) unpleasant or harmful to S.

RESOURCES of S with respect to A constitute any kinds of circumstances which create the possibility to perform A and which are under the control of S.

The values of the dimension obligatory/prohibited are in a sense absolute: something is obligatory or not, prohibited or not. But the dimensions pleasant/unpleasant and useful/harmful are, rather, scalar ones: something is pleasant or useful, unpleasant or harmful to a certain degree. We should thus represent these dimensions by certain scales on which the intervals should be differentiated when reasoning.

In the reasoning process their concrete values, or weights, are summed up in some way. Before the final decision about A is made, its pleasant and unpleasant, useful and harmful aspects should be weighed up and the general "balance" of the weightings of positive and negative aspects computed. This suggests that the corresponding scales should be represented in some form which makes the cross-scale comparison possible (in the formal representation e.g. in numeric form, where the use of the concrete numeric values should be empirically grounded, of course).

There exists a natural correspondence between these three dimensions and the three input determinants considered before, i.e. WISH-, NEEDED- and MUST-determinants:

1) a positive value concerning the aspects of *A* on the pleasant/unpleasant scale is presupposed by the WISH to do *A*; 2) a positive value on the useful/harmful scale is presupposed by the NEEDED-determinant; and 3) a positive value on the obligatory/prohibited scale is presupposed by the MUST-determinant.

In the considered context the process of reasoning itself consists in the interaction of the WISH-, NEEDED- and MUST-determinants and the judgements concerning the dimensions pleasant/unpleasant, useful/harmful and obligatory/prohibited.

Beyond these, the reasoning model contains a number of general principles, or postulats, which characterise the human motivational and reasoning system in general (e.g.: people want pleasant states and do not want unpleasant ones; the more pleasant is the imagined future state the more intensively a person strives it, etc.). And there are a number more concrete preference rules, e.g.:

- if *A* has been found pleasant (and also the subject wishes to do it) then the subject checks the NEEDED- and MUST-determinants first from the point of view of their possible negative values ("what harmful consequences *A* would have?");

- if the sum of the values of the inner (WISH- and NEEDED-) determinants and the value of the external (MUST-) determinant appear equal in a situation (i.e. there arises a conflict) then the decision suggested by the inner determinants is preferred.

Let us present now the reasoning algorithm corresponding to three kinds of input determinants: 1) *S* WISHes to do *A*, 2) *A* NEEDs to be done by *S* and 3) *S* MUST do *A*. For representing the concrete aspects of the reasoning process in the algorithm the following abbreviations are used: *W(pleas)* - the weighting of the pleasant aspects of *A*; *W(harm)* - the weighting of the harmful aspects of *A*; etc. All these weightings are given from the point of view of *S* and constitute a model of the reasoning subject.

The reasoning algorithm is universal and does not depend on the concrete subject. We represent the algorithm as a so-called schematic program [6].

```
| -- reasoning
| W(pleas) > W(unpleas)?
| procedure WISH
|-
| W(use) > W(harm)?
| procedure NEEDED
|-
| is A obligatory?
| procedure MUST
|-
| decide not to do A
```

Let us explicate here the contents of the procedure *WISH*, i.e. the reasoning which is triggered by a wish or want of the subject. For the details of other procedures - the reasoning which departs from the considerations of usefulness and obligatoriness of *A*, see [7].

```
-- procedure WISH
-- presumption: S wishes to do A, i.e.
-- $W(pleas) > W(unpleas)$
 are there enough resources for A?
   $W(pleas) > W(unpleas) + W(harm)$?
     is A prohibited?
      $W(pleas) > W(unpleas) + W(harm) + W(punish)$?

     $W(pleas) + W(use) < W(unpleas) + W(harm) + W(punish)$?


     $W(pleas) + W(use) < W(unpleas) + W(harm)$?
     is A obligatory?

     is A prohibited?
     $W(pleas) + W(use) < W(unpleas) + W(harm) + W(punish)$?

  decide: to do A

decide: not to do A
```

4. Communicative space

What we have talked about represents the model of decision making - of reasoning - of a subject S without his/her relations to and communication with other subjects.

But we are interested in a situation where there is still another subject (i.e. there are $S1$ and $S2$) and the subject $S1$ is trying to bring about in the reasoner $S2$ one certain decision concerning A, for instance, to do A.

How can $S1$ "penetrate" the reasoning algorithm used by $S2$ so as to direct it and influence its outcome?

First, $S1$ must choose the starting determinant (WISH, NEEDED, MUST) which he/she must then try to trigger. This input delimits also the set of possible further strategies.

Second, when the chosen input determinant has been triggered, $S2$ must guarantee that $S1$ in his reasoning process moves along one of the routes that brings him/her to the decision "to do A".

The scheme presented above shows the critical points in the reasoning process on which the outcome of the process depends.

But to influence this process - the thinking of $S2$ - subject $S1$ can only by communication with $S2$. And here - in the "communication space" between $S1$ and $S2$ - empirical regularities of its own are at work. There are dimensions and factors that one has to take into account when having in view certain result in a concrete situation of communication.

Let us present first an overview of the dimensions of communication space which are relevant for our discussion, and then to consider the problem of representing the model of communicative strategy.

First we introduce two dimensions of general character which as if are placed "on top of" the other, more concrete dimensions. These are

(1) cooperative-confrontational character of communication and

(2) personality-impersonality of communication.

According to the first dimension the communicative encounters can be ordered into a continuum where at the one end are absolutely confrontational encounters (quarrels) and at the other end interactions without any element of disagreement (e.g. conversation between lovers). The majority of interactions lie somewhere between these extremes, and as a rule the participants themselves choose the concrete value of the dimension.

Personality-impersonality: according to this dimension the speaker chooses, how personally, i.e. departing from himself/herself he/she presents the material. There are types of interaction in which the individuality of the speaker has no real meaning (e.g. some types of official talk) and there are interactions which necessarily require the participation of the speaker as an individual person. And again in the majority of cases the speaker can decide how personally-impersonally he/she acts in presenting his/her communicative acts (e.g. question, offer, proposal, remark, criticism, etc.), and in this way manipulate the possible reactions of the recipient.

The whole diversity of the remaining aspects of communication we will divide between three dimensions:

(3) the distance between participants,

(4) the modal, attitudinal (evaluative) dimension, and

(5) the intensity of communication.

The distance between participants is one of the most widely discussed dimensions of communication, especially in the literature on conversational analysis, e.g. [2]. Distance should be confused with the personal closeness of the participants. Even when communicating with unfamiliar people one should fix a certain distance. It reflects the amount of "common ground", the amount of factors which unite the participants in a concrete encounter. As such, it plays a central role in person-to-person communication. The "shortening" or "widening" of distance with respect of the neutral one by one of the participants is communicatively meaningful. For instance, shortening the distance (by $S1$) can in certain circumstances be considered as "doing a favor" to the partner $S2$ (as, for instance, in case of expressing interestedness and/or positive apraisal towards the partner as a person) and $S1$ is in position to expect the counterfavor (e.g. the fulfilment of a request) by $S2$.

The other two dimensions - modality and intensity of communication - are again of more general nature and are not so relevant for the type of communication we are interested in here. Let us look at them only briefly.

By modality of communication we mean the dimension as the values of which occur such characteristics as friendly, deferential, respectful, attentive, distrustful, unfriendly, careless, rough, irritated, etc. Most of concrete values of the dimension are such that they can be correlated with certain strategies only. For instance, in the case of blandishment or flattery the values of the given dimension (on the part of *S*1) such as "rough" or "careless" are excluded. On the other hand, one may think of strategies where just these values of the dimension are used.

Lastly, by intensity of communication we mean the energy by which it is carried out. An encounter can be peaceful or reserved, or it can be vehement. *S*1 can behave, in presenting his aims, modestly and discretly, or he can behave obtrusively. In the same way *S*2 can vary his reactions.

In sum, the communication space as described above may be represented as a 5-dimensional space where the coordinate axes are fuzzy scales [11].

In order to achieve the aimed decision (by *S*2) concerning the action A, *S*1 should create in *S*2 the right configuration of the evaluations of the pleasant and unpleasant, useful and harmful, prohibited and obligatory aspects of *A*. but to guarantee that the information about the relevant aspects of *A* which *S*1 plans to give to *S*2 would have the right effect, *S*1 should choose, first, the right configuration of the values of communicative dimensions, to move to the right point of the communication space, before (when) giving *S*2 the "objective" information.

5. Communicative strategy

Let us treat a dialog between *S*1 and *S*2. The subject *S*1 has:

1) an understanding of the reasoning algorithm which would be used "by every normal person", that is also by *S*2 (and by *S*1 himself/herself);

2) a picture of the evaluative appraisals of *S*2 of the pleasant, unpleasant, useful, harmful etc. aspects of the action *A*, i.e. a partner model. In general, this model does not necessarily coincide with the model which *S*2 himself/herself operates when reasoning about *A*;

3) a communicative strategy - a method of influencing the reasoning process of *S*2.

For reaching one certain goal several different strategies can be used. For instance, enticing, threatening, frightening, persuading - as a rule, these are not single speech acts but may represent quite long-lasting endlavors of one participant of communication in the names of his communicative goal.

On the other hand, a strategy is not a plan where it is precisely determined what to say and in what order. In dialog it is hard to make detailled plans since one cannot foresee the exact reactions of the partner.

CS in this sense is a structure of higher level than plans. It constitutes a possible basis for concrete plans and for their variations in a session of communication.

CS-s can be classified - and represented - in several different ways; for instance, according to communicative goals. As a goal may function "objective" knowledge of the partner about something; his conviction (that something holds or does not hold); his evaluations (that something is good or bad); his decision to do or not to do something.

The hierarchy of communicative goals determines also the hierarchy of CS-s. The CS-s which are directed towards attainment of one and the same goal may be differentiated according to the input determinant which the author of CS is trying to trigger.

Let us present here a general communicative strategy where the goal of its author (*S*1) is to reach certain decision of *S*2 (to do *A*). The encounter begins with the determination by *S*1 of the concrete manner of influencing *S*2 (i.e. his concrete strategy: blandishing, persuading, frightening etc.), and with the fixation of the co-ordinates of the starting point in the communication space. *S*1 chooses the degree of cooperativity, of personality-impersonality and of intensity of the planned communication, and decides whether the default value of his distance to *S*2 should be changed or not. When *S*1 founds that the distance should be widened

11

or shortened, then he/she should start the encounter with the corresponding introductory remark(s), without necessarily referring to his/her real goal, but using the chosen values of other dimensions.

The actualisation of the point in communication space ("communication point") can be represent by the following scheme.

```
-- actualization of the point in communication space
choose the degree of cooperativity
choose the degree of personality
choose modality
choose intensity
   to change the communicative distance?
      make remark(s) to widen/shorten the distance
```

After choosing the communication point $S1$ has to inform $S2$ of his/her communicative goal. This turn - as well as the following ones - shoud be formulated by taking into account the communication point, i.e. by using the chosen degree of cooperativity etc. If $S2$ agrees, the communicative goal of $S1$ is reached and the communication may end. But if $S2$ refuses then $S1$ has to decide whether to continue the interaction or not. If he/she chooses to continue, he/she should decide, first, whether to follow the chosen strategy or to change it, and look whether to "move" to another point in communication space, for instance, by shortening the communicative distance or by rising the intensity of communication. Further performance of $S1$ depends on the concrete strategy chosen by him/her.

CS can be represented by the following scheme.

```
-- communicative strategy (author S1)
choose a concrete strategy
actualize the (initial) point in the communication space
inform S2 of the communicative goal
   did S2 agree?
      -- the goal is reached
      to give up?
         -- the goal will not be reached
         to change the concrete strategy?
            choose a concrete strategy
         to change the communication point?
            actualize a point in communication space
         apply the concrete strategy
```

The concrete strategies used to reach a fixed communicative goal differ from one another, first of all - as pointed out above -, by induced input determinants. For instance, *S*1 can entice, persuade or force *S*2 to decide to do *A* by inducing or rising in him/her, accordingly, a wish, an understanding of usefulness or an understanding of necessity concerning *A*.

We will confine ourselves here to representing one communicative strategy - the strategy of enticing. The goal of enticing is to induce in *S*2 a wish to do *A*, and by stressing the pleasant (and other positive) aspects of *A* and, at the same time, possibly, by downgrading the negative aspects of *A*, to bring *S*2 to the decision to do *A*. We represent the enticing strategy by a case scheme, where as the key functions the answer of *S*2; the strategy gets started when *S*2 has refused to do *A*.

```
-- strategy of enticing
answer of S2 (refusing)?
- ("no" ∨ "little benefit" ∨ "not obligatory" ∨ "little pleasant")
present a counterargument in order to stress pleasant aspects of A
- ("no resources")
present a counterargument in order to point at the presence of possible resources or at the
  possibility to gain them
- ("much harm")
present a counterargument in order to downgrade
  the value of harm
- ("A is prohibited and the punishment is great")
present a counterargument in order to downgrade the weight of the punishment
- ("much unpleasant")
present a counterargument in order to downgrade the value of the unpleasant aspects of A
```

On the ground of the concrete reply of *S*2 *S*1 is able to make the necessary changes in the partner's model and to understand what route *S*2 is traversing in his process of application of the reasoning algorithm, and to choose the counterargument in order to turn the reasoning process of *S*2 in the needed direction. The contents of the counterargument depend on the concrete action, of course; we don't consider here the question of the verbal formulation of replies either.

We confined ourselves to considering the use of communicative strategies from the point of view of *S*1 only. In reality, of course, *S*2 will use his strategies in the same way.

6. Concluding remarks

Intuitively, we interpret communicative strategies as general methods of pursuing one's goals in dialogical interaction. As examples may serve tempting, soothing, frightening, persuading etc. someone in order to get him to do an action, to believe something, to make a certain value judgement etc.

For the present discussion, we have chosen strategies where the goal of the author of the strategy, *S*1, is to get the partner, *S*2, to do a certain action *A* (more concretely, to make the decision to carry out the action).

The aim of the paper was to demonstrate one possibility for the formalization of the process of reasoning which brings the subject to a certain decision, and of the concept of communicative strategies which can be used to direct the processes of reasoning.

The ideas are implemented in a computer system where the values of the reasoning determinants and of the communicative dimensions can be manipulated in different ways, and the task of the system is to "compute" the decision concerning the suggested action.

The presented models turn out to be useful, in addition to AI, also in linguistics providing a conceptual framework for interpreting individual facts of linguistic communication.

References

1. Auramäki E., Hirschheim R., Lyytinen K. 1992. Modelling offices through discourse analysis: The SAMPO. - *The Computer Journal*, vol. 35, No. 4, pp. 342-352.

2. Brown P., Lewinson S. 1977. Universals in language usage: politeness phenomena. - Goody, Esther (ed.), *Questions of Politeness*. Cambridge: Cambridge University Press, pp. 56-289.

3. Dijk T.A. van 1983. Cognitive and conversational strategies in the expression of prejudice. - *Text*, vol. 3-4, pp. 375-404.

4. Garcia C. 1989. Apologizing in English: politeness strategies used by native and non-native speakers. - *Multilingua*, vol. 8, No. 1.

5. Garcia C. 1989. Disagreeing and requesting by Americans and Venezuelans. - *Linguistics and Education*, vol. 1, pp. 299-321.

6. Kiho J. 1984. Schematic programming (in Russian). - *Transactions of the Tartu University Computing Centre*, Tartu, No. 50, pp. 52-68.

7. Koit M., Õim H. 1990. An approach to the modelling of natural reasoning. - *AIMSA'90*. Varna, Bulgaria, Sept. 1990. Proceedings. Ed. by P.Jorrand and S.Sgurev.

8. Litman D.J., Allen J.F. 1987. A plan recognition model for subdialogues in conversations. - *Cognitive Sciences*, vol. 11, No 2, pp. 163-200.

9. McKeown K.R. 1982. Generating natural language responses to questions about database structure. Ph. D. Dissertation, University of Pennsylvania.

10. Pollack M.E. 1990. Plans as complex mental attitudes. - *Intentions in Communication*. Ed. by Ph.R.Cohen, J.Morgan and M.E.Pollack. Cambridge & London, The MIT Press, pp. 77-103.

11. Zadeh L. 1965. Fuzzy sets. - *Information and Control*, June, pp. 338-353.

A SEMANTIC-SYNTACTIC RECOGNITION SYSTEM BASED ON ATTRIBUTED AUTOMATA

Antti Koski
Department of Computer Science
University of Turku
Lemminkäisenkatu 14 A, 20520 Turku
e-mail: akoski@cs.utu.fi
Finland

ABSTRACT

In this paper we present an idea of combining syntactic and semantic information in the form of attributed automata. Our motivation comes from syntactic pattern recognition of ECG signals. There we need to detect various syntactic structures and calculate also many different numeric values. We have developed a system of attributed automata, which parses an input string according to the system grammar. An attributed automaton is a finite state machine, whose states are augmented with finite number of semantic variables, attributes. Every automaton is designed to parse some specific substring of input and return successfully if the desired structure is detected and the values of the attributes satisfy some predetermined conditions. Parsing with these subgoals needs also a suitable searching strategy, which we have chosen to be depth-first order. We have designed a language of our own for writing system specifications. We have also implemented a pre-processor, which reads the specification and generates necessary C-functions, which need to be compiled together with the base interpreter and this finally gives an executable system parser. An object-oriented view in our system reveals also simple class hierarchy among different automata. Some automata are clearly like specialised versions of other more general types of automata. Our primary goal is to increase the intelligence of the system so that it could acquire more knowledge during parsing processes. This can be achieved through the classification and clustering of detected substructures.

Keywords

Syntactic pattern recognition, Signal analysis, Attributed automaton

1. INTRODUCTION

Pattern recognition methods are widely used in various tasks in computer science today. First approaches were statistic methods, where we calculated quantities from the phenomenon and tried to classify this phenomenon according to these numeric values. However, in many occasions the most important feature of the object is its structure and the way how it is built. This is the idea behind syntactic methods in pattern recognition(1). In syntactic methods we first analyse an unknown pattern and try to extract primitives. Primitives are the most elementary pieces of structure of the object. Primitive extraction has resemblance to the feature extraction phase in statistical methods, but the way we use primitives is different from the way we use numeric features. After this primitive extraction we examine the way they are organised to form the object itself. We have a pattern grammar to show the structure of the desired objects. This grammar has primitives as its nonterminals. Then we find out whether the object is constructed according to the rules in the pattern grammar, i.e. we parse the primitive string to find out if it belongs to the language generated by the given pattern grammar. If this is the case then we classify the object to the class defined by this grammar.

This skeleton has worked well in purely structural classification problems, but in many occasions we need to resort to the semantic meaning of the primitives. This is the case in the recognition of the electrocardiograms (ECG) which are largely employed as a diagnostic tool in clinical practice in order to assess the cardiac status of subjects. Several methods have been used in the recognition and analysing of the ECG signals(2,3). In ECG diagnoses we have to calculate several amplitude and duration values of the patterns in the signal. We could use some kind of feature coding in the primitive extraction phase, but then our primitive alphabet usually grows too large to be practical. If we do the primitive extraction by k features and each feature i is divided to n_i classes then the total number N of primitives is

$$N = \prod_{i=1}^{k} n_i$$

which soon becomes too large in any applications. A better way is to perform the primitive extraction phase by few structural features and leave other features connected to the primitive as they are. These represent the semantic interpretation of the structural primitive. In addition to the coding problem, we need to express sometimes also complex semantic dependencies in the structure of the pattern. If we leave this totally to the pattern grammar, then the complexity of the grammar becomes soon unmanageable, for instance many dependencies in signal processing are context-depending.

Semantic information can be combined with the syntactic information by the means of the attribute grammars. Attribute grammars are context-free grammars, whose nonterminals and terminals are augmented with semantic variables, attributes. If the context-free grammar is right or left linear then we have a regular syntactic structure. This is usually the case in attributed syntactic pattern recognition because the attributes can express the non-regular information.

2. COMPUTATION OF PRIMITIVES

Selection of primitives is an essential point for the syntactic pattern recognition. There are plenty of different ways to compute primitives of ECG, for example. We applied a typical technique of line segments in which the signal is divided into consecutive, almost linear segments of various length. The character of the ECG signal is suitable for the varying length, because there are long flat parts in the signal but also very steep peaks (Fig. 1).

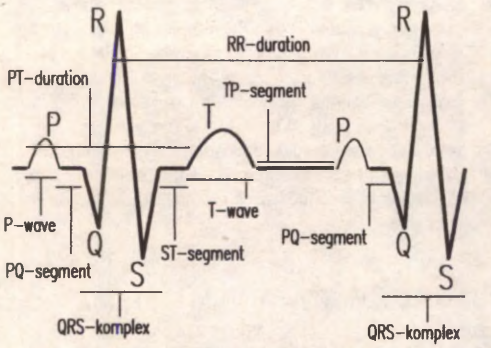


Fig. 1 An ideal model of the one cardiac cycle in an ECG recording.

ECG signal originates from the action of human heart. It describes various phases in cardiac cycles. The most important phase in heart action is the QRS-complex which is the electrical phenomenon of the depolarisation of the ventricles where heart pumps blood to the arterial system. P- and T-waves originate from pre- and post operations to the depolarisation of ventricles. ECG is a very varying signal in nature and continues numerous error components that can distort the signal.

The use of the varying length primitives is effective, since it compresses data and thus decreases the number of primitives. If we applied the fixed length, the chosen length of primitives should be rather small so that even steep QRS-complexes could be detected. Segmentation starts with a fixed length segment which has k sample points. If all sample points between the first and the last point are close enough to the line from the first sample to the last sample then we can add k sample points more to the segment. We enlarge this segment repeatedly by k samples until some sample point is further from the line than it is allowed. Then we start shrinking our segment so that the new end sample point will be set to the sample that lies most far away from the segment line. When finally all sample points lie between allowed limits then we accept this segment and start the next segment from the end of the current segment.

We approximate the slope of each segment by computing the slope of the line between the first and the last sample of the segment. After having computed a segment and its slope we transform the segment to a terminal symbol according to the slope of the segment (Fig. 2). In principle, it is best to prefer as small a number as possible, since we can then construct simpler automata and recognition systems. On the other hand, we can describe more information with primitives when we use a greater number of different terminals. Thus, we have searched for a reasonable trade-off between those two goals. We added also two attributes to each terminal: the dx-duration and the dy-duration.

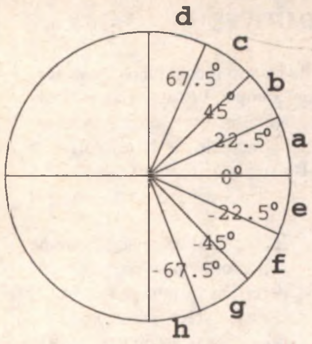


Fig. 2 Primitive extraction

Selection of primitives is one of the most essential parts in syntactic pattern recognition(4,5). For some problems the set of primitives can be determined naturally, as in our application they follow the structure of the signal. There are many applications, for instance EGG processing, where we cannot easily determine the most appropriate primitive set, because there the most important feature is the energy of the EGG in different frequencies. The general rule in primitive selection is that the simpler primitives mean more complex grammars. Augmenting attributes to the primitives reduces the complexity. For instance, in many applications we need to measure the phenomenon itself. We could use fixed length primitives and compare and calculate the number of primitives to get measurement estimates but this leads immediately to context-free grammars and very often over that.

3. ATTRIBUTED AUTOMATA SYSTEM

An attributed automaton is a finite automaton to which attributes are added to contain semantic information collected and used in parsing(7). By employing attributes we make pattern recognition more effective because we enhance the intelligence of the recognition system by considering attributes as semantic information during the parsing process(6). We organised a set of attributed automata for the ECG recognition process. Each automaton has a particular task in the recognition process. There is an initial automaton which starts the whole parsing process and calls some other automata of the set in order to parse some specific substructures. Every automaton can call any other automaton, also itself recursively.

When the call of an automaton is performed, this means that the called automaton commences to process the string of terminals (or primitives in the signal) from the location at which the former automaton called the latter. The called automaton tries to recognise the subpattern. If it succeeds, the calling automaton accepts this result and continues from the location where the called automaton stopped. Otherwise, it fails and calls the next alternative automaton to start again from the same location as the previous automaton. If there is no uncalled automaton left, the automaton moves to the next state if a transition is available. If there is no transition available, the system returns from the automaton to the upper level in the system. The initial automaton halts the process when all terminals of the signal have been considered.

We consider an attributed automaton as a finite automaton with a finite memory for attributes. At any move a new configuration of this memory is evaluated. We defined the system of attributed automata as a tuple of $(A, Q, \Sigma, a_0, Q_0, R, ATT, F, \mu_0)$ in which

1) A is a finite non-empty set of automata.

2) Q is a finite non-empty set of states. Q(a) refers to the states of automaton a. Sets Q(a) are disjoint between the automata.

3) Σ is the terminal alphabet, i.e. collection of primitives that constitutes a pattern string. Our alphabet Σ is common to every automaton in the system.

4) The initial automaton is denoted by a_0, and Q_0 is the set of initial states of the automata. Every automaton has exactly one initial state, $Q_0(a)$ for automaton a.

5) The transition relation R of the system comprises two relations R_1 and R_2. The former is a deterministic partial function from (A, Q, Σ) to Q, which maps transitions within automaton a in A. R_2 is a nondeterministic transition relation (A, Q, Σ, A, Q), which describes actions of the system between automata. Tuple (a, p, c, q) of R_1 is a transition in automaton a from state p to q on input letter c. Tuple (a, p, c, b, q) of R_2 is a call of automaton b from state p of automaton a on input letter c. If automaton b succeeds in parsing, the system returns to state q of automaton a.

6) ATT is a finite set of the attributes of the states.

7) F is a set of the computation rules of the attributes. At each transition which an automaton makes, the attributes of the current state are used to compute the attributes of the state to which the system moves. When the system returns from the subautomaton, the attributes computed by the subautomaton are used to update the attributes of the state to which the system returns.

8) μ_0 contains the initial values of the attributes of the initial state of the initial automaton at the start moment of the system. Thus, attributes of automata represent the semantic information collected from subpatterns already analysed.

4. FUNCTION OF THE SYSTEM

Our basic idea was to construct attributed automata each of which is intended to recognise a certain subpattern. An automaton can then call a certain subautomaton to recognise the corresponding subpattern. The automaton at a higher level may have several alternatives to choose from for some phase of the parsing. Order of calls of the subautomata is arranged depending on the application so that more probable alternatives are used before those which are less probable. The nature of our task in ECG processing is more like analysing signals and not recognising them. Therefore we can first try more specific structures that follow the structure of the ideal ECG signal, although they may not be the most common shapes in ECG signals.

Processing of the automaton stops when there are no alternatives (transitions) to be chosen or the automaton compels a return to the calling automaton which is tested with predicate return. For the latter case there are two possibilities. First, the subpattern which was looked for was already found, although there are still transitions unused. For example, such an event is encountered when we have detected the left side of a peak and we are processing the middle of the right side, and these sides are of equal height at that moment, but the right side would still continue after the midpoint. However, the recognition of that peak is completed at that sample, because we want both sides to be of approximately equal height. Second, the automaton can discover, on the basis of values of the attributes, that the recognition of the subpattern cannot be successful in this choice.

After having returned to the calling automaton the predicate *success* tests by means of attributes whether the subautomaton called could parse the substring, i.e. recognise the subpattern. We could set absolute bounds for such attributes which concern the time domain of the signals. However, those attributes associated with the amplitude relative values are necessary, since for the amplitude attributes we cannot fix any unequivocal values the amplitude scale being so varying.

We define the action of the system with configurations which are tuples of (a, p, w, μ, P). In the tuple a is the current automaton to be processed, p in Q(a) is the current state in a, w is the input string not yet considered, μ is the set of attribute values of p, and P is the control stack of the system. An initial configuration is $(a_0, Q\ (a\ ), w, \mu_0, \varepsilon)$ where w is the whole input string (signal) and ε is the empty stack. There are the following cases for the changes of states:

1. $(a, p, cw, \mu, P) \rightarrow (a, q, w, \nu, P)$ if (a, p, c, q) is in R and ν is equal to $F(\mu)$, i.e. obtained from μ according to computation rules of the attributes. This is a deterministic move inside an automaton a by the input letter c.

2. push action: $(a, p, cw, \mu, P) \rightarrow (b, Q_0(b), cw, \nu, (a, p, cw, \mu, q)||P)$, where $||$ is the concatenation operation, if (a, p, c, b, q) is in R and ν is obtained by initialising the attributes of the initial state of b with μ. This is the nondeterministic call of a subautomaton, where backtracking information is stored to the stack

3. pop action: $(a, p, cw, \mu, (b, q, z, \nu, r)||P) \rightarrow (b, r, cw, \kappa, P)$ if *success*(a) is equal to true and either there are no transitions unconsidered in automaton a or *return*(a) is equal to true. We compute κ from ν, which is updated with μ. This case is the successful return where we have detected the subpattern denoted by automaton a.

4. pop action: $(a, p, cw, \mu, (b, q, z, \nu, r)||P) \rightarrow (b, q, z, \nu, P)$ if *success*(a) is equal to false and either there are no transitions unconsidered in automaton a or *return*(a) is equal to true. This is the unsuccessful return where we could not parse the subpattern denoted by automaton a. Note that the attributes ν of the calling automata are not altered. This means that we take no information of unsuccessful return, we only notify it. We could use the information somehow to evaluate attributes ν in order to find out what caused the failure. This would be one place to insert automatic learning capabilities by allowing metaprocessing where automaton rules are altered by another automaton rules.

While being in a state the system may have several alternatives to call automata. At the call of a subautomaton information about the parsing situation is stored in the stack. This includes the name of the current subautomaton, state, input string not yet processed, attribute values, and the return state which is used after the successful return. After having pushed them to the stack the system calls the subautomaton. If the parsing fails, the system returns to the calling automaton and the next subautomaton is tried. So the action of the system is to perform depth-first search in a set of possible parsing orders. As mentioned above, the order of the tries is arranged appropriately in our application. In addition to subautomata calls, the system can make a transition by input letter after trying every possible subautomaton, which is deterministic, and then no return information is stored.

When the whole input string has been considered, the system has the final configuration (a, p, ε, μ, P) where ε is the empty input string. If automaton a is not then a_0, the lowest record has to be fetched from the stack. This stack record is always a configuration of a_0. Attribute values of a_0 express semantics of the input string.

Our system can be viewed also as a specialised version of a recursive transition network system where we have semantic attributes and conditional predicates and evaluation functions augmented to the states (Fig. 3).

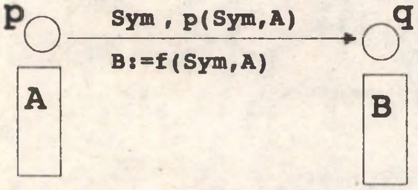


Fig. 3 A transition model in automata system.

Here we can move from the state p to the state q if we can detect the structure Sym, and the structure Sym and the attribute values of the state p satisfy predicate p. Sym can be a structural symbol in the case we start a subautomaton and see if the result satisfies predicate p. If Sym is a terminal symbol in the parsing string then the predicate is T, because the detection of one specific terminal symbol is sure. After a successful move we update the attributes of the state B with the evaluation function f. If the parsing of the structure Sym fails then we try another alternative move from the state p. If there is none left then the work of this level network is finished.

Attribute grammars contain very much so called copy-rules where we just copy attributes from one state or nonterminal to another and do nothing else. This is especially the case in attributed automata system where we have encapsulated one task in one automaton. That is why we have taken the attributes out of the states and put them together with an automaton. We can namely assume that all the states have the same attributes.

5. SYSTEM SPECIFICATION LANGUAGE

We have implemented our system with our own application language where we can determine all the actions of the automata system. All the application specifications are written in the following syntax.

```
[ START <automaton>
  ATTRIBUTES
  <definition of attributes>
   [ INITIALIZE [ <automaton> ]
     <initialising of attributes> ]+
   [ OUTPUT
     <output actions> ]
   [ TRANSITION
     <state> ' [ <terminal> ]+ ' ->
        [ CALL <automaton> <state>
          <evaluation of attributes> ]*
        [ MOVE <state>
          <evaluation of attributes> ] ]+
  STOP <automaton> ]+
[ FUNCTIONS
  <definitions of functions> ]
```

This definition is then pre-compiled with a pre-processor. The definition of the attributes, initialisation and evaluation follow the syntax of the language C. In **INITIALIZE**-blocks we define the actions which take place when some other automaton calls this automaton. There we use the symbol @ when we refer to the attributes of the current automaton and symbol $ when we refer to the attributes of the calling automaton. **OUTPUT**-block is performed when this automaton has finished its actions successfully. However, here we have the problem of destructive operations. Although this automaton succeeds, the upper level automaton that has called this automaton may fail and then the work done by this automaton is wasted and must be forgotten and if we have done some changes in global data those changes must be undone. We are trying to avoid this kind of backtracking allowing only operations to the local data of the current automaton.

In **TRANSITION**-blocks we define the actions of the current automaton. In **CALL**-statements we define the order of subautomata tries. When a subautomaton returns successfully then we can evaluate the attributes of the current state, to which we refer with a symbol @, by the attributes of the called automaton, to which we refer with the symbol $. In a **MOVE**-transition we can refer to the attributes of the current automaton with the symbol @. In **FUNCTIONS**-block we can define various evaluations functions in the language C. These functions can be used in transitions, initialisations and returnings of automata.

6. ECG APPLICATION

We have designed an automata system of fourteen automata to perform ECG analysis. We have named automaton *EKG* to be the initial automaton which starts the process and collects various average information out of the signal. Automaton *BEAT* parses one cardiac cycle at a time (Fig. 4). It tries to find out how the cycle is constructed by using several other automata who are specialised to certain substructures. Each state in *BEAT* corresponds to a certain phase in cycle parsing where we try to parse appropriate structures. If a substructure was detected then we copy important semantic substructure values to the attributes of *BEAT*. If we could not detect any substructures then we move one input letter ahead (denoted by letter x in Fig. 4) and try again to parse interesting substructures.

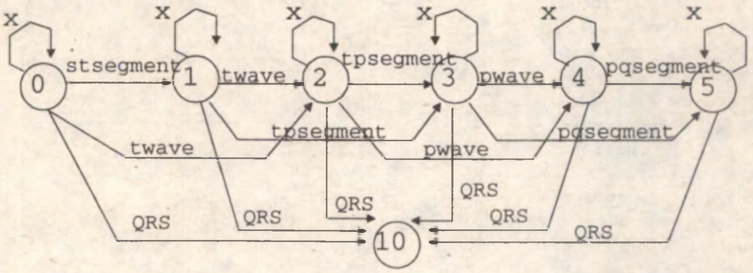


Fig. 4 Automaton *BEAT*.

Automata *STSEGMENT*, *TPSEGMENT* and *PQSEGMENT* parse flat parts of the signal. They use automaton *SEGMENT* to parse a flat structure. *SEGMENT* uses recursively itself to detect the end point of the signal. Automaton *TWAVE* parses the T-wave and *PWAVE* parses the P-wave (Fig. 5 and 6). They both have approximately the same syntactic structure, but their semantic conditional predicates use different thresholds and formulas in the accepting of the substructure. Their parsing is successful if the amplitude of the left and right arm are close to each other and the amplitude is a certain part of the average primitive height and the width of the wave is a certain part of the amplitude of the wave and the flat top area of the wave is not wider than half of the total width. In addition, in the detection of the P-wave we have to define the accepting of the P-wave so that no other P-wave structures lie between the detected P-wave structure and the next QRS-complex. This means that we accept the last P-wave structure as a real P-wave. Automaton *QRS* parses a QRS-complex. It is successful if the maximum height of the QRS-complex is more than c times the average primitive height where c is some predetermined constant (usually about five), and the start and the end point are close in amplitude scale. Automaton *QRS* uses two other subautomata to detect steep slopes of the QRS-complex. They accept also slight errors in these arms of the QRS-complex where the edges of the arms can have small distorted noisy primitives.

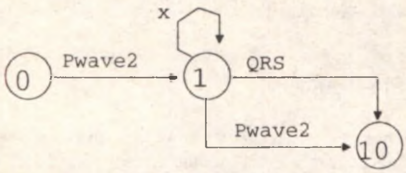


Fig. 5 P-wave automaton.

We can strengthen or weaken these conditions depending on the parsing result. Usually constant threshold values are not useful because ECG is so varying a phenomenon. This is where we need capabilities of metaprocessing where we can adapt to input string in according to the analysing goals. In ECG processing we aim to detect correctly as many subpatterns as possibly. Correctness can be achieved through the context-dependent conditions and the checking of our analysing result. This increases our time-complexity because we have to do more searching in order to find the best analysing result.

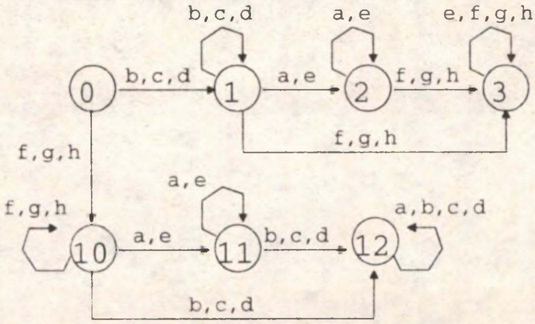


Fig. 6 The structure of the P-wave (Automaton *Pwave2*).

The results of our system were encouraging. The system detected almost all of the QRS-complexes and most of the P- and T-waves in the set of 42 test signals. Our signals were recorded with a heart monitor and they contained relatively large amount of noise. We examined also our backtracking efficiency by calculating a ratio of primitives in input string to the total numbers of move-transitions made by the system. This ratio was between 0.5 and 0.9 and most often about 0.8. So we can state that on the average 20% of our application work is wasted because of unsuccessful choices. If all our test signals would be almost ideal then this efficiency ratio would be near one.

7. CLASS HIERARCHY OF AUTOMATA

We have detected a form of class hierarchy among our application for ECG signals. For instance P- and T-wave automata are two subclasses from the more general wave-shape automaton class. Similarly automaton *QRS* could be divided into more specific classes of QRS-morphology. We can say that in the most general form an automaton can accept anything, i.e. Σ^* and calculates something about this universal signal string like average dx- and dy-values and the number of primitives. Then we can specialise this model by adding more transitions and evaluation rules and specialising our predicate from true to something else more restricting predicate. To be more specific we can define the class hierarchy of automata as follows.

An automaton class AC_L is a set of automata that accept the language L. Class AC_K is a subclass of AC_L iff $K \subseteq L$. This definition is based on the set of strings accepted by automaton, but if we have an automaton A as an instance of class AC_L and B as an instance of class AC_K then we can study the structural differences between A and B and the subclasses generated by certain structural operations. Basically, there are infinitely many ways to make structural changes to automata but we were interested only in two following.

1) Restricting our predicate p in certain transition in automata system, i.e. replacing predicate p_1(A,Sym) with p_2(A,Sym) where p_1(A,Sym) is a logical consequence of p_2(A,Sym). This kind of operations could be used in thresholding problem.

2) Changing the structure Sym_1 to the Sym_2 where $Sym_2 \subseteq Sym_1$. These operations are useful in the QRS-complex classification and clustering.

These are found by examining the application structures of several signal analysis tasks. They both restrict the set of recognised strings. They are used in automata definition to make it more compact and easier to use. We can define a suitable class hierarchy where we collect the common structure to the upper level automata and just specialise this definition to be suitable for some particular substructure.

Those operations could be used also in automatic information acquisition. Especially important is the thresholding problem. We can easily describe the absolute structure of a subpattern but relative restrictions to the subpattern depends on the context. For example in ECG processing we need to reject small waves in order to avoid noise waves to be recognised as P-waves. But the next recording can be almost error free and P-waves are very small maybe because the electrodes are settled so that the electrical fields diminish P-waves. There we need to lower our threshold values and change our automaton to another automaton class. We could adapt to the current signal type by calculating the distance of the current signal from some template signal types. However, here we have the problem of finding only the first solution in depth-first search. We namely have to find also other solutions, i.e. subpatterns and examine also them. We choose the most suitable one from the set of found subpatterns. This technique seems to lead to membership degree calculations and fuzzy decision making. This area is currently under research.

8. SUMMARY

On the basis of our tests and experience, attributed automata appear to be very suitable for the syntactic pattern recognition problems. Using attributes we can semantically control the syntax analysis of input strings. A pure formal grammar or automaton cannot express all kinds of dependencies effectively, because those may be context sensitive. With attributes we have added semantic features to the recognition, and thus increased the general recognition efficacy of the system.

REFERENCES

1. A. Cohen, Biomedical signal processing vol. II. CRC Press, Boca Raton, Florida (1986).

2. E. Skordalakis, Syntactic ECG processing: a review, Pattern Recognition 19, 305-313 (1986).

3. P. Trahanias and E. Skordalakis, Syntactic pattern recognition of the ECG, IEEE Trans. Pattern Anal. Machine Intell. 12, 648-657 (1990).

4. M. Juhola, A syntactic method for analysis of saccadic eye movements, Pattern Recognition 19, 353-359 (1986).

5. M. Juhola, A syntactic analysis method for sinusoidal tracking eye movements, Comput. Biomed. Res. 24, 222-233 (1991).

6. M. Juhola and M. Meriste, An attributed automaton for recognising of nystagmus eye movements, IAPR Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland (1992). Also in Advances in Structural and Syntactic Pattern Recognition (ed. H.Bunke), World Scientific, Singapore 1992, pp. 194-203.

7. M. Meriste and J. Penjam, Attributed finite automata, International Workshop on Compiler Construction CC '92, Report 103, Univ. of Paderborn, 40-51, 1992.

An Implementation of ASN.1 (Abstract Syntax Notation One)

Jukka Paakki, Kari Granö
Department of Computer Science, University of Jyväskylä
P.O.Box 35, SF - 40351 Jyväskylä, Finland
email: {paakki,grano}@cs.helsinki.fi

Ari Ahtiainen, Sami Kesti
Nokia Research Center
P.O.Box 156, SF - 02101 Espoo, Finland
email: {aanen, kesti}@rc.nokia.fi

Abstract

ASN.1 (Abstract Syntax Notation One) is a protocol engineering language used in specifying on an abstract level the messages transmitted in computer network communication. The language is associated with encoding rules that specify in which binary form the actual concrete data values are represented in a physical medium during transmission. Both the language and its encoding rules have been standardized by ISO, and currently a revised standard is under development by a joint committee of ISO and CCITT. An implementation is presented that translates a specification given in ASN.1 into a set of data structures and encoding/decoding functions in C. Using these data structures and functions, network applications can realize the exchange of their communication data. The central features of ASN.1 are presented, and the problems in automatically processing the language are discussed. The presentation covers both the ASN.1 language defined in the presently valid standard as well as the forthcoming extensions.

1. Introduction

In the last decades, one of the most important areas in computer science has been the research on *computer networks*. A computer network is a collection of interconnected computers that communicate via some form of transmission lines. In order to understand and synchronize the messages sent within a network, the computers must follow a common set of rules, *a communication protocol*. The key aspect in building a computer network is to specify and implement the protocol of communication between the involved computers.

Modern advanced networks are rather complex and large systems. *OSI* (Open Systems Interconnection) is the most well-known reference model designed for reducing this inherent complexity by introducing a layered structure on the network architecture. The OSI model has seven layers: (1) the physical layer, (2) the data link layer, (3) the network layer, (4) the transport layer, (5) the session layer, (6) the presentation layer, and (7) the application layer. Each of these layers has a standardized abstract functionality, and the only form of communication is across a narrow interface between neighboring layers. It is only the application layer that provides services to the actual application, most notably a primitive for sending data from the application to another application running in another machine. This data (with some protocol control information) is transferred from layer (7) to layer (6) after executing layer-specific operations, from layer (6) to layer (5), etc., until the lowest layer (1) in the hierarchy is reached. The physical layer finally transmits the data to the target machine through a physical medium. The process is reversed in the target machine where the data reaches the receiving application through layers (1), (2), ..., (7). For extensive introductions to OSI, see e.g. [22] or [24].

The research on computer networks has generated a rich set of methods and tools to set up a network. On the communication software side the term often used for the developed methodology is *protocol engineering*. The term characterizes the view that in the current state-of-the-art even complex protocols can be really engineered from specification right down to implementation. The foundation to the discipline is laid by a number of formal protocol specification languages and their implementation tools. The leading standardized OSI oriented specification languages are SDL [2], Estelle [4], and LOTOS [3]. The protocol engineering discipline is discussed in full e.g. in [11] and in [17].

Layer (6) in the OSI model, the *presentation layer*, is responsible for the syntax of the data transmitted in the network. The main task of the layer is to encode messages such that the physical (binary) representation sent can be correctly interpreted by all parties of the communication. The key to the problem of representing, encoding, transmitting, and decoding a message is to have a way to describe the data structures composing the message. The method must be both flexible enough to be useful in a wide variety of applications and standard enough to be commonly understood. As part of the OSI development, one such language for describing structured data has been developed. This language, *ASN.1* (*Abstract Syntax Notation One*), has been adopted as the specification tool of data in virtually all application and presentation layer standards of OSI.

ASN.1 is a language specifying structured data types and their values on an abstract level. With ASN.1, the protocol designer can describe the relevant data without having to consider its physical representation. The bit stream actually transmitted is defined for the protocol implementer by the set of *basic encoding rules* (BER). ISO (the International Organization for Standardization) has standardized the ASN.1 language [12], as well as the basic encoding rules [13]. An informative tutorial on ASN.1 is given in [18] and in [23].

The fact that makes ASN.1 a most valuable language in protocol engineering is that it can be automatically implemented. A number of ASN.1 based software tools exist that can assist in implementing and testing the data transmission part of communication protocols. One such tool is *CASN* (Compiler for ASN.1) [20] which can translate a specification written in ASN.1 into data structures and encoding/decoding functions in C. Using the data structures and their associated encoding functions the application can transmit abstract ASN.1 values in a concrete from to the receiver application which can catch them using the corresponding data structures and decoding functions, also generated by CASN.

Besides CASN, a number of ASN.1 implementations have been developed, including e.g. [5], [7], [19], and [21]. All these translate ASN.1 specifications into data structures and encoding/decoding routines written in some programming language. Because of the complex nature of ASN.1, all the translators work in a multi-pass manner, i.e., they process an input several times when producing the target code. All the systems implement only a subset of ASN.1. Most notably the controversial macro facility is either totally left out or it is provided only in a limited form as a number of built-in macros. All these ASN.1 implementations have been designed within a more general frame of protocol engineering, and therefore each of them provides some form of an interface to other protocol production tools.

With respect to the related systems, CASN is quite advanced from a number of viewpoints. CASN has been developed in close co-operation with both users and developers of other protocol engineering tools, and it is therefore flexible in its interactions with external systems. CASN has been applied in many industrial computer communication projects that have introduced into it a number of special facilities tuning it for specific tasks and target environments. The most notable design principle has been keeping CASN up-to-date with respect to revisions of the ASN.1 language. Virtually since the release of the first ASN.1 standard [12, 13], a group formed collaboratively by ISO and CCITT (Comité Consultatif de Téléphonique et Télégraphique) has been working on a revised version of the language. The CASN development team has played an active role in this standardization committee by constantly verifying the practicality of the proposed new ASN.1 features by at least sketching their implementation with CASN. This effort has not only contributed to having the extensions realistic from an implementation point of view, but also to keeping CASN consistent with the forthcoming standard.

In this paper we present ASN.1 and CASN, not from the protocol engineering perspective, but rather from a language and implementation perspective. That is, we consider ASN.1 as an application oriented special-purpose language and discuss implementation problems caused by some of its peculiar features. We proceed as follows: In Section 2 ASN.1 is briefly introduced. In Section 3 we describe the basic functionalities of CASN. The problematic features of ASN.1 with respect to implementation are discussed in Section 4. We conclude in Section 5 by discussing the current and future development of both ASN.1 and CASN.

2. ASN.1 - An Overview

Consider the typical computer communication scheme sketched in Figure 1. An application (or a user) on machine *A* wants to send data to another application (user) on machine *B*. In order to do this, (1) the data must be stored in the local representation form of machine *A*, (2) the local *A*-representation must be encoded into an external form used during transmission, and finally (3) the receiver on machine *B* must decode the incoming data and transform it into the local representation of machine *B*.

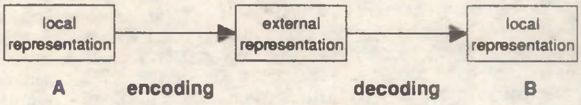


Figure 1. Message communication.

The communication process may get laborous since the data may be complex, since its representation on machine *A* can be quite different from that on machine *B*, and since the software realizing the communication must ensure that the data is not lost and that the meaning of the data is not changed during transmission. Furthermore, the concrete implementation of the communication process is different for each different pair <*A*,*B*>, even though the protocol pattern may be quite similar in many cases. Thus, when changing one of the machines *A* and *B*, the communication scheme would have to be completely reimplemented.

To overcome these problems, the international consultative committee CCITT defined the ASN.1 (Abstract Syntax Notation One) data description language in connection with the X.409 mail standardization activity [6]. Using ASN.1, the data conveyed from machine *A* to machine *B* can be defined on an abstract level, without having to consider the particular machine specific representations. The language was accompanied by a set of rules that specified how the abstract ASN.1 values shall be represented in concrete binary form during physical transmission. Thus, the encoding rules relieve the protocol designers of defining the external form of messages. The language soon gained popularity within the protocol community to the extent that both ASN.1 and its encoding rules have been standardized by ISO in 1987 [12, 13]. Example protocols standardized using ASN.1 include the electronic mail handling system X.400, the X.500 directory, and the OSI presentation protocol.

ASN.1 provides facilities to define primitive and structured data *types* and *values*, in the same principal style as ordinary programming languages. The main differences to type mechanisms of programming languages are due to the application area of ASN.1: a field within a structured value may for instance be missing, or it can have a default value. The most striking difference to programming languages is that ASN.1 includes no control statements: the sole purpose of the language is to provide notations for specifying the structure of network communication messages. The only actual functionality associated with the messages, encoding / decoding, is implicitly defined by the basic encoding rules of ASN.1.

ASN.1 includes a set of built-in *simple types*, and a number of *structured types* composed of other types. The simple types are the following:

INTEGER	NumericString
REAL	PrintableString
BOOLEAN	TeletexString (synonym: T61String)
ENUMERATED	VideotexString
NULL	VisibleString (synonym: ISO646String)
BIT STRING	IA5String
OCTET STRING	GraphicString
OBJECT IDENTIFIER	GeneralString

The types **INTEGER** (for integer numbers), **REAL** (for real numbers), **BOOLEAN** (for truth values), and **ENUMERATED** (for named enumeration values) are conventional. The **NULL** type (with one single value, also called **NULL**) represents missing information, **BIT STRING** stands for binary sequences, **OCTET STRING** for octet (8-bit byte) sequences, and **OBJECT IDENTIFIER** for "information objects" each of which has a unique value of the type. The "information objects" are frequently referenced entities (such as protocol standards, companies, or even persons) that in this way can be globally identified. The types NumericString, ..., GeneralString represent different kind of character strings; VisibleString for instance accepts only visible ASCII characters in its values, while IA5String accepts all the ASCII characters.

The structured types of ASN.1 are defined in terms of other types, their component types. The structured types are the following:

SET
SET OF
SEQUENCE
SEQUENCE OF
CHOICE
ANY

SET represents unordered sequences ("sets") of values, each having a type of its own. **SET OF** also represents unordered sequences, but now the values are of the same type. **SEQUENCE** and **SEQUENCE OF** are similar to **SET** and **SET OF** respectively, but the values in the sequence must be in a specific order. **CHOICE** is a collection of alternative types any of which can serve as the type of a CHOICE value (as the union type in e.g. C). **ANY** is a "universal" ASN.1 type, representing any value of any type. A component of a **SET** or **SEQUENCE** type can be denoted as optional, or it can be associated with a default value. Being optional, the component can be totally omitted in the corresponding structured value. A component with a default value can also be omitted in the value notation, but then the default value is implicitly included in the conveyed message.

In addition, ASN.1 includes some "useful types" that could be defined in terms of ordinary ASN.1, but that are given predefined names because they are frequently used. The useful types are "GeneralizedTime" and "UTCTime" representing time and date, "EXTERNAL" representing references to other ASN.1 specifications, and "ObjectDescriptor" giving a textual representation for OBJECT IDENTIFIER values.

ASN.1 is a modular language, a module being a collection of type and value definitions. Each value has a type-specific notation. A simple example of an ASN.1 module is given below:

```
Library DEFINITIONS ::=
BEGIN
      MyLibrary              ::= SET OF BookInformation
      BookInformation        ::= SET {
              author         PersonalName DEFAULT ernie,
              title          [1] PrintableString,
              code           [2] ISBN-code OPTIONAL
              }
      PersonalName           ::= SEQUENCE {
              surName        PrintableString,
              givenName      PrintableString OPTIONAL
              }
      ISBN-code              ::= NumericString (SIZE(1..10))
      ernie PersonalName     ::= { surName "Hemingway", givenName "Ernest" }
END
```

Module *Library* defines four types (*MyLibrary*, *BookInformation*, *PersonalName*, *ISBN-code*), and one value (*ernie*) of type *PersonalName*. When transmitting a *MyLibrary* value, a sequence of book descriptions of type *BookInformation* is sent (in any order). Each book description is composed of the book's author indication (field *author*), the book's title (field *title*), and the book's ISBN code (field *code*) which can be left out (***OPTIONAL***). The fields can be defined and transmitted in any order. The author is indicated by the family name (field *surName*) and an optional first name (field *givenName*). If the sender does not explicitly supply an author indication, the protocol assumes the author to be *ernie* with first name *"Ernest"* and with family name *"Hemingway"*. A type *T* can be *subtyped*; that is, a new type can be derived from *T* by restricting its value set. In this example, a value of type *ISBN-code* may contain at most 10 numeric characters.

Tagging is a special property of ASN.1 types. Each type (whether built-in or user-defined) has an associated tag (an integer value) which is included in the external binary representation of the transmitted values. The purpose of the tag is to notify the receiver about the type of the value, especially in those cases where the type is not unambiguously specified by the value itself. In the example above, the *title* field of a book description is defined to have tag 1 and field *code* to have tag 2; otherwise the receiver would not know whether the incoming string "123" is a book's title or an ISBN code.

The basic encoding rules (BER) of ASN.1 define how the values specified in ASN.1 must be coded during transmission. The encoding of a value consists of (1) the type tag, (2) the value's length indication, and (3) the actual value. If the value is structured, each of its components is recursively such a triplet. Each element of the encoding is an integral number of octets. For instance, given the definition

```
      v INTEGER ::= 51
```

the BER-encoding of value v is $\underline{02}\mathbf{01}33_{16}$ where $\underline{02}$ is the **INTEGER** tag, **01** is the length of the value (in octets), and 33 is the value in hexadecimal.

The most controversial feature in ASN.1 are the *macros*. Using macros, a protocol designer can *extend* the base ASN.1 language by introducing new type and value notations that suit better for the particular application domain than the standard ASN.1 notations. Macros can also be used e.g. for grouping a set of related abstract types and values, and for semi-formally specifying dependencies between components of structured values.

The widespread use of ASN.1 in protocol engineering has revealed a number of notable shortcomings in the current standard [12, 13]:

(1) The standard is erroneus or ambiguous in many respects. For instance, the scope and type compatibility rules of ASN.1 are not clearly stated. This has resulted in differences both in use and in implementations of the language.

(2) The language is hard to analyze automatically.

(3) The macro facility is hard to understand and implement. An analysis of existing standards employing the macro facility has shown that they have used the facility in a rather diversified manner, and even erroneously.

(4) The standard introduces too many (eight) character string types, some of them being too limited and some too general. Moreover, the compatibility between different string types is unclear.

(5) The basic encoding rules are too primitive in some cases, and they produce too long encodings with a large number of redundant bits.

Because of these problems, a joint activity by ISO and CCITT is currently under way on revising ASN.1 and its encoding rules. The standardization group tackles all the troublespots mentioned above (1) by rewriting the standard, (2) by improving the parsability of ASN.1, (3) by replacing macros with more restricted *information object classes*, (4) by introducing a universal character string type that is a supertype of the current character string types, and (5) by defining alternative encoding rules e.g. for supporting packed encodings. The new standard, currently in a DIS (Draft International Standard) phase, is expected to be released in 1993.

3. CASN - A Compiler for ASN.1

CASN [20] is an implementation of ASN.1. As an ordinary compiler, CASN translates its source language (ASN.1) into a target language (C). The application area of CASN is taken into account by providing for the user a standard run time package (1) for embedding the generated encoding/decoding utilities into a complete protocol implementation, and (2) for integrating the protocol with the application. The overall scheme is illustrated in Figure 2.

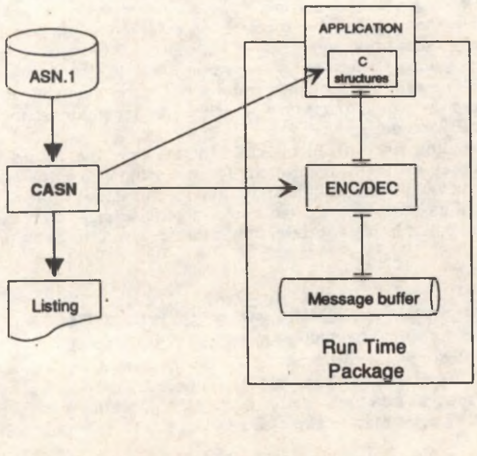


Figure 2. The CASN compiler.

Figure 2 sketches protocol implementation using the *stand-alone* version of CASN. The compiler has also been integrated with a general protocol engineering environment [1] that most notably includes, in addition to CASN, a system for defining a protocol as an extended finite state machine, and a protocol test system. CASN has been applied for example in implementing the X.400 electronic mail protocol, the X.500 directory, the CMISE network management protocol, and the FTAM file processing protocol.

Each type *T* defined in an ASN.1 program is compiled by CASN into the corresponding C type, called the *home type* of *T*. Encoding and decoding of ASN.1 values is achieved by generating for each home type an *entry function* in C. Each entry function is complemented by an *interface function* for integrating the encoding/decoding routines with the application and with the protocol driver. In the following list we give the home type for each built-in ASN.1 type:

ASN.1 type	Home type	
INTEGER	**short** (range -32767..+32767)	
	long (otherwise; default)	
REAL	**double**	
BOOLEAN	**char** (0: FALSE, otherwise: TRUE)	
ENUMERATED	**short** (range -32767..+32767)	
	long (otherwise)	
NULL	**char** (value never assigned)	
BIT STRING	**struct** { **long** length; bits; }	(*)
OCTET STRING	**struct** { **long** length; octets; }	(*)
String types	**struct** { **long** length; characters; }	(*)
OBJECT IDENTIFIER	**struct** { **int** length; **int** elem [MAX_ID]; }	
SET	**struct**	
SEQUENCE	**struct**	
SET OF	**struct** { **long** length; elements; }	(+)
SEQUENCE OF	**struct** { **long** length; elements; }	(+)
CHOICE	**struct** (**union**)	
ANY	pointer to value of some home type	

Depending on the used compiler options and on the defined size constraints, the ASN.1 string types (marked (*) above) can give rise to different concrete data structures. In short: if the values of a string type are of moderate length, a static array is used; if the values are of moderate length which, however, may significantly vary, a dynamic list is used; if the strings may get very long, a segmented dynamic list is employed. The same principle holds for the list types as well (marked (+) above).

The latest release (1.42) of CASN implements most of the 1987 ASN.1 standard. The most significant features not supported are the general macro facility (however, a number of frequently applied macros are provided as built-in), and some subtyping mechanisms. These shortages are going to be removed in the next release of the compiler currently under development. That version is based on the forthcoming 1993 standard where macros are replaced with a more advanced mechanism (see Section 4); the replacement makes macros useless in the future and therefore they will be excluded even from the subsequent releases of CASN.

The compiler is organized into three passes: (1) lexical analysis, parsing, and construction of an intermediate tree, (2) semantic analysis, and (3) code generation. This multi-pass solution is mostly due to the obscure nature of ASN.1 as a language (see Section 4). Another reason is that the first pass has been produced with the parser generator MIRA [9] which does not provide any advanced support for multi-pass compilation. The new macro-replacing mechanism introduces yet another pass into the next release (see Section 4), making CASN eventually a four-pass compiler. CASN is written (partly generated) in C, and the compiler runs both under UNIX and under MS-DOS.

4. Implementing ASN.1

ASN.1 is not the easiest language to implement. The deepest reason to this originates from the design process of the language. An ancestor of ASN.1, known as X.409, was developed by a committee in connection with standardizing electronic mail systems. Implementation matters have had a low priority in the design of both X.409 and its successor ASN.1 and that is why these languages can hardly be considered as masterpieces in the ranking of programming and specification languages.

During the development of the CASN compiler, several troublespots have been encountered. One of the sources is the current standard [12] which is in many respects insufficient for an ASN.1 implementer. The main reasons, however, lie in the language itself which contains a number of bizarre features that cannot be implemented with conventional techniques. Our decision to base the first pass of the compiler on the LL(1) parser generator MIRA has also turned out to be short-sighted with respect to later developments.

In the following subsections we present more closely the major troublespots in implementing ASN.1 and the way we have solved them in CASN. The discussion concentrates on the new standard of ASN.1 (which includes the same basic features as the original ASN.1) and on its implementation (CASN version 2.0), except where otherwise stated. When applicable, we also discuss alternative, conceptually cleaner implementation possibilities. Some of the problems have been reported also in [15].

Defects in the standard

The current standard of ASN.1 [12] is imperfect with respect to the quality requirements on programming language definitions. The most serious defect is the omission of exact visibility and type rules of ASN.1. For example, the standard does not touch the following (perfectly legal) situation where *idA* is overloaded:

```
TypeA         ::= INTEGER  { idA(0) } -- idA is a named number with value 0
idA TypeA     ::= 1                   -- idA is a defined value with value 1
idB TypeA     ::= idA                 -- 0 or 1?
```

In CASN, such ambiguity of visibility is resolved in favor of named numbers, and thus in the example *idB* stands for the value 0. This solution is going to be adopted in the forthcoming ASN.1 standard where also other precise visibility rules are going to be given.

Compatibility rules are given in the standard imprecisely. In essence, it remains open whether *structural equivalence* or *name equivalence* is applied in the type system of ASN.1. The effect of tagging and subtyping on type compatibility is also not explicitly defined. Thus, the standard does not tell whether or not *TypeA* and *TypeB* are compatible in the following example, and whether or not the value definition for *valB* is legal:

```
TypeA         ::= [0] INTEGER (0..100)    -- an integer subrange with tag 0
TypeB         ::= [1] INTEGER (0..50)     -- an integer subrange with tag 1
valA TypeA    ::= 0
valB TypeB    ::= valA
```

Our solution in CASN is to apply a modified form of name equivalence where tagging, subtyping, and referencing (i.e., renaming) have no effect on type compatibility. Therefore, the value definition above is legal. This solution has been included in the draft proposal of the forthcoming ASN.1 standard as the framework on compatibility issues [14].

Free definition order

In ASN.1 the definitions can be given in any order. As in the case of ordinary programming languages, this is nice for a protocol designer but it makes things harder both for an implementer and for a reader of a lengthy specification. The problem is exceptionally severe in ASN.1 where new syntactic notations (defined by macros and their prospective replacement) can be used before introducing them. The CASN compiler is in part produced with the parser generator MIRA with a conceptually underlying one-pass (L-attributed) grammar model, and therefore no support for multi-pass compilation is provided by the system. The problem has been solved with an explicit pass-wise organization of the compiler. A multi-pass compiler generator would provide some assistance in processing ASN.1, but even such a semantically powerful system would fall short in coping with the extensible syntax. Some concrete problems discussed in the sequel are connected to the free definition order principle of ASN.1.

Syntactic ambiguity

The context-free grammar of ASN.1 is ambiguous for several constructs. For instance, the following piece of code can be parsed in two ways:

```
a B ::= c d E ::= F g H ::= ...
```

If *B* stands for a CHOICE type and *c d* for a CHOICE value, we get the following interpretation (each definition on a line of its own):

```
a B   ::= c d
E     ::= F
g H   ::= ...
```

On the other hand, if *c* is a value reference, *E* an ANY type, and *F g* an ANY value, we get the following interpretation:

```
a B   ::= c
d E   ::= F g
H     ::= ...
```

This particular case, as well as other similar ambiguities, is caused by the lack of delimiters in ASN.1. For instance, no delimiter (such as a semicolon) is used between definitions. The solution in CASN has been to introduce explicit delimiters into ASN.1, where necessary. Thus, the first alternative above has to be expressed in the form

```
a B ::= [ c d ] E ::= F g H ::= ...
```

and the second alternative in the form

```
a B ::= c d E ::= [ F g ] H ::= ...
```

Note that the symbol table cannot be consulted to aid parsing because of the free definition order in ASN.1.

The syntactic ambiguities are going to be resolved in the forthcoming ASN.1 standard by the same principle as in CASN: by introducing explicit delimiters into the problematic constructs.

Identifier representation

It is customary to apply the "longest match" principle in scanning of programming languages; that is, the scanner tries to maximize the length of the current token by reading characters from the input as long as they form a valid prefix of some token. This principle cannot be universally applied on ASN.1 where (1) identifiers may contain hyphens (only one in succession and not as the last character), and (2) comments begin with two hyphens. Thus, in the following *a-b-c* is an identifier and *--d* is a comment:

```
a-b-c--d
```

The scanner of CASN (generated by MIRA) makes use of a buffer of 2 characters which is employed when recognizing identifiers; in the example above, the buffer (containing "--") is consulted after processing *a-b-c* to check whether or not the identifier continues. A methodological solution would be use a "trailing context" facility (in the style of e.g. Lex [16]) when defining the lexical structure of ASN.1 but unfortunately this feature is not available in MIRA.

Extensible syntax

Macros can extend the core syntax of ASN.1 by defining new syntactic notations for type and value definitions. This is one of the reasons why CASN currently does not implement the general macro facility. In the forthcoming standard macros are going to be removed from the language but the extensible syntax principle will remain.

In the new version of ASN.1, application elements can be described with *information object classes*. A class definition may give, besides the structure of the associated *objects*, also a class-specific syntax for defining them. For instance, the following class definition can be given:

```
C ::= CLASS {
        &T1  OPTIONAL,
        &T2  OPTIONAL,
        &T3  }
        WITH SYNTAX  {
        [ T1  IS  &T1  ]
        [ &T2  ]
        T3  IS  &T3  }
```

Here *&T1*, *&T2*, and *&T3* are type fields ("open types") that represent arbitrary information to be filled during transmission. The ***WITH SYNTAX*** clause gives the syntax for defining objects of class *C*, for example:

```
c C ::= { T1 IS INTEGER T3 IS SET OF INTEGER }
```

In an object definition, the type fields of the class can be associated with an arbitrary ASN.1 type (e.g., **INTEGER** for *&T1*, and **SET OF INTEGER** for *&T3* above). Each case within a ***WITH SYNTAX*** clause conceptually corresponds to a context-free production that extends the core ASN.1 syntax, [...] denoting optionality. Thus, the example above gives rise to the following additional productions to the ASN.1 grammar:

```
Object_definition      -> '{' P1 P2 'T3' 'IS' Type '}'
P1                     -> 'T1' 'IS' Type | Empty
P2                     -> Type | Empty
```

Type represents an arbitrary ASN.1 type notation and *Empty* represents the empty string. The additional productions are distinguished during parsing from their first terminal symbol (*T1* or *T3* above). The syntactic extensions must follow the LL(1) convention, and this is checked by CASN.

Recently methods have been suggested to parse extensible languages (e.g., [8], [10]). In these approaches the parser is dynamically adjusted during parsing to accept new syntactic notations when recognizing their grammatical definitions. These methods, while being conceptually elegant, cannot be applied on ASN.1 where new notations can be used *before* they have been defined. Therefore our solution has been to to divide the parsing process into two phases. In the first phase the parser processes its input only partially, collecting the syntactic definitions into a tree and simply skipping the object definitions (and other syntactically similar constructs) and storing them in an intermediate form. In the second phase the object definitions are finally parsed, taking into account the additional productions collected during the first phase.

Currently, CASN provides scanning, parsing, and semantic analysis of the new features, and the code generation phase is under implementation. Since an object set is conceptually analogous to an *associated table*, CASN will generate C arrays for object sets (one row for each object).

Infinite lookahead

Recognizing some constructs may require an infinite lookahead in top-down parsing. Consider, for instance, the following notation in extended ASN.1:

```
obj { p1, p2, ..., pn }.&f1.&f2. ... .&fm.&f
```

Here *obj* is a parameterized object, *p1*, ..., *pn* are actual parameters, and the *&f1*, ..., *&fm* fields are object references. This construct may denote e.g. a type or a value, depending on the last field *&f*. Our solution is to parse such a construct in a very general manner, to build an intermediate descriptor for it during parsing, and to make an additional traversal over the descriptor after parsing the whole construct. In a case like this, a bottom-up parsing technique would be more powerful than the top-down method because then the recognition of the syntactic structure could be deferred until having scanned the last symbol. (Note, however, that even a conventional bottom-up parser would have problems since the syntax of ASN.1 is not of type LR(1).)

While CASN is based on LL(1) parsing, it occasionally makes use of a lookahead longer than 1 symbol for recognizing some ASN.1 constructs. For instance, "{ i" can be the beginning of a number of value notations. In such cases the scanner of CASN provides the parser with an extended lookahead that contains the distinguishing token. In maximum, 2 tokens are needed in the first parsing phase and 7 tokens in the second phase; thus CASN locally employs LL(2) and LL(7) parsing.

Parameterization

There are many situations in protocol engineering where it is useful to design simultaneously a set of related types or values that are similar in structure but that differ in details. The new version of ASN.1 supports this by allowing *parameterized definitions*. Each entity of ASN.1 (type, value, value set, class, object, object set) can be generically defined by associating with it a list of formal parameters. Such a generic entity can then be instantiated by fixing the structure with actual parameters. This facilitates reusability of ASN.1 definitions.

The orthogonality of parameterization and the free definition order make it hard to parse entities with actual parameters. That is why detailed syntactic analysis (and semantic analysis) of such constructs is moved in CASN from the parsing phase into subsequent phases. The analysis of some constructs may require several partial passes even when making use of the symbol table, because actual parameters may involve semantic right-to-left dependencies:

```
T { v: U, ..., U }      ::= SET { f U DEFAULT v }
                                        -- v and U formal parameters:
                                        -- U a type or a class,
                                        -- v a value or an object
S                       ::= T { {1,2}, ..., SET OF INTEGER }
```

Now, there is a semantic linkage between the formal parameters *U* and *v* of *T*. When analyzing the definition of *S*, the actual parameter list has to be traversed from right to left (or several times from left to right which is actually the method applied since parsing is also involved).

Context-sensitive name environments

Some keywords of ASN.1 (e.g., *EXTERNAL*) are not reserved, and some have a special meaning only in a context. For instance, *iso* stands for value 1 in the first position of an object identifier value, while it is undefined elsewhere. Such anomalies complicate the analysis of ASN.1, and that is why all the keywords are universally reserved in the CASN release 1.42. Version 2.0 allows the redefinition of keywords (excluding the reserved words) by considering them as ordinary predefined entities that are always implicitly imported from a standard module.

Module interfaces

An ASN.1 module can provide entities to other modules and use entities defined in other modules with exporting and importing clauses. The default actions, however, are rather surprising: (1) if no export clause is given, *all* the symbols defined in the module are externally available, and (2) if no import clause is given, an entity *e* defined in *any* module *M* can be referenced with *M.e*. These (and other strange conventions) cause difficulties in the implementation since the module dependencies may be spread all over an ASN.1 program. Modules can be circularly dependent; that is, a module *M1* can import an entity from module *M2* that (directly or indirectly) imports an entity from module *M1*.

The separate compilation method in the CASN release 1.42 does not accept mutual dependencies between two modules, but the dependencies must have a linear order. This practical inconvenience has been removed in version 2.0 that during parsing of a module *M* collects a list of modules that *M* has applied (either via an explicit import clause or via external references). After compilation of *M*, the modules in the list are compiled, the list is iteratively updated with modules that are applied in these modules and that have not yet been compiled, etc. The syntax trees for compiled modules are stored on disk with a timestamp to be directly loaded without reparsing. The external references are solved using module specific symbol tables that are produced during the semantic analysis phase on the basis of the syntax trees. During reference resolving, circular dependencies between two ASN.1 entities are checked (note that two modules may be mutually dependent, but their components cannot be circularly defined).

5. Discussion

Computer communication will have a still increasing significance in the future because of distribution and globalization of working environments. Thus, application oriented methods and tools will be necessary in building complex computer networks and especially their software.

We have presented CASN, an application oriented tool designed for implementing the encoding and decoding routines of communication protocols. CASN is a compiler for ASN.1 which has a stable and evolving role within the area. CASN can be used both as a self-standing tool and as part of a general protocol engineering environment. Currently the compiler is already in extensive industrial use, and it has become a standard tool in a number of companies.

In the near future, the current version of ASN.1 is going to be replaced with a new standardized one. The main motive for language revision is to remove the serious shortcomings from it. From an implementer's point of view the revision is most valuable since many unclear features will be more accurately specified or replaced with their more "friendly" counterparts. The development of CASN follows closely the standardization efforts. The analysis phase of the forthcoming features has already been implemented in CASN, and the synthesis phase is currently under implementation. This up-to-date nature has been made possible by the active role of the CASN development team in the joint standardization group of ISO and CCITT.

References

[1] Ahtiainen A.P., Keskinen J., Simolin J., Tarpila K., Turunen J.: Protocol Software Engineering Tools for Implementation of a General Purpose OSI Stack. In: *Proc. of the IFIP TC6 Conf. on Computer Networking (COMNET'90)* (L.Csaba, T.Szentivanyi, K.Tarnay, eds.), Budapest, 1990. Elsevier Science Publishers (North-Holland), 1990, 337-351.

[2] Belina F., Hogrefe D.: The CCITT Specification and Description Language SDL. *Computer Networks and ISDN Systems* 16, 4, 1989, 311-341.

[3] Bolognesi T., Brinksma E.: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14, 1, 1987, 25-59.

[4] Budkowski S., Dembinski P.: An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems* 14, 1, 1987, 3-23.

[5] Caneschi F., Merelli E.: An Architecture for an ASN.1 Encoder/Decoder. *Computer Networks and ISDN Systems* 14, 2-5, 1987, 297-303.

[6] CCITT Red Book: Recommendation for X.409, Message Handling Systems: Presentation Transfer Syntax and Notation. International Telecommunication Union, 1984, 62-93.

[7] Cleghorn C., Ural H.: ASNST: an Abstract Syntax Notation-One Support Tool. *Computer Communications* 12, 5, 1989, 259-265.

[8] Christiansen H.: The Syntax and Semantics of Extensible Languages. Report No. 14, Department of Computer Science, Roskilde University, 1988.

[9] Expert Software Systems n.v. (E2S): *MIRA, User's Manual*, 1984.

[10] Heering J., Klint P., Rekers J.: Incremental Generation of Parsers. *IEEE Transactions on Software Engineering* 16, 12, 1990, 1344-1351.

[11] *IEEE Software* 9, 1, 1992. Theme Issue on Network Protocols.

[12] Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1). ISO International Standard 8824, 1987.

[13] Information Processing - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). ISO International Standard 8825, 1987.

[14] Kesti S., Paakki J.: Revised ASN.1: Type Compatibility, Assignment Compatibility, Subtype Compatibility, New Syntactic Notation. COMPET Report #11, Nokia Research Center, 1991. (Also published by: CCITT SG VII DAF Working Group , 1990.)

[15] Kesti S.O., Rönkä K.T.: Use and Applicability of ASN.1. In: *Proc. of the IFIP TC6 Second Int. Workshop on Protocol Test Systems* (J. de Meer, L.Mackert, W.Effelsberg, eds.), Berlin, 1990. Elsevier Science Publishers (North-Holland), 1990, 39-50.

[16] Lesk M.E.: Lex - A Lexical Analyzer Generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[17] Liu M.T.: Protocol Engineering. In: *Advances in Computers*, vol. 29 (M.C.Yovits, ed.), Academic Press, 1989, 79-195.

[18] Neufeld G., Vuong S.: An Overview of ASN.1. *Computer Networks and ISDN Systems* 23, 5, 1992, 393-415.

[19] Neufeld G.W., Yang Y.: The Design and Implementation of an ASN.1-C Compiler. *IEEE Transactions on Software Engineering* 16, 10, 1990, 1209-1220.

[20] Nokia Research Center: *CASN - Compiler for ASN.1: User's Reference Manual*, 1991.

[21] Ohara Y., Suganuma T., Senda S.: ASN.1 Tools for Semi-Automatic Implementation of OSI Application Layer Protocols. In: *Proc. of the Second Int. Symposium on Interoperable Information Systems (ISIIS'88)* (H.Tanaka, A.Tojo, eds.), IOS & OHMSHA, 1988, 63-70.

[22] Rose M.T.: *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, 1990.

[23] Steedman D.: *ASN.1 - Tutorial & Reference*. Technology Appraisals Ltd., 1990.

[24] Tanenbaum A.: *Computer Networks, 2nd ed.* Prentice-Hall, 1988.

SPECIFYING A TRANSACTION MANAGER USING TEMPORAL LOGIC

É. Rácz
L. Eötvös University, Budapest

There are a lot of papers dealing with specification of concurrent systems [1],[4],[6],[7], but few of these are applied for specification of database management systems. In this paper a transaction management system is described as a network of processes which are connected by named synchronous communication channels.

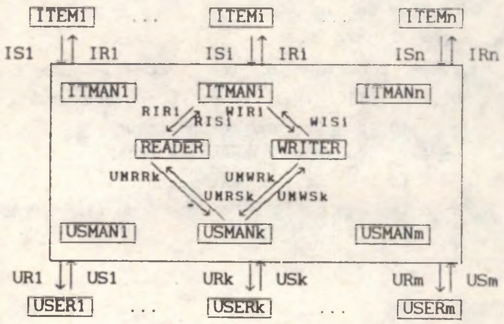
1. THE STRUCTURE OF THE SYSTEM TO BE SPECIFIED

The transaction management system consists of the following parts:

- the user transactions USER1, USER2, ..., USERm;
- the items ITEM1, ITEM2, ..., ITEMn, i.e. the units of data to which the access is controlled;
- the transaction manager TM itself.

A transaction is a sequence of elementary steps. Locking, reading, unlocking and writing items are the elementary steps interesting us.

The transaction manager affects its outside world /the transactions and the items/ only by communications. The named communication channels are unidirectional and owned by two processes, one at each end.



The user USERk sends his various requests along the channel USk, and waits for the answers of the transaction manager at the channel URk. The item ITEMi waits for the requests of the transaction manager along the channel IRi, and sends its answers along the channel ISi.

The transaction manager TM is responsible for scheduling the conflicting requests. Messages relating to reading and read locking of items are sent to/from READER, messages relating to writing and write locking of items are sent to/from WRITER along the appropriate channels.

Chaochen's specification technique [5] has been chosen to describe the behaviour of the transaction manager. We assume that only the channels of the system are observable, at each moment of the discrete time. We can either observe passing of messages along the channels, or the capability of the processes to communicate.

Let c denote a channel, m a message and M_c the message type of channel c. Let us use the following basic predicates:

PASS(c,m)- means the occurrence of an event of passing message $m \in M_c$ along channel c;

PASS(c) - $= \exists m \in M_c$ PASS(c,m);

To record the state of the channel c we associate two basic variables to c, one for the input end - I_c, and one for the output end - O_c. These variables may take the following values:

$m \in M_c$ - the process is receiving or sending a message along c;

req - the process is requesting a communication;

rej - the process is rejecting the communication;

clo - the process closed the channel end.

Using these variables we can write:

PASS(c,m) $\equiv I_c = m \wedge O_c = m$

For describing the behaviour of the system we use a linear time temporal logic [8],[10],[11]. The temporal operators used are as follows:

$\Box A$	--- A is true now and in the future;
<>A	--- A is true now or sometime in the future;
$\circ A$	--- A is true at the next moment;
A until B	--- A is true until B becomes true (if ever);

$\circ x$ --- denotes the value of x at the next time point.

In order to shorten our specification we can introduce the following predicates :

ISREQ(c) $\equiv$ I_c=req until PASS(c)

OSREQ(c) $\equiv$ O_c=req until PASS(c,m)

IWREQ(c) $\equiv$($\Box$<>I_c=req) until PASS(c)

OWREQ(c) $\equiv$($\Box$<>O_c=req) until PASS(c,m)

UWANT(URk) $\equiv$ $\Box$ [OSREQ(URk,m) $\Rightarrow$ <>PASS(URk,m)]

IWANT(IRi) $\equiv$ $\Box$ [OSREQ(IRi,m) $\Rightarrow$ <>PASS(IRi,m)]

Let P denote the following operator: A P B $\equiv$ $\neg$ ($\neg$A until B).

2. EXTERNAL BEHAVIOUR OF THE TRANSACTION MANAGER

In the course of the specification the free variables of each formula are assumed to be implicitly universally quantified over the following (fixed, finite) domains:

USERk	ranges over the set of users;
ITEMi	ranges over the set of items;
m	ranges over the set of messages corresponding the channel;
URk	ranges over the set of channels leading from TM to users;
USk	ranges over the set of channels leading from users to TM;
IRi	ranges over the set of channels leading from TM to items;
ISi	ranges over the set of channels leading from items to TM.

The different message types are as follows:

M_{USk}={ (rlock, ITEMi), (wlock, ITEMi),
(read, ITEMi), (write, *value*, ITEMi),
(runlock, ITEMi), (wunlock, ITEMi) }

M_{URk}={ (rlock granted, ITEMi), (wlock granted, ITEMi),
(rlock denied, ITEMi), (wlock denied, ITEMi),
(not read, ITEMi), (got, *value*, ITEMi),
(written, ITEMi), (not written, ITEMi),
(runlocked, ITEMi), (wunlocked, ITEMi) }

M_{IRi}={ (*value*, USERk), (read, USERk) }

$M_{ISi}=\{\ (\mathit{value},\mathrm{USERk}),\ (\textbf{written},\mathrm{USERk})\ \}$

We require the messages to be uniquely identified. This requirement can be expressed in the following form:

$\Box\,[\mathrm{PASS}(c,m) \Rightarrow \bigcirc\Box\neg\mathrm{PASS}(c,m)]$ for every message m and channel c.

2.1. External safety properties

The transaction manager must not grant any lock request for an item write locked by an other user and must not grant a write lock request for an item read locked by an other user:

$$\Box \left\{ \begin{array}{l} \mathrm{PASS}(\mathrm{URk},(\textbf{wlock granted},\mathrm{ITEMi})) \Rightarrow \\ \neg\ [\ \mathrm{PASS}(\mathrm{URj},(\textbf{wlock granted},\mathrm{ITEMi})) \lor \\ \quad \mathrm{PASS}(\mathrm{URj},(\textbf{rlock granted},\mathrm{ITEMi}))\ ] \\ \textbf{until}\ \mathrm{OSREQ}(\mathrm{URk},(\textbf{wunlocked},\mathrm{ITEMi})) \end{array} \right\} \quad j\neq k$$

$$\Box \left\{ \begin{array}{l} \mathrm{PASS}(\mathrm{URk},(\textbf{rlock granted},\mathrm{ITEMi})) \Rightarrow \\ \neg\ \ \mathrm{PASS}(\mathrm{URj},(\textbf{wlock granted},\mathrm{ITEMi})) \\ \textbf{until}\ \mathrm{OSREQ}(\mathrm{URk},(\textbf{runlocked},\mathrm{ITEMi})) \end{array} \right\} \quad j\neq k$$

A request should be sent before it could be fulfilled or denied:

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{rlock},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{rlock granted},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{rlock},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{rlock denied},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{wlock},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{wlock granted},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{wlock},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{wlock denied},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{runlock},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{runlocked},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{wunlock},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{wunlocked},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{read},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{got},\mathit{value},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{read},\mathrm{ITEMi'}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{not read},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{write},\mathit{value},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{written},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{write},\mathit{value},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{not written},\mathrm{ITEMi}))\ ]$

$\Box\ [\ \mathrm{PASS}(\mathrm{USk},(\textbf{read},\mathrm{ITEMi}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{ISi},((\mathit{value},\mathrm{USERk}))]$

$\Box\ [\ \mathrm{PASS}(\mathrm{ISi},(\textbf{written},\mathrm{USERk}))\ \mathbf{P}\ \mathrm{PASS}(\mathrm{URk},(\textbf{written},\mathrm{ITEMi}))\ ]$

□ [PASS(USk,(**write**,*value1*,ITEMi))
P PASS(IRi,((*value2*,USERk)) ∧ *value1*=*value2*]

Before sending an item to the user the manager must have got it:

□ [PASS(ISi,(*value1*,USERk))
P PASS(URk,(**got**,*value2*,ITEMi)) ∧ *value1*=*value2*]

The user should have a read lock granted before getting an item:

□ [PASS(URk,(**rlock granted**,ITEMi)) P PASS(URk,(**got**,*value*,ITEMi))]

□ [PASS(URk,(**runlocked**,ITEMi)) ⇒
o ¬PASS(URk,(**got**,*value*,ITEMi))
until PASS(URk,(**rlock granted**,ITEMi))]

Similarly for write requests:

□ [PASS(URk,(**wlock granted**,ITEMi)) P PASS(URk,(**written**,ITEMi))]

□ [PASS(URk,(**wunlocked**,ITEMi)) ⇒
o ¬PASS(URk,(**written**,ITEMi))
until PASS(URk,(**wlock granted**,ITEMi))]

2.2. External liveness properties

To have a useful system we must require that the system accepts the requests of its environment supposing that the environment is accepting the messages too. Moreover the item managers must fulfill the requests of the transaction manager:

UWANT(URk) ⇒ □<>IWREQ(USk)

IWANT(IRi) ⇒ □<>IWREQ(ISi)

□ [PASS(IRi,(**read**,USERk)) ⇒ <>PASS(ISi,(*value*,USERk))]

□ [PASS(IRi,(*value*,USERk)) ⇒ <>PASS(ISi,(**written**,USERk))]

The transaction manager should answer the user requests:

□ { PASS(USk,(**rlock**,ITEMi)) ⇒
<> [OSREQ(URk,(**rlock granted**,ITEMi)) ∨
OSREQ(URk,(**rlock denied**,ITEMi))] }

□ { PASS(USk,(**wlock**,ITEMi)) ⇒
<> [OSREQ(URk,(**wlock granted**,ITEMi)) ∨
OSREQ(URk,(**wlock denied**,ITEMi))] }

```
□ { PASS(USk,(read,ITEMi)) ⇒
                           <> [OSREQ(URk,(got,value,ITEMi)) v
                               OSREQ(URk,(not read,ITEMi)) ] }
□ { PASS(USk,(write,value,ITEMi)) ⇒
                           <> [OSREQ(URk,(written,ITEMi)) v
                               OSREQ(URk,(not written,ITEMi)) ] }
□ [ PASS(USk,(runlock,ITEMi)) ⇒ <>OSREQ(URk,(runlocked,ITEMi)) ]
□ [ PASS(USk,(wunlock,ITEMi)) ⇒ <>OSREQ(URk,(wunlocked,ITEMi)) ]
□ { PASS URk,(rlock granted,ITEMi)) ⇒
        o[ PASS(USk,(read,ITEMi)) ⇒ <>PASS(URk,(got,value,ITEMi)) ]
  until OSREQ(URk,(runlocked,ITEMi)) }
□ { PASS URk,(wlock granted,ITEMi)) ⇒
        o[ PASS(USk,(write,ITEMi)) ⇒ <>PASS(URk,(written,ITEMi)) ]
  until OSREQ(URk,(wunlocked,ITEMi)) }
```

3. INTERNAL BEHAVIOUR OF THE TRANSACTION MANAGER

The message classes of the internal channels are as follows:

```
MUMWSk = { (wlock,ITEMi),              MUMWRk= {  (wlock granted,ITEMi),
           (write,value,ITEMi),                   (written,ITEMi),
           (wunlock,ITEMi) }                      (wunlocked,ITEMi),
                                                  (wlock denied,ITEMi),
                                                  (not written,ITEMi) }

MUMRSk = { (rlock,ITEMi),             MUMRRk = {  (rlock granted,ITEMi),
             (read,ITEMi),                        (got,value,ITEMi),
             (runlock,ITEMi) }                    (runlocked,ITEMi),
                                                  (rlock denied,ITEMi)
                                                  (not read,ITEMi) }

MWISi  = { (wlock,USERk),              MWIRi  = {  (wlock granted,USERk),
           (write,value,USERk),                    (written,USERk),
           (wunlock,USERk) }                       (wunlocked,USERk),
                                                   (wlock denied,USERk),
                                                   (not written,USERk) }
```

M_{RISi} = { (**rlock**, USERk), (**read**, USERk), (**runlock**, USERk) }

M_{RIRi} = { (**rlock granted**, USERk), (**got**, *value*, USERk), (**runlocked**, USERk), (**rlock denied**, USERk), (**not read**, USERk) }

In order to specify the internal safety properties we define m auxiliary variables [9] for each ITMANi process. The variable $LockTablei_k$ describes the rights of USERk regarding ITEMi.

$LockTablei_k = 0 \wedge$
$\Box$ { PASS(RIRi, (**rlock granted**, USERk)) $\wedge$ $\circ LockTablei_k = R$
$\vee$ PASS(RIRi, (**runlocked**, USERk)) $\wedge$ $\circ LockTablei_k = 0$
$\vee$ PASS(WIRi, (**wlock granted**, USERk)) $\wedge$ $\circ LockTablei_k = W$
$\vee$ PASS(WIRi, (**wunlocked**, USERk)) $\wedge$ $\circ LockTablei_k = 0$
$\vee$ **unchanged** }

Let us introduce the following predicates:

RLOCKABLE(ITEMi, USERk) = $(\forall k)(LockTablei_k \neq W)$
READABLE(ITEMi, USERk) = $(LockTablei_k = R \vee LockTablei_k = W)$
WLOCKABLE(ITEMi, USERk) = $(\forall k)(LockTablei_k = 0)$
WRITEABLE(ITEMi, USERk) = $(LockTablei_k = W)$

3.1. Internal safety properties

The USMANk process passes only messages sent by USERk

$\Box$ (PASS(USk, m) P PASS(UMRSk, m)) $m \in UMRSEND$
$\Box$ (PASS(USk, m) P PASS(UMWSk, m)) $m \in UMWSEND$

or to USERk :

$\Box$ (PASS(UMRRk, m) P PASS(URk, m)) $m \in UMRREQ$
$\Box$ (PASS(UMWRk, m) P PASS(URk, m)) $m \in UMWREQ$

The READER and WRITER processes send their messages to the adequate items/users:

$\Box$ (PASS(UMRSk, m) P PASS(RISi, m') $\wedge$ m=(..., ITEMi) $\wedge$ m'=(..., USERk))
$m \in UMRSEMD$, $m' \in RIMSEND$

□ (PASS(UMWSk,m) P PASS(WISi,m') ∧ m=(...,ITEMi) ∧ m'=(..., USERk))
m∈UMWSEMD, m'∈WIMSEND

□ (PASS(RIRi,m) P PASS(UMRRk,m') ∧ m=(...,USERk) ∧ m'=(..., ITEMi))
m∈RIMREQ, m'∈UMRREQ

□ (PASS(WIRi,m) P PASS(UMWRk,m') ∧ m=(...,USERk) ∧ m'=(..., ITEMi))
m∈WIMREQ, m'∈UMWREQ

Before reading/writing an item it must be readable/writeable by the user :

□ [PASS(RISi,(**read**,USERk)) P PASS(IRi,(**read**,USERk))]

□ [PASS(RISi,(**read**,USERk)) ⇒
¬PASS(IRi,(**read**,USERk)) **until** READABLE(ITEMi,USERk)]

□ (PASS(WISi,(*value1*,USERk)) P PASS(IRi,(*value2*,USERk))
∧ *value1=value2*)

□ (PASS(WISi,(*value1*,USERk)) ⇒
¬PASS(IRi,(*value*,USERk)) **until** WRITEABLE(ITEMi,USERk))

A request must be accepted by processes READER and WRITER before answering it correctly:

□ [PASS(RISi,(**rlock**,USERk)) P PASS(RIRi,(**rlock granted**,USERk))]

□ [PASS(RISi,(**rlock**,USERk)) ⇒
¬PASS(RIRi,(**rlock granted**,USERk)) **until** RLOCKABLE(ITEMi,USERk))]

□ [PASS(WISi,(**wlock**,USERk)) P PASS(WIRi,(**wlock granted**,USERk))]

□ [PASS(WISi,(**wlock**,USERk)) ⇒ ¬PASS(WIRi,(**wlock granted**,USERk))
until WLOCKABLE(ITEMi,USERk))]

□ [PASS(RISi,(**rlock**,USERk)) P PASS(RIRi,(**rlock denied**,USERk))]

□ [PASS(RISi,(**rlock**,USERk)) ⇒
¬PASS(RIRi,(**rlock denied**,USERk)) **until** ¬RLOCKABLE(ITEMi,USERk))]

□ [PASS(WISi,(**wlock**,USERk)) P PASS(WIRi,(**wlock denied**,USERk))]

□ [PASS(WISi,(**wlock**,USERk)) ⇒
¬PASS(WIRi,(**wlock denied**,USERk)) **until** ¬WLOCKABLE(ITEMi,USERk))]

□ [PASS(ISi,)*value1*,USERk)) P PASS(RIRi,(**got**,*value2*,USERk))
∧ *value1=value2*]

□ [PASS(RISi,(**read**,USERk)) P PASS(RIRi,(**not read**,USERk))]

□ [PASS(RISi,(**read**,USERk)) ⇒
¬PASS(RIRi,(**not read**,USERk)) **until** ¬READABLE(ITEMi,USERk))]
□ [PASS(RISi,(**runlock**,USERk)) P PASS(RIRi,(**runlocked**,USERk))]
□ [PASS(WISi,(**wunlock**,USERk)) P PASS(WIRi,(**wunlocked**,USERk))]
□ [PASS(ISi,(**written**,USERk)) P PASS(WIRi,(**written**,USERk))]
□ [PASS(WISi,(**write**,*value*,USERk)) P PASS(WIRi,(**not written**,USERk))]
□ [PASS(WISi,(**write**,*value*,USERk)) ⇒
¬PASS(WIRi,(**not written**,USERk)) **until** ¬WRITEABLE(ITEMi,USERk)]

3.2. Internal liveness properties

The processes USMANk, READER and WRITER send further the accepted messages to the addressed processes:

□ [PASS(USk,m) ⇒ <>PASS(UMRSk,m)]   m∈UMRSEND
□ [PASS(USk,m) ⇒ <>PASS(UMWSk,m)]   m∈UMWSEND

□ [PASS(UMRRk,m) ⇒ <>PASS(URk,m)]   m∈UMRREQ
□ [PASS(UMWRk,m) ⇒ <>PASS(URk,m)]   m∈UMWREQ

□ [PASS(UMRSk,m) ⇒ <>PASS(RISi,m')
∧ m=(...,ITEMi) ∧ m'=(...,USERk)]   m∈UMRSEND, m'∈RIMSEND
□ [PASS(UMWSk,m) ⇒ <>PASS(WISi,m')
∧ m=(...,ITEMi) ∧ m'=(...,USERk)]   m∈UMWSEND, m'∈WIMSEND
□ [PASS(RIRi,m) ⇒ <>PASS(UMRRk,m')
∧ m=(...,USERk) ∧ m'=(...,ITEMi)]   m∈RIMREQ, m'∈UMRREQ
□ [PASS(WIRi,m) ⇒ <>PASS(UMWRk,m')
∧ m=(...,USERk) ∧ m'=(...,ITEMi)]   m∈WIMREQ, m'∈UMWREQ

The processes ITMANi lock the items correctly:
□ [PASS(RISi,(**rlock**,USERk)) ∧ RLOCKABLE(ITEMi,USERk) ⇒
<>PASS(RIRi,(**rlock granted**,USERk))]
□ [PASS(WISi,(**wlock**,USERk)) ∧ WLOCKABLE(ITEMi,USERk) ⇒
<>PASS(WIRi,(**wlock granted**,USERk))]

□ [PASS(RISi,(**rlock**,USERk)) ∧ ¬RLOCKABLE(ITEMi,USERk) ⇒

$\langle\rangle$PASS(RIRi,(**rlock denied**,USERk))]

$\Box$ [PASS(WISi,(**wlock**,USERk)) $\wedge$ $\neg$WLOCKABLE(ITEMi,USERk) $\Rightarrow$ $\langle\rangle$PASS(WIRi,(**wlock denied**,USERk))]

The process ITMANi reads/writes the item if the user sending the message has rights to read/write it:

$\Box$ [PASS(RISi,(**read**,USERk)) $\wedge$ READABLE(ITEMi,USERk) $\Rightarrow$ $\langle\rangle$PASS(IRi,(**read**,USERk))]

$\Box$ {PASS(ISi,(*value1*,USERk)) $\Rightarrow$ $\langle\rangle$[PASS(RIRi,(**got**,*value2*,USERk)) $\wedge$ *value1=value2*]}

$\Box$ [PASS(WISi,(**write**,*value1*,USERk)) $\wedge$ WRITEABLE(ITEMi,USERk) $\Rightarrow$ $\langle\rangle$PASS(IRi,(*value2*,USERk)) $\wedge$ *value1=value2*]

$\Box$ [PASS(ISi,(**written**,USERk)) $\Rightarrow$ $\langle\rangle$PASS(WIRi,(**written**,USERk))]

Unlocking can be executed unconditionally:

$\Box$ [PASS(RISi,(**runlock**,USERk)) $\Rightarrow$ $\langle\rangle$PASS(RIRi,(**runlocked**,USERk))]

$\Box$ [PASS(WISi,(**wunlock**,USERk)) $\Rightarrow$ $\langle\rangle$PASS(WIRi,(**wunlocked**,USERk))]

4.CONCLUDING REMARKS

The method presented here seems to be a useful tool for specifying transaction management systems.

In [2] it is given a short external temporal specification of a transaction management system, including serializability, two-phase protocols.

In [3] it is proved by means of tableau method [12],[13] that there exist models satisfying this specification.

LITERATURE

[1] *Rácz,É.*, Specifying a Transaction Manager as Communicating System, Proceeding of the Conference on Computer Science and Software Technology, Visegrad, Hungary (1990)

[2] *Rácz,É.*, *Kozma,L.*, A Temporal Logic Specification of a Transaction Management System, Publications on Pure and

Applied Mathematics, Ser.A. Vol.1 (1990) 369-374.

[3] *Rácz,É.*,Temporal Specification of a Transaction Manager, PhD.Th.1992. (in Hungarian)

[4] *Ed.by Banieqbal,B., Barringer,H., Pnueli,A.*, Temporal Logic in Specification, LNCS 398 (1987)

[5] *Chaochen,Z.*, Specifying Communicating Systems with Temporal Logic, LNCS 398 (1987) 304-323.

[6] *Clarke,E.M., Emerson,E.A., Sistla,A.P.*, Automatic Verification of Finite-state Concurrent systems Using Temporal Logic Specifications ACM TOPLAS 8 (1986) 244-263.

[7] *Dawids,U.G., Lovengreen,H.H.*, Rigorous Development of a Distributed Calendar System, LNCS 259 (1987) 188-205.

[8] *Emerson,E.A.*, Temporal and Modal Logic, in: J.van Leeuven, ed., Handbook of Theoretical Computer Science, Vol.B (North-Holland, Amsterdam, 1990) 995-1072.

[9] *Koymans,R.*, Specifying Message Passing Systems Requires Extending Temporal Logic,LNCS 398 (1987) 213-223.

[10] *Kröger,F.*, Temporal Logic of Programs, Springer-Verlag, Berlin Heidelberg (1987)

[11] *Manna,Z., Pnueli,A.*, Verification of Concurrent Programs: The Temporal Framework, In: The Correctness Problem in Computer Science Academic Press London (1982) 215-273.

[12] *Manna,Z., Wolper,P.*,Synthesis of Communicating Processes from Temporal Logic Specifications, ACM TOPLAS 6 (1984) 68-93.

[13] *Wolper,P.*,The Tableau Method for Temporal Logic:an Overview, Logique et Anal. 28 (1985) 119-136.

Eva Racz
1117 Budapest,Bogdanfy 10/b
Hungary

e-mail: eszk007@ursus.bke.hu
Tel:(36-1) 1669-023
Fax:(36-1) 1811-976

String Matching Animator SALSA

Erkki Sutinen and Jorma Tarhio
Department of Computer Science
P.O. Box 26
SF-00014 University of Helsinki, Finland

Abstract

Animation is a way to illustrate the behavior of complex algorithms and systems. We introduce the SALSA package, an animating and experimenting tool for string algorithms. SALSA was implemented using the XTango animation environment. SALSA contains animations for seven string algorithms including Boyer-Moore, Rabin-Karp, and Aho-Corasick algorithms. We also discuss the usefulness of animation for Computer Science education and research.

1 Introduction

Animation is a useful approach for Computer Science education and research. For example, the idea of an algorithm is easier to grasp by following an animation: the user may observe how the algorithm behaves with various inputs. Because animation can lead to more efficient algorithms, it is therefore to be considered a helpful interactive tool for algorithm research.

String algorithms form an important area of algorithm research. A typical problem is string matching, where approximate or exact occurrences of a pattern is searched in a text. String algorithms are applied in various areas including speech recognition, data compression, text processing, data communications, image processing, and computational biology.

Up till now, there are practically no visualizations made for string algorithms. We collected experiences in a project constructing an animation package, called SALSA, for string algorithms (the name is an acronym for String ALgorithmS Animator). The project was organized as a software engineering assignment for a team of four students of the fifth year [1].

The SALSA package is running in the Xwindows environment on Sun workstations. The implementation of SALSA was based on an animation environment XTango [4, 10, 11]: the algorithms to be animated were supplied with calls to XTango routines for creating and moving visual objects on the screen.

The rest of the paper is organized as follows. In Section 2, we will discuss the principles of the XTango environment and give an overview of different approaches to algorithm visualization. Section 3 lists the goals of the project and outlines the architecture of SALSA. In Section 4, we will discuss our experiments, concentrating on the usefulness of animation in learning and research. The final section will introduce our future plans.

2 Different approaches to visualization in Computer Science

In Computer Science, visualization is used in many ways, like in drawing flowcharts, designing systems, developing user interfaces, simulating phenomena, visual programming, and teaching data structures.

In this section, we will first define the concept of algorithm visualization, according to Myers' taxonomies [5]. After that we will present the algorithm animation framework and the path-transition paradigm of the XTango environment [9, 10, 11]. Starting from this conceptual background, we will itemize the phases to produce an animation. And finally, we will consider future opportunities for perceptional exploration of algorithms.

2.1 Myers' classification of program visualization systems

Myers defines program visualization as illustrating "some aspects of the program after it is written" [5]. Visualization is code, data, or algorithm visualization according to the visualized aspect, and visualization is static or dynamic depending on the produced display.

To exemplify the taxonomy, the traditional flowchart falls into the category of static code visualization, while a graphical debugger showing the line under execution would represent dynamic code visualization. Moreover, a static data visualization system would illustrate a program's tree structure, while its dynamic counterpart would display also the changing values of the nodes.

Another example of code animation is a teaching tool ASSEM with which a user can simulate the CPU of a simple computer [7]. The user specifies the memory location of the first instruction, and ASSEM will step instruction by instruction and show the contents of the main memory and the registers.

For instance by setting the necessary parameters, it is possible to automate both the code and data visualization, without touching the code. Algorithm visualization systems, on the contrary, necessitate the programmer to explicitly add information to the code of the animated algorithm, to create an animation.

In fact, there are animation systems which can visualize the program without any additional information in the code. For instance, PASTIS animates Fortran, C, and C++ programs by making use of the debugger [6]. According to Myers' taxonomy, this kind of systems would not belong to the category of algorithm visualization. However, this is the only natural choice for PASTIS; it should not make any difference whether the animations are produced by additional code in the algorithm or, like in PASTIS, the animation modules are separated from the source code from which they get data through the debugger. The essence of (dynamic) algorithm visualization might, therefore, be defined by producing a (event-driven) visual abstraction of an algorithm.

2.2 The XTango algorithm animation environment

XTango (for XWindows Transition-based ANimation GeneratOr) is a public domain software package, delivered and maintained at the Georgia Institute of Technology.

The XTango algorithm animation environment is based on two principles: (1) the framework to map the interesting events of an algorithm to their visual counterparts, supported by (2) the path-transition paradigm, guiding the design of the animations [9, 10, 11].

The algorithm animation framework The framework consists of three components: (1) the algorithm, (2) the animation, and (3) the mapping component. In the algorithm component, the designer defines the algorithm's interesting events, or in the XTango terminology, algorithm operations. These operations correspond to the important elements of the algorithm's semantics. For instance in the string matching algorithms, the algorithm operations include at least character comparison and moving along a string. The algorithm operations are added to the algorithm as function calls.

The animation part includes the graphical objects for visualizing the algorithm, and the routines for changing their size, color, place etc. For example, to illustrate character comparison, one might specify the characters as rectangles, flash the characters under comparison, and move one on the other to show the difference. The routines for changes in the screen are called animation scenes. Although implemented at higher level by the designer, they call XTango routines to take care of the low level graphics.

The mapping component has two parts. First, XTango uses a kind of symbol table to connect a visual object with a set of parameters from the algorithm. This mechanism is called association. In our string matching algorithm example, the places of the rectangles visualizing the characters T[1], T[2],... might be stored as Assoc(ID,T,1), Assoc(ID,T,2), ...

The second part of the mapping component includes the relation between the algorithm operations and the animation scenes. In our example, the algorithm operation Character Comparison maps to a group of animation scenes: flashing and moving.

The path-transition paradigm The idea behind the path-transition paradigm is to separate the design work of the animations from the implementation phase. The paradigm supplies the designer with four abstract data types with the operations. If the designer specifies the animation scenes with these operations, the implementation should be straightforward, using the XTango routines corresponding to the abstract data type operations.

The four data types of the paradigm include (1) location, (2) image, (3) path, and (4) transition. They relate to each other as follows: An image has a location in the infinite coordinate system of the XTango window. A path is an ordered sequence of coordinate pairs, which defines relative changes in X- and Y-axis. A transition changes an image according to a path.

Let us illustrate the paradigm with an example. To design the animation scene for moving a rectangle on another, the images concerned are these rectangles. The path along which Xtango will perform the transition is defined by the locations of the rectangles.

The path-transition paradigm, in addition to the algorithm animation framework of XTango, gives the guidelines for the different phases in the design and implementation of an animation. The designer starts with identifying the algorithm operations of interest. Then, he/she will decide the animation scenes necessary to visualize the operations; the scenes must be designed using the path-transition paradigm. The crucial phase is to define the relation between the algorithm operations and the animation scenes. Last, the implementor turns animation scenes into C functions, calling the appropriate XTango routines.

2.3 Future trends in algorithm experimentation

Among the possibilities to study an algorithm using technology, visualizing is only a beginning. However, even visualization can be of greater advantage. In the design of animation, much more attention must be paid on psychological factors. For instance, the designer should use colors in a way which helps following the animation. Brown and Hershberger list the following use of colors [2]: encoding the state of data structures, highlighting activity, tying views together (in the case of multiple windows for different aspects of an algorithm), emphasizing patterns, and making the history of an algorithm visible.

Besides the visualization, one could also make use of auralization (interpreting interesting events as sound effects) [2, 8]. As colors, they can present fundamental information on an algorithm when used for reinforcing visuals, conveying patterns (e.g. by using multiple instruments), replacing visuals (to reduce the visual information), and signaling exceptional conditions.

This is just the beginning. Maybe computer-assisted algorithm exploration only starts with seeing and hearing, leading us to smell, taste, and touch the algorithms! Virtual reality is coming inside the researcher's chamber.

3 An overview of SALSA

In this section, we will explain our objectives in the SALSA project and go through the main components of the SALSA package [1]. We will first outline the architecture of the package with comments on the choice of the algorithms.

As stated in the introduction, the SALSA package is running in the Xwindows environment. When started, SALSA opens its main window, which controls other components. The respective program module calls for the animated algorithms. These algorithms, implemented in C, include calls to the animation routines, programmed by our team; however, to control the graphics, the animation routines use the XTango functions. One of these functions opens the XTango animation window; as a result, the animations run in this window, and the user can control the animation by pushing the icons provided by XTango.

In the beginning of our project, we had no prior expertise in principles and techniques of algorithm visualization. However, we aimed at a working animation tool suitable for introductory purposes in teaching string algorithms. For these reasons, we decided to start with the very basics. Therefore, our choice comprised seven algorithms for one- and two-dimensional pattern matching, and calculating the edit distance [3].

3.1 The aims of SALSA

The overall goal of the SALSA project was to develop a computer-assisted instruction package. We intended this package to serve as an introduction to string algorithms either in a classroom or as a self-education material. To make our aim clear, we divided it into smaller subgoals:

First of all, the package should help one to understand string algorithms of different types. For this purpose, we decided to use animation. When animating an algorithm, the student should be able to follow its behavior with various inputs and so he/she can gain insight into the essence of the algorithm.

To make SALSA as pedagogical as possible, we followed the principles of computer-assisted instruction (CAI) in the implementation [13]. This required special attention to the user interface design.

Besides the educational perspective, SALSA should also be useful for research. This means that the package should support implementation of other string algorithms. It has usually taken too much time for the researcher to design and implement an animation of an algorithm; if this phase were considerably reduced, however, the animation itself would give new ideas in analyzing the algorithm and developing it further.

In addition to animations, SALSA should also include a kind of workbench for testing the algorithms' efficiency. This is important in learning as well as in research. While animation helps in understanding the algorithm's idea, only the hard facts about the CPU time usage tell the conditions in which to apply the algorithm.

In our project, we also wanted to evaluate the usefulness of the XTango environment, although this was not particularly the aim of SALSA. We were interested especially in how efficient the path-transition paradigm would be in the animating process.

3.2 The components of SALSA

The SALSA package consists of four main components: the graphical user interface, the animations for selected basic string matching and edit distance algorithms, the CPU time measurement, and the test data generator [1].

The graphical user interface To let the user of SALSA to concentrate on the algorithms, not the package itself, it was important to make SALSA as easy to use as possible. Therefore, we decided to make the user interface graphical, implementing it with Devguide, a development tool operating in the OpenWindows environment. In the design, we followed the OpenLook standard.

Beyond the technical implementation of the user interface, it was essential to identify the suitable learning strategies supported by SALSA. It seemed quite natural to make use of processive learning. SALSA could easily take the student all the way through the different phases of the learning process: motivating, orientating, deepening, exemplifying, practicing, evaluating.

However, we can regard an algorithm also as a system which the user can simulate by perceiving the animation with varying inputs. In addition to processive learning, SALSA would support, thus, learning by simulation.

These principles in mind, we designed the main window (Fig. 1), where the user first selects the algorithm and its input, possibly setting some parameters, and then pushes the button indicating the desired function. With the function completed, the control returns to the main window, and the user may define a new procedure.

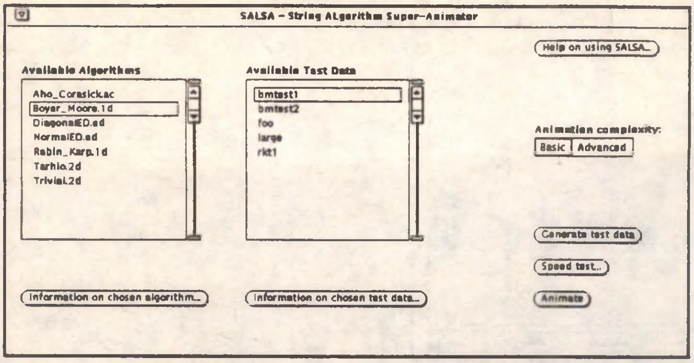


Figure 1: The SALSA main window. The user first chooses an algorithm and test data. Pushing the Animate button starts the respective animation, while the Speed test button outputs an efficiency report.

Note that it is also possible to study the algorithms only at a theoretical level, by pushing the Information button.

The animations The user can start the desired animation from the main window, pushing the Animate button. For instance, after choosing Boyer-Moore from the algorithm list and bmtest1 from the test data list, the XTango window will open and show the initial scene of the respective animation (Fig. 2).

Let us exemplify the animation procedure by looking closer at the animation of the Boyer-Moore algorithm (Fig. 3). We decided to visualize the characters by rectangles of equal width, with the height indicating the character's position in the alphabet. Second, the pattern would travel above the text. Moreover, we illustrated the comparison of two characters by moving them on one another. We displayed a match by coloring the respective rectangles black, while a zigzag arrow indicated a mismatch. A matched suffix was visualized by a line below the equal substring in the pattern.

With this design specified, we implemented the animation routines. These routines defined the locations, images, paths, and transitions, using the XTango functions.

At present, SALSA consists of the animations of basic algorithms for one-dimensional pattern matching (Boyer-Moore, Rabin-Karp, and Aho-Corasick). In addition, we implemented also the animations of calculating the edit distance with normal and diagonal methods [3]. To get insight into how a researcher may benefit from animation, we also prepared a visualization of a two-dimensional algorithm under development.

The CPU time measurement The user who is interested in the practical efficiency of an algorithm may forget the animations and run the speed test. SALSA will store the results in a log file specified by the user.

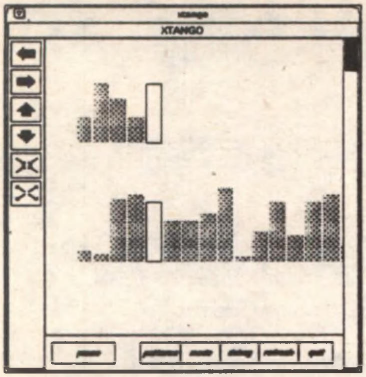


Figure 2: The initial scene of the Boyer-Moore animation. The implementor defines the visual objects inside the window by using XTango routines, while XTango provides the window with the basic operations: With the left-hand buttons, the user can pan and zoom the animation window. Moving the right-hand scrollbar downwards slows down the animation. By pushing the pause button, the user can pause the animation.

Generating and storing the test data To help the user to experiment the algorithms with various inputs, we included a test data generator SWING (for String WeavING). The user may create different inputs consisting of the desired text and patterns by connecting together two files: one containing the text, the other the patterns.

Because SALSA creates a file for each new test data generated, it is easy for the user to repeat the same run of an algorithm or to test several algorithms with the same data.

4 Discussion

Our experiences with the SALSA project produced four conclusions: First, in teaching string algorithms, animation serves as an activating teaching method which inspires students to experiment. Though rather short in code, the essence of a string algorithm is often quite hard to uncover.

A group of students of our department tested the SALSA package. The results were promising: by using the animation, it was easier to learn the idea behind the algorithm. When we made a video on SALSA, even the cameramen were interested how the algorithms worked, with no prior knowledge in Computer Science!

Second, designing an animation is a learning process which leads to a profound understanding of the algorithm. When transforming the detailed algorithm to the higher level of abstraction, the designer little by little gets closer to the essence of the algorithm. Actually, the designing process is interaction between learning, teaching, and research.

The students of our team had no prior knowledge of string algorithms. However, all of them got interested in the problem area and studied themselves the area

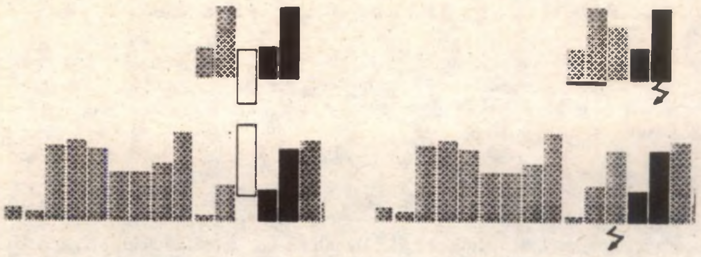


Figure 3: Two phases in the Boyer-Moore animation. In the left-hand figure, the animation illustrates the comparison of two characters, with the respective rectangles approaching each other. In the right-hand phase, the algorithm has found a matched suffix in the pattern, visualized by an underline; the zigzag arrow indicates the character responsible for the unmatch.

beyond the algorithms to be animated. What happened with the Aho-Corasick algorithm, describes well the learning process. The designer of this animation, having seen how the Boyer-Moore algorithm works, noticed that it would be efficient to combine these two methods. It was a pity that this approach is already known as the Commentz-Walter algorithm.

Third, animation is useful also for research. Experimenting with animations can indicate weaknesses in specific cases in the behavior of an algorithm under study and thus help in developing the algorithm further. Animation may also help in analyzing the complexity of the algorithm.

We got a nice example of this when animating a new algorithm for two-dimensional pattern matching. The co-operation between the animator and the researcher led to a more efficient algorithm.

In fact, it is possible to infer the aims of SALSA partly from the functions of the university: how to create a natural interaction between learning, teaching, and research. At the same time, there has been discussion about the communicative role of the university. The projects like SALSA teach the participating students to pay attention on how to present "the professional issues" for laymen.

Fourth, the path-transition paradigm of XTango proved to be an approach practicable enough for use in animating at least string algorithms. An overall evaluation of the tool was carried out during the project.

The team regarded especially the conceptual design (the path-transition paradigm) behind the XTango environment as easy to learn. For a beginner, it took about four days to implement an animation of the Rabin-Karp algorithm. The animation scenes took about 800 lines of code.

The main problem with XTango was its poor performance in the OpenWindows environment. According to John T. Stasko, the designer of XTango, the bottleneck lies in the performance of the X graphics implementation of the workstation [12].

As a consequence, the number of the images in the animation must not exceed the order of tens.

Despite other minor shortcomings, like the lack of multiple windows (to compare algorithms with one another, or to display different aspects of an algorithm), we are looking forward to the future. A new, C++ based animation environment, called Polka, is already available on the ftp.

5 Future work

Our experiences in construction of SALSA and in using animations encourage us to develop new animations. Next year we will produce an enhanced version of SALSA by incorporating animations of a new set of string algorithms. The possibility of using parallel animations and sound output will be considered.

We have also plans to make animations for other areas. There are many subject areas with hardly any animation packages, since most implementations visualize sorting, graph algorithms,.computational geometry, or simulation of computer systems. One neglected area is compiling of programming languages. Many compiling techniques including parsing, attribute evaluation, and code generation are based on a parse tree. Such schemes are conceptionally easy to animate using a graphical representation of a tree.

Our promising experiences of using XTango for construction of SALSA show that production of animation packages in not any more tedious prototyping it used to be. We believe that the use of animation will rapidly increase in Computer Science education and research in the near future.

Acknowledgements

We wish to thank Prof. Esko Ukkonen for his continuous support and Prof. Kari-Jouko Räihä for introducing us the animation technology. We are grateful to Juhana Britschgi, Timo Joutsenvirta, Kai Järvenranta, and Antti-Pekka Tuovinen for implementation and fruitful co-operation.

References

[1] Britschgi, J., Joutsenvirta, T., Järvenranta, K., Tuovinen, A.-P., The SALSA animator: An animating and experimenting tool for string algorithms (in Finnish). Report C-1993-14, Department of Computer Science, University of Helsinki, 1993.

[2] Brown, M., Hershberger, J., Color and sound in algorithm animation. *Computer* 25, 12 (1992), 52–63.

[3] Gonnet, G., Baeza-Yates, R., Handbook of Algorithms and Data Structures in Pascal and C. 2nd edition. Addison-Wesley, Wokingham 1991.

[4] Hayes, D., The XTANGO environment and differences from TANGO. Electronic document included in the XTango package, November 3, 1990.

[5] Myers, B., Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing* 1990, 97–123.

[6] Müller, H., et al., The program animation system PASTIS. *The Journal of Visualization and Computer Animation* 2, 1 (1991), 26–33.

[7] Nasri, A., Computer graphics in simulating the functioning of a simple computer. *Computer Science Education* 2, 2 (1991), 161–170.

[8] Negroponte, N., Beyond the desktop metaphor. In: Meyer, A., et al., editors, *Research Directions in Computer Science: An MIT Perspective*, The MIT Press. Cambridge, Massachusetts, 1991, 183–190.

[9] Stasko, J., The path-transition paradigm: A practical methodology for adding animation to program interfaces. Electronic document included in the XTango package, College of Computing, Georgia Insitute of Technology, June 10, 1991.

[10] Stasko, J., TANGO: A Framework and System for Algorithm Animation. Ph.D. Dissertation, Technical Report No. CS-89-30, Department of Computer Science, Brown University, 1989.

[11] Stasko, J., Tango: A framework and system for algorithm animation. *Computer* 23, 9 (1990), 27–39.

[12] Stasko, J., Personal communication, 1993.

[13] Steinberg, E., Computer-Assisted Instruction: A Synthesis of Theory, Practice, and Technology. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1991.

Specifying User Interfaces as Joint Action Systems

Kari Systä
Tampere University of Technology
BOX 553, SF-33101 Tampere, Finland
ks@cs.tut.fi

Abstract

Formal methods can be used in the specification of behavioral aspects of user interfaces. Most previously used formal methods cannot, however, describe properties that are consequences of concurrency in modern user interfaces. If parallelism can be described, the level of abstraction is often too low, and includes unnecessary implementation details.

In this paper we introduce a new approach that allows concurrency to be described at a high level of abstraction without implementation details. More specifically we use an executable specification language DisCo, for which we have developed support tools including an execution environment with graphical animations. In addition to validation by execution we can also use formal proofs for critical properties of the specifications. Another important aspect of DisCo is its support for stepwise refinement of specifications. This allows the addition of new properties in such a way that safety properties of previous stages are preserved.

The approach and the language are exemplified by a stepwise specification of an electronic mail system.

Keywords: formal specifications, user interfaces.

1. Introduction

User interface contains two parts: representation and behavior. The representation defines the symbols that are used in the display and input devices. The behavior determines the relation between user input and system responses. Often the representation alone is considered to be the user interface, but the understandability of the behavior is also an important criterion for a good user interface. In [5] user interface is defined to be *all user and machine behavior that is observable by an external observer.* We share this view.

A complete specification of the behavior specifies both the user interface and application semantics of the system. When the emphasis is on the user interface, we concentrate on the observable behavior, and we may even allow more nondeterminism than is acceptable in semantically correct system behaviors. In particular, we do not concentrate on internal issues or on interfaces to other software components.

According to [10, 15] there are three types of concurrency in user interfaces: 1) concurrent output, 2) concurrent input, and 3) concurrent dialogue. An example of concurrent output is simultaneous updating of several windows or views. Concurrent input is possible, for instance, when a mouse and a keyboard are used together. Concurrent dialogues are used when the user may supply input to several windows simultaneously.

Even if we do not have explicit concurrency in the user interface, there is always a need for concurrency in the model, because both the user and the computer system are active agents. For example, the user can press the interrupt key at any time during the execution of the program. This means that program execution and keyboard monitoring are parallel operations.

1.1. Motivation for formal methods

In the design phase, a formal description of system behavior can expose inconsistencies and errors in the design. A formal description also gives a rigorous basis for its implementation.

When an existing user interface is described, formal specification is also helpful as a reverse engineering or documentation aid. The specification of an existing interface can be used as a supplement to the user manual and design documentation. Especially, a formal specification can complement the user manual by giving exact answers to questions that are typically missing in informal descriptions. If the specification is given in a formal notation, it is possible to recognize inconsistencies and design flaws in the current version of the user interface, and the quality of the next version can be improved.

1.2. Methods used in formal description of user interfaces

Based on literature, Booth [3] gives two different uses for the term *formal methods* in human-computer interface engineering. One type of methods attempt to cognitively model the user. An example of such methods is *Task-Action Grammar* (TAG) [16]. The purpose of these methods is to show how complex tasks are in cognitive terms. In the other type of formal methods the behavior of a computer system is described in a formal notation. In this case general purpose specification methods are used instead of special user interface methods. When used for interface specification, their purpose is to expose logical inconsistencies within a system and its user interface. Our aim also falls in this category. Compared to most other specification methods, our notation of joint actions looks like a programming language instead of mathematical formulas, but the purpose is the same: to express logical properties and to expose logical inconsistencies.

Z [9] is one method that is used for specification of user interfaces [4]. Z is based on sets, which describe the internal state and variables of the program. In a Z specification all operations are described as operations on sets. When Z is used for specification of user interfaces, the representation components like window locations are also described by using these sets.

One way of specifying user interfaces is to use *context free grammars* to specify languages that are used in man-machine communication. The behavior of the system can be added to the productions as actions [7].

Different kinds of state automata or *transition networks* (*TN*) are also used for specification of reactive systems. For imposing structure on transition networks the concept of subdiagrams is introduced. If a subdiagram can call itself, the notation is called *recursive transition networks* (*RTN*). A common way to extend the expressive power of transition networks is to add actions to states or transitions. Such notations are called *augmented transition networks* (*ATN*). They can be improved further by adding conditional branches to transitions [20], where the conditions depend on the return values of the actions.

The use of *algebra* is one possible formal specification method, and in [5] algebraic specification is one of the example notations. In the algebraic approach a specification consists of classes of objects and a set of functions that operate on these classes. The semantics is defined by a set of algebraic axioms.

In order to support concurrency, different kinds of *event* approaches are used. In [5] the notation is called *event algebra*. In [10] an *event-response language (ERL)* and *local event broadcast method* (*LEBM*) are introduced. LEBM is basically a structuring method that supports communication and synchronization between modules. A third event-based notation is used in [7]. A common denominator in these approaches is that event handlers are used to react to events sent by the external world or by other event managers. Many researchers prefer event-based models over the other methods mentioned above, mainly because event-based approaches support concurrency [5, 7, 10, 14].

A novel approach described in this paper is to use joint actions. With them we can describe the synchronization without implementation details like communication mechanisms. With event-based methods our approach shares the ability of describing concurrency. The advantages of our method include its abstraction in the description of actions and support for stepwise refinement.

The rest of the paper is organized as follows. Section 2 gives an introduction to joint actions and to the DisCo language. The purpose is to provide the details and principles that are needed in understanding the rest of this paper. In Section 3 the usage of DisCo in the specification of user interfaces is discussed in the light of an example. In Section 4 conclusions and some directions for future work are given.

2. Introduction to specification in DisCo

Joint actions [1, 2] can be used to specify reactive systems. Reactive systems are ones that are in continuous interaction with their environment. This distinguishes them from traditional input/output computations. Reactive systems typically contain parallelism and they are often embedded systems. This concurrency may be included in the system, or at least the system and its environment may both execute simultaneously. The aim of the DisCo approach is to specify reactive systems with potential or real concurrency at a high level of abstraction.

A joint action specification consists of multiparty actions and objects that participate in them. A difference between joint actions and most other operational specification methods is that the notion of processes is not inherent in joint actions. The objects in joint action systems can be implemented either as processes or as passive data structures. An important aspect in DisCo is that the effects of actions are expressed as special syntactic entities instead of distributing to the descriptions of the effects to the participating objects.

In this paper only a brief overview is given on the DisCo specification language. A more detailed description can be found in [11, 12]. DisCo supports object-oriented modelling; DisCo objects always belong to a *class* that defines the data and state structures of objects within that class. A comparison of DisCo and the object-oriented paradigm can be found in [12]. The finite-state structure of a DisCo object is similar to the hierarchical structure used in *statecharts* [8]. An object can be understood to consist of two interrelated parts: a finite state part that is a state automaton, and a data part that contains variables and constants. Figure 1 gives an example of a class definition and the corresponding statechart. In DisCo state transitions correspond to actions, and they have been omitted from Figure 1.

```
class Proc (Next:Proc) is                  -- Next is a parameter, which is a reference to another Proc-object
    state *Prepare, Compute;               -- two states, default state is marked with "*"
    state *Important,Unimportant;          -- two independent states
    extend Important by
        state *Quite_I, Very_I;            -- substates of Important
    end Important;
    Data : integer;                        -- a variable
end Proc;
```

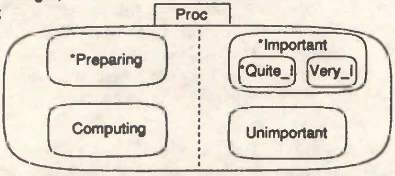


Figure 1. An example class and the corresponding statechart. The state transitions are not shown.

The other basic components in DisCo are *actions*, in which the objects may participate. An action definition has a *name*, zero or more *parameters*, one or more *participants*, a *body*, and an enabling *guard*. Each participant is specified by its class, and it is given a formal name called *role*. Whenever an action is executed with some objects as participants, the action body may change the participating objects. An action cannot modify objects that are not participants of the action. The guard is a boolean expression which has to be true for the action to be executed. The guard can therefore be used to restrict the possible participant combinations. The following is an example of an action:

```
action Exchange (X: integer) by L,R:Proc is
when X > 0 and L.Data > R.Data+X do-- guard
    R.Data := L.Data || L.Data := R.Data; -- body of the action
end Exchange;
```

This action has two participants named L and R, which are both instances of class Proc. The identifier X denotes a parameter, whose value is determined whenever the execution of the action begins. The guard of this action restricts the participating objects and the possible values of the parameter X. The action is enabled only for those objects of class Proc that satisfy the condition given in the guard. If there is freedom in selecting the possible participants or parameter values, they are determined by a nondeterministic choice. An execution of a DisCo specification consists of successive actions; at any moment the next action can be any of the enabled actions.

In addition, *assertions* can be included in specifications. An example of an assertion is:

```
    assert ∀ P:Proc:: Proc.Data > 0;
```

which states that the Data variables of all Proc-objects must be positive. Assertions can be used to specify invariants for the whole execution, and to restrict the initial state of the system. In the language the assertions are for documentation of the expected properties, and the tool checks their validity during execution.

Reasoning about DisCo specifications is based on an execution model in which actions are executed sequentially - not in parallel. However, the next action to be executed is selected by a nondeterministic choice from the enabled actions. This means that an interleaving model of parallelism is used. The relation between nondeterministic serial execution and parallel execution has been addressed in [2].

DisCo has two mechanisms for stepwise derivation and reuse of specifications: *inheritance* that is similar to the corresponding concept in object-oriented programming languages, and an extension mechanism that is suited for *superposition*.

To support the specification process, a tool has been implemented [17]. The user interface of the tool can be used for browsing, execution, and visual animation. The graphical displays are always generated from a textual specification by the tool. The graphical animation of the tool makes its possible to extend the group that works with the specification. In addition to execution (simulation), formal proofs can be carried out to reason about critical properties in the specification. So far the tool assists in verification of formal properties only by an automatic execution-time checking of assertions.

Executability gives the possibility to validate specifications by testing. It is clear that critical properties cannot be verified by testing, because the execution may never reach all possible states of the system. On the other hand, formal reasoning is expensive and time consuming, and therefore applicable only to the most critical properties. Another problem with formal reasoning is the recognition of the properties to be proved. Our goal is to combine testing and experimentation with formal reasoning.

3. Application of DisCo in the specification of user interfaces

The DisCo language and the joint action methodology have been designed for specification of reactive systems. The originally intended application area is embedded systems like lift controls or telephone exchanges. The same model can be used also for specifying the interaction between a computer system and its user. The user interfaces of modern workstation environments are, in fact, reactive systems because the user can continuously control the programs. Programs are no longer batch programs which first read the input data, then perform the computation, and finally output the results. Also, modern workstations allow several interactions with one or several applications to exist simultaneously.

As discussed above, event-based approaches can be used in the specification of concurrent user interfaces. One problem with event-based approaches is their low level of abstraction in the sense that synchronous operations are distributed to several event handlers. Event handlers also provide an implementation-oriented structure for the control, which we consider harmful at the specification level.

In DisCo we have an action-oriented view instead of a process-oriented view. This means that the abstraction level is raised to independence of process structure and communication mechanisms. We only describe what is done and which participants are needed, not who is responsible for initiating actions and which communications mechanisms are used in them.

The rest of this section gives an example of user interface specification in DisCo. The purpose of this section is to give an example of how DisCo can be used inn specification of user interfaces. In order to show structuring capabilities, the specification is written in several refinement steps.

3.1. An example specification

In this section we describe a specification of an electronic mail system. In this example we show how the behavior of the system can be described in DisCo specification language. The theorems describing the critical properties of the design are expressed as assertions. A proof of one of these theorems is also shown.

This specification has been tested by using our execution and animation tool. Because the theorems have been included as assertions, the tool has ensured that the theorems hold at least in all states reached by the execution. Because we have not included any theorem prover in our tool, formal proofs for all theorems have not been carried out.

The specification proceeds in four superposition steps:

(1) simple sending and reading;

(2) spooling and mail boxes;

(3) user interface with windows;

3.1.1. Sending and reading

First we introduce a class for mail messages. The message can be in three possible states: Idle (nonexisting) Incomplete (under preparation) and Ready (ready to be read). The variable Recipient contains the user to whom the message is sent. The integer variable Body represents the content of the message. The value of variable Body is nonnegative, and value 0 represents an empty body. We assume that the number of available messages is unlimited.

```
class Message is
    state *Idle, Incomplete, Ready;
    Recipient: User;        initially Recipient = null;
    Body : Integer ;        initially Body = 0;
end;
```

The users have two sets of messages: Sending (messages under preparation) and Reading. The number of messages being concurrently prepared and read by an individual user is unlimited. This means that we can have concurrent interactions (dialogs) with several messages.

```
class User is
    Sending: set Message; initially Sending = ∅;
    Reading: set Message; initially Reading = ∅;
end;
```

The action that starts preparing a message is the following:

```
action Start_Send by M:Message; U:User is
when M.Idle do
    → M.Incomplete;
    assert M.Body = 0  ∧  M.Recipient = null ∧ (M ∉ U.Sending);
    U.Sending := U.Sending ∪ {M};
end;
```

This action inserts an idle message to the set of messages under preparation. The assertions in the action bodies describe our assumptions, and they can be proven in later phases of the specification.

The next action adds an instance of class recipient to a message:

```
action Add_Recipient (R:User) by M:Message; U:User is
when M.Incomplete ∧ M ∈ U.Sending ∧ M.Recipient = null do
    M.Recipient := R;
end;
```

The guard of this action requires that the recipient is not already given. This means that, once given, the recipient cannot be changed, and a formal analysis could expose that this is an irreversible operation. This is an example of an user interface property that can be examined by using formal methods.

The next action adds a body to a message:

```
action Add_Body (I:Integer) by M:Message; U:User is
when M.Incomplete ∧ M ∈ U.Sending ∧ 0 < I ∧ M.Body = 0 do
    M.Body := I;
end;
```

A completed message is sent by action Send:

```
action Send by M:Message; U:User is
when M ∈ U.Sending ∧ M.Incomplete ∧ M.Body > 0 ∧ M.Recipient ≠ null do
    → M.Ready;
    U.Sending := U.Sending – {M};
end;
```

The guard requires that both the body and recipient of the message are given prior to sending.

We have also an action for canceling the sending:

```
action Cancel by M:Message; U:User is
when M ∈ U.Sending ∧ M.Incomplete do
      U.Sending := U.Sending - {M};
      → M.Idle;
      M.Body := 0;
      M.Recipient := null;
end;
```

Action Read inserts a ready message to the list of messages under reading:

```
action Read by U:User; M:Message is
when M.Ready ∧ U = M.Recipient ∧ M ∉ U.Reading do
      U.Reading := U.Reading ∪ {M};
end;
```

The guard ensures that messages are read only by the recipient and that the message is not already being read.

If the user reads the message but saves the message for further reading, action Keep is executed:

```
action Keep by U:User; M:Message is
when M ∈ U.Reading do
      U.Reading := U.Reading - {M};
end;
```

If the user discards the message after reading, action Dispose is executed:

```
action Dispose by U:User; M:Message is
when M ∈ U.Reading do
      U.Reading := U.Reading - {M};
      → M.Idle;
      M.Body := 0;
      M.Recipient := null;
end;
```

This completes the simplest specification of the electronic mail.

3.1.2. Theorems and proofs for simple sending and reading

Theorem 1. *Only ready messages are in User.Reading:*

```
assert Only_Ready_In_Reading is
      ∀ U1:User :: (∀ M1:Message | M1 ∈ U1.Reading :: M1.Ready);
```

In assertions and proofs we have used the following naming conventions: indexed names (M1, U1, ...) are used for quantified variables. The unindexed names (M, U, ...) are used for action participants.

This can be proven as an invariant by showing that the statement is true in the initial state, and no action breaks the invariant. This is done in the following reasoning so that actions are treated one by one:

- The assertion is initially true because U.Reading is initially empty for all U.
- Actions Add_Recipient and Add_Body are actions that do not change the state of any message or change the content of any U.Reading. Thus they can not break the assertion.
- Action Start_Send changes the state of a message to incomplete but only if the original state is idle (i.e. not ready). Thus, if the assertion holds before the execution of Start_Send, its execution cannot break it.
- Action Send changes the state of a message but only if the original state is incomplete. Thus, if the assertion holds before, the execution of Start_Send cannot break it.
- Action Cancel changes the state of a message but only if the original state is incomplete. Thus, if the assertion holds before, the execution of Start_Send cannot break it.
- Action Read inserts a message to U.Reading, but the guard of Read requires that the state must be ready.
- Action Keep removes a message from U.Reading, which cannot violate the assertion.

- Action Dispose changes the state of a message from ready to idle, but it also removes the message from U.Reading. So, provided that the message is never in two U.Reading, theorem 1 holds.
  A message can never be in two U.Reading because they are added to U.Reading only in action Read, and the guard of Read requires that U.Recipient is equal to U, and U.Recipient is changed only when U is not in any U.Reading (i.e. in state incomplete).

The theorem 1 can also be proven using temporal logic of actions [13]. In the following we use a notations where an unprimed predicate *P* refers to variable values before an action and primed *P'* refers to variable values after execution.

Now, in order to proof invariant *P* we must prove that *P* is true in the initial state and for all actions:

$Action \wedge P \Rightarrow P'$

For the initial state we know that

$\forall\ U1{:}User{::}U1.Reading = \emptyset$

which directly implies theorems 1 and 1b.

For action Start_Send we can write (U and M refer to participating objects and a pseudo variable Values refers to all variables – including state):

$$\begin{aligned}
&Start\_Send \Leftrightarrow\\
&\quad \forall\ U1{:}User \mid U1 \neq U :: U1.Values' = U1.Values \wedge\\
&\quad \forall\ M1{:}Message \mid M1 \neq M :: M1.Values' = M1.Values \wedge\\
&\quad M.State = Idle \wedge M.State' = Incomplete \wedge U.Sending' = U.Sending \cup \{M\} \wedge\\
&\quad M.Recipient' = M.Recipient \wedge M.Body' = M.Body \wedge\\
&\quad U.Reading' = U.Reading
\end{aligned}$$

where the first two conjuncts guarantee that all nonparticipating objects are unchanged.

Now we can prove that Start_Send does not break theorem 1. First we express action Start_Send and theorem 1 in temporal logic of actions:

$$\begin{aligned}
&Start\_Send \wedge Only\_Ready\_In\_Reading\\
\Leftrightarrow\ &\forall\ U1{:}User \mid U1 \neq U :: U1.Values' = U1.Values \wedge && (1)\\
&\forall\ M1{:}Message \mid M1 \neq M :: M1.Values' = M1.Values \wedge && (2)\\
&M.State = Idle \wedge M.State' = Incomplete \wedge U.Sending' = U.Sending \cup \{M\} \wedge && (3)\\
&M.Recipient' = M.Recipient \wedge M.Body' = M.Body \wedge && (4)\\
&U.Reading' = U.Reading \wedge && (5)\\
&\forall\ U1{:}User :: (\forall\ M1{:}Message \mid M1 \in U1.Reading :: M1.State = Ready) \wedge && (6)
\end{aligned}$$

Based on these numbered conjuncts we can make the following reasoning:

$$\begin{aligned}
&(2) \Rightarrow\\
&\quad \forall\ M1 : Message \mid M1 \neq M :: M1.State' = M1.State && (7)\\
&(5) \wedge (1) \Rightarrow\\
&\quad \forall\ U1{:}User :: U1.Reading' = U1.Reading && (8)\\
&(3) \wedge (6) \Rightarrow (because\ M.State = Idle)\\
&\quad \forall\ U1{:}User :: M \notin U1.Reading && (9)\\
&(7) \wedge (9) \Rightarrow\\
&\quad \forall\ U1{:}User :: (\forall\ M1{:}Message \mid M1 \in U1.Reading :: M1.State' = M1.State) && (10)\\
&(6) \wedge (10) \Rightarrow\\
&\quad \forall\ U1{:}User :: (\forall\ M1{:}Message \mid M1 \in U1.Reading :: M1.State' = Ready) && (11)\\
&(11) \wedge (8) \Rightarrow\\
&\quad \forall\ U1{:}User :: (\forall\ M1{:}Message \mid M1 \in U1.Reading :: M1.State' = Ready) && (12)\\
&\quad \Leftrightarrow\ Only\_Ready\_In\_Reading'
\end{aligned}$$

This means that we have proved:

Start_Send ∧ Only_Ready_In_Reading ⇒ Only_Ready_In_Reading'

The above proof concerned only one action for one theorem, and a proof for all actions would take several pages. It is clear that this kind of formal proofs are difficult and time consuming to carry out manually. Without computer assistance this can be done to the most critical properties only.

We have also the following theorems, but the proofs have been omitted from this paper.

Theorem 2. *Messages to be sent are incomplete and all incomplete messages are in some U.Sending:*

```
assert Sending_Means_Incomplete is
    { M1:Message | M1.Incomplete } = {∃ M1:Message | (∃ U1:User :: M1 ∈ U1.Sending)};
```

Theorem 3. *An idle message is not read or prepared by anybody:*

```
assert Idle_Is_Idle is
    ∀ M1:Message | M1.Idle :: ¬ (∃ U1:User :: M1 ∈ (U1.Reading ∪ U1.Sending));
```

Theorem 4. *The same message is not in more that one U.Reading or U.Sending:*

```
assert Message_Not_In_Two_Sets is
    ∀M1:Message :: (∀ U1:User | M1 ∈ U.Reading ∪ U1.Sending
                                  :: ¬ (∃ U2:User | U2 ≠ U1 :: M1 ∈ U2.Reading ∪ U2.Sending) ∧
                                   ¬ (M1 ∈ U1.Reading ∩ U1.Sending));
```

3.1.3. Spooling mail boxes

A mail system has usually a mailbox for each user. The incoming mail is collected to that mailbox. In our specification, action Send cannot add messages directly to these mailboxes because the recipient user[1] may me committed to another action, or the receiving mailbox is not available. We do not want to delay the sender unnecessarily. This is one of the reasons why spooling is used also in real electronic mail systems. Thus, we specify a spooling queue for messages to be added to the mailbox.

We add these properties by using *superposition*, which is the main method for specification refinement in DisCo. In superposition we can add new properties so that all *safety properties* are maintained. This means that the new specification cannot do anything that was not possible in the old system. We can, however, add new variables and new operations for these variables because these operations are not visible in the old specification.

We add a new class for the spooling queue, and we extend the existing class User with a mailbox:

```
class Spool is
    Queue: set Message;
end;

extend User by
    Mailbox: set Message; initially Mailbox = ∅;
end;
```

It is assumed that we have only one instance of spool, which can be expressed as an initial condition:

```
initially Only_One_Spool is (+/ S:Spool :: 1) = 1;
```

Action Send is refined to have a new participant Q, and the body of the action is extended by a statement to add the message to the queue:

```
refined Send by ... Q:Spool is
when ... do
    ...
    assert ¬(M ∈ Q.Queue);
    Q.Queue := Q.Queue ∪ { M };
end;
```

This refinement as well as all other refinements and new actions change only newly added variables of the specification.

A totally new action is needed to move a message from the spool to a mailbox. Notice that the sending user is not involved here.

```
action From_Spool by S:Spool; U:User; M:Message is
when U = M.Recipient ∧ M ∈ S.Queue do
    assert ¬(M ∈ U.Mailbox);
```

1. Actually we have not talked about what we mean by a user. At this level of specification a user can be understood as a set of resources (e.g. home directory, mail box, workstation) reserved for a user.

```
          U.Mailbox := U.Mailbox ∪ { M };
          S.Queue := S.Queue − {M};
     end;
```

For action Read we add a new conjunct which requires that the message must be in a mailbox, i.e. the message cannot be taken directly from the spool.

```
     refined Read is
     when ... M ∈ U.Mailbox do
          ...
     end;
```

The body of action Dispose is extended by a statement to remove the message from a mailbox:

```
     refined Dispose is
     when ... do
          ...
          assert M ∈ U.Mailbox;
          U.Mailbox := U.Mailbox − {M};
     end;
```

3.1.4. Theorems for spooling

Theorem 5. Only ready messages in spool ∪ mailbox and messages in spool ∪ mailbox are ready:

```
  assert Only_Ready_In_Spool_And_Mailbox is
     { M1:Message | M1.Ready } =  { M1:Message | (∃ S1:Spool :: M1 ∈ S1.Queue)} ∪
                                  { M1:Message | (∃ U1:User :: M1 ∈ U1.Mailbox) };
```

Theorem 6. A message cannot be in both spool and mailbox:

```
  assert Not_Both_Mailbox_And_Spool is
     {M1:Message | (∃ S1:Spool :: M1 ∈ S1.Queue)} ∩ {M1:Message | (∃ U1:User :: M1 ∈ U1.Mailbox)} = ∅;
```

Theorem 7. A message is at most in one mailbox:

```
  assert Unique_Messages is
     ∀U1:User :: (∀M1:Message | M1 ∈ U1.Mailbox ::¬(∃ U2:User | M1 ∈ U2.Mailbox ::U2 /= U1));
```

Theorem 8. A message body is not seen by anybody else but the sender and recipient.

```
  assert Read_Only_Mine is
     (∀ U1:User :: (∀ M1: Message | M1 ∈ U1.Reading :: M1.Recipient = U1)) ∧
     (∀ U1:User :: (∀ M1: Message | M1 ∈ U1.Sending
                                    :: ¬ (U2:User |  U2 ≠ U :: M ∈ U2.Sending ∪ U2.Reading) ∧
     (∀ M1: Message | M1.Idle :: M1.Body = 0 ∧ M1.Recipient = null);
```

We assume that a user can see the body only when the message is in U.Reading ∪ U.Sending. This is ensured by allowing reading only to users that are indicated by the variable Recipient. When a message is disposed, i.e. state is changed to idle, the old contents of the body is wiped off.

Theorem 9. A sent message has both recipient and body:

```
  assert Send_Complete_Only is
     (∀ M1:Message | M1.Ready :: M1.Body > 0 ∧ M1.Recipient ≠ null) ∧
     (∀ M1:Message | (∃ U1:User :: M1 ∈ U1.Reading) :: M1.Ready);
```

Theorem 10. Messages can only be read from mail box (U.Reading is a subset of U.Mailbox.):

```
  assert Read_From_Mailbox is
     ∀ U1:User :: {M1:Message | M1 ∈ U1.Reading} ⊂ {M1:Message | M1 ∈ U1.Mailbox};
```

Theorem 11. A message cannot be both in mailbox and in preparation (stronger than theorem 4.):

```
  assert Mailbox_Sending_Distinct is
     ∀ U1:User :: U1.Sending ∩ U1.Mailbox = ∅;
```

3.1.5. A window interface to the mail system

Next we will specify a windowing user interface to our electronic mail system.

A new class for windows is needed. A window can be either mapped (visible) or unmapped (not visible). While mapped, the window is displaying some message.

```
class Window is
    state *Unmapped, Mapped(M:Message);
end;
```

The user has a set of visible windows that can be used either for sending or reading of messages. The maximum number of windows - i.e. the number of "open" messages for a single user is not limited.

```
extend User by
    Windows : set Window; initially Windows = Ø;
end;
```

Actions Start_Send and Read must be refined to have a participating window. This window has to become mapped and the window must be added to the set of user's active windows.

```
refined Start_Send by ... W:Window is
when ... W.Unmapped do
    ...
    → W.Mapped(M);
    assert (W ∈ U.Windows);
    U.Windows := U.Windows ∪ {W};
end;

refined Read by ... W:Window is
when ... W.Unmapped do
    ...
    → W.Mapped(M);
    assert ¬ (W ∈ U.Windows);
    U.Windows := U.Windows ∪ {W};
end;
```

In actions Send, Cancel, Keep and Dispose the window must be unmapped and removed from the set of active windows:

```
refined Send by ... W:Window is
when ... M = W.Mapped.M do
    ...
    →W.Unmapped;
    assert W ∈ U.Windows;
    U.Windows := U.Windows − {W};
end;

refined Cancel by ... W:Window is
when ... M = W.Mapped.M do
    ...
    →W.Unmapped;
    assert W ∈ U.Windows;
    U.Windows := U.Windows − {W};
end;

refined Keep by ... W:Window is
when ... M = W.Mapped.M do
    ...
    assert W ∈ U.Windows;
    U.Windows := U.Windows − {W};
    → W.Unmapped;
end;

refined Dispose by ... W:Window is
when ... M = W.Mapped.M do
    ...
    U.Windows := U.Windows − {W};
    → W.Unmapped;
end;
```

A window is added to actions Add_Recipient and Add_Body to describe the fact that the user will give the recipient and body by typing at the window:

```
refined Add_Recipient by ... W:Window is
when ... M = W.Mapped.M do
    ...
end;

refined Add_Body by ... W:Window is
when ... M = W.Mapped.M do
    ...
end;
```

3.1.6. Theorems for windows

Theorem 12 All mapped windows belong to a user and no unmapped windows belong to a user:

```
assert Mapped_Owned_By_User is
    (∀ W1:Window | W1.Mapped :: (∃ U1:User :: W1 ∈ U1.Windows)) ∧
    (∀ W1:Window | W1.Unmapped :: ¬ (∃ U1:User :: W1 ∈ U1.Windows));
```

Theorem 13 A window can be owned at most by one user:

```
assert Window_Not_Owned_Two_Users is
    ∀ M:Window :: (∀ U:User | M ∈ U.Windows :: ¬ (∃ U1:User | U1 ≠ U :: M ∈ U1.Windows));
```

Theorem 14 Windows contain only messages that a user is sending or reading (Notice that theorem 4 ensures that messages are at most in one U.Reading or U.Sending.):

```
assert My_Messages_In_Window is
    ∀ W1:Window :: ∀ U1:User | W1 ∈ U1.Windows
                                :: (W1.Mapped.M ∈ U1.Reading ∨ W1.Mapped.M ∈ U1.Sending);
```

4. Conclusions and directions for future work

The DisCo language and tool can describe the behavior of user interfaces in a formal manner. The tool can be used for experimentation, and temporal logic of actions can be used for formal proofs.

The main advantages of our approach over previous methods are the description of concurrent user interfaces in an abstract and implementation independent way, and the support for stepwise refinement by the structuring capabilities of the DisCo language.

In future we plan to experiment with specifications of more complicated user interfaces. Especially the suitability of temporal logic of actions for complex user interface specifications needs further experiments. The examples should also be refined by the constraints caused by the implementation architecture. It would be valuable to know how the implementation structure limits concurrency, and how we can recognize those limitations from the specification.

We should also test our approach for verification of user interface properties like consistency and reversibility.

Currently, the tool and method have no support for user interface implementation and prototyping, the current DisCo tool can only specify the behavior of the user interface, and no representation issues can be handled. In future, the animation capabilities of DisCo tool could be extended by typical user interface input and output components.

5. Acknowledgements

The idea of using joint actions as a specification tool was originally presented by Ralph Back and Reino Kurki-Suonio. The DisCo language was designed by Hannu-Matti Järvinen. Olavi Eerola and Tatu Männistö have worked with me in the implementation of the tool. Reino Kurki-Suonio gave also many valuable suggestions to previous versions of this paper. Email discussions with Matti Pettersson have also helped me in clarifying the ideas.

6. References

[1] Back, R. J. R., Kurki-Suonio, R., Decentralization or process nets with a centralized control. *Distributed Computing 3*, 3, May 1989, 73-87.

[2] Back, R. J. R., Kurki-Suonio, R., Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems 10*, 4, October 1988, 513-554.

[3] Booth, P., *An Introduction to Human-Computer Interaction*. Lawrence Erlbaum Associates, 1989 (Reprinted 1990).

[4] Bowen, J., *Formal specification of window systems*. Technical Monograph PRG-74, Oxford University Computing Laboratory, Programming Research Group, June 1989.

[5] Chi, U. L., Formal specification of user interfaces: a comparison and evaluation of Four Axiomatic Approaches. *IEEE Transaction on Software Engineering 11*, 8, August 1985, 671 - 685.

[6] Cohen, B., Harwood, W. T., Jackson, M. I., *The Specification of Complex Systems*. Addison-Wesley Publishing Company, 1986.

[7] Green, M., A survey of three dialogue models. *ACM Transactions on Graphics 5*, 3, July 1986, 245-275.

[8] Harel, D., Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8*, 1987, 231-274.

[9] Hayes, I. (ed.), *Specification case studies*. Prentice-Hall International (UK) Ltd, 1987.

[10] Hill, R. D., Supporting concurrency, communication and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Transactions on Graphics 5*, 3, July 1986, 179-210.

[11] Järvinen, H-M., Kurki-Suonio, R., The DisCo Language, Tampere University of Technology, Software Systems Laboratory, Report 8, March 1990.

[12] Järvinen, H-M., Kurki-Suonio, R., Sakkinen, M., Systä, K., Object-oriented specification of reactive systems. Proc. *12th International Conference on Software Engineering*, Nice, France, 1990, IEEE Computer-Society Press, 1990, 63-71.

[13] Lamport L., *A Temporal Logic of Actions*. Research Report 57, Digital Systems Research Centre, 1990. (A revised and extended version in preparation).

[14] Pettersson, M., *Specifying the User Interface*, Licentiate Thesis, University of Tampere (Manuscript).

[15] Pfaff, D. (Ed.), *User Interface Management Systems*. Springer-Verlag, New York, 1985, 67-79.

[16] Schiele, F., Green, T., HCI formalisms and cognitive psychology: the case of Task-Action Grammar. In *Formal Methods in Human-Computer Interaction*, Edited by Harrison, M. and Thimbleby H., Cambridge University Press, 1990.

[17] Systä, K., A graphical tool for specification of reactive systems. Proc. *Euromicro'91 Workshop on Real-Time Systems*. Paris, France, June 1991, IEEE Computer Society Press, 1991, 12-19.

[18] Tanner, P. P., Buxton W. A. S., Some issues in future user interface management system (UIMS) development. In *User Interface Management Systems*. G. Pfaff, Ed. Springer-Verlag, New York, 1985, 67-79.

[19] Wasserman, A., I., Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transaction on Software Engineering 11*, 8, August 1985, 699 - 713.

ONE MORE EXPONENTIAL ALGORITHM FOR ESTABLISHING SATISFIABILITY OF PROPOSITIONAL FORMULA

Mati Tombak

Department of Computer Science
University of Tartu
EE2400 Tartu J.Liivi 2
E-mail: mati@csd.ut.ee
ESTONIA

Let $a_1, \dots, a_n$ be propositional variables ($n > 0$). Propositional formula in conjunctive normal form (CNF) is

$$A = \bigwedge_{i=1}^{p} A_i \qquad (1 \le p \le 3^n), \tag{1}$$

where

$$A_i = \bigvee_{j=1}^{k_i} x_{ij} \qquad (1 \le k_j \le n) \tag{2}$$

are clauses and

$$x_{ij} \in \{a_l, \bar{a}_l : 1 \le l \le n\} \tag{3}$$

are literals. Propositional variable a_l in (3) is a variable of literal x_{ij} ($a_l = \mathbf{prop}(x_{ij})$). We assume that all clauses in (1) are different and that all propositional variables in every clause are different. If $k_i = n$ in (2), then A_i is *maxclause*. If every clause in (1) is maxclause, then A is *complete CNF*. For complete CNF $1 \le p \le 2^n$. It is a well-known fact, that every clause A_i in complete CNF determines one evaluation, for which A is false: $\mathbf{neg}(A_i) = (\sigma_{i1} \dots \sigma_{in})$, $(\sigma_{i1} \dots \sigma_{in}) \in \{0,1\}^n$, where $\sigma_{ij} = \mathbf{neg}(x_{ij})$ and

$$\mathbf{neg}(x_{ij}) = \begin{cases} 1, \text{if } x_{ij} = \bar{a}_{ij}, \\ 0, \text{if } x_{ij} = a_{ij}. \end{cases}$$

Hence, complete CNF $A = \bigwedge_{i=1}^{p} A_i$ is satisfiable iff $p < 2^n$, and $2^n - p$ is the number of true evaluations of A.

Let $\mathcal{A}$ be a CNF. If we transform $\mathcal{A}$ into complete CNF, then every clause A_i in $\mathcal{A}$ generates a certain set of maxclauses, $\mathbf{gen}(A_i)$. Let

$$P_i = \{a_i, \ldots, a_n\} \setminus \{\mathbf{prop}(x_{i1}), \ldots, \mathbf{prop}(x_{ik_i})\},$$

i.e. P_i is the set of all propositional variables, which do not appear in clause A_i. So

$$\mathbf{gen}(A_i) = \{(\bigvee_{j=1}^{k_i} x_{ij}) \vee (\bigvee_{a_{il} \in P_i} a_{il}^{\sigma_l}) : \ \sigma_l \in \{0,1\}, 1 \leq l \leq |P_i|\}$$

and complete CNF for $\mathcal{A}$ is

$$\mathbf{gen}(\mathcal{A}) = \bigcup_{i=1}^{p} \mathbf{gen}(A_i).$$

Clauses A_i and A_j are *separated* if $\mathbf{gen}(A_i) \cap \mathbf{gen}(A_j) = \emptyset$. Let $\mathbf{weight}(A_i) = |\mathbf{gen}(A_i)|$. It is easy to see that $\mathbf{weight}(A_i) = 2^{n-k_i}$. It is obvious therefore that the number of maxclauses in complete CNF $\mathbf{gen}(\mathcal{A})$ is equal to $\sum_{i=1}^{p} \mathbf{weight}(A_i) = \sum_{i=1}^{p} 2^{n-k_i}$ iff, for every $i \neq j$, clauses A_j and A_j are separated.

Iwama's algorithm for establishing satisfiability (see [2]) calculates the number of maxclauses in $\mathbf{gen}(\mathcal{A})$ using the inclusion-exclusion principle. We will try to transform CNF $\mathcal{A}$ to equivalent CNF $\mathcal{A}'$ with separated clauses.

Theorem 1. *Clauses A_i and A_j are separated if and only if there exists literal x so that $x \in A_i$ and $\bar{x} \in A_j$.*

Proof. 1) Suppose there does not exist literal x so that $x \in A_i$ and $\bar{x} \in A_j$. We can write A_i and A_j in form

$$A_i = x_1 \vee \ldots \vee x_k \vee y_1 \vee \ldots \vee y_l,$$

$$A_j = x_1 \vee \ldots \vee x_k \vee z_1 \vee \ldots \vee z_m,$$

where

$$\{\mathbf{prop}(y_1), \ldots, \mathbf{prop}(y_l)\} \cap \{\mathbf{prop}(z_1), \ldots, \mathbf{prop}(z_m)\} = \emptyset,$$

Let $\{b_1, \ldots, b_r\}$ be the set of all propositional variables from $\{a_1, \ldots, a_n\}$, which do not occure in $A_i \cup A_j$, $(r \geq 0)$. Then

$$\mathbf{gen}(A_i) = \{x_1 \vee \ldots \vee x_k \vee y_1 \vee \ldots \vee y_l \vee z_1^{\alpha_1} \vee \ldots \vee z_m^{\alpha_m} \vee b_1^{\beta_1} \vee \ldots \vee b_r^{\beta_r} :$$

$$(\alpha_1, \ldots, \alpha_m) \in \{0,1\}^m,\ (\beta_1, \ldots \beta_r) \in \{0,1\}^r\},$$

$$\mathbf{gen}(A_j) = \{x_1 \vee \ldots \vee x_k \vee y_1^{\gamma_1} \vee \ldots \vee y_l^{\gamma_l} \vee z_1 \vee \ldots \vee z_m \vee b_1^{\beta_1} \vee \ldots \vee b_r^{\beta_r} :$$

$$(\gamma_1, \ldots, \gamma_l) \in \{0,1\}^l,\ (\beta_1, \ldots \beta_r) \in \{0,1\}^r\}$$

and

$$\mathbf{gen}(A_i) \cap \mathbf{gen}(A_j) = \{x_1 \vee \ldots \vee x_k \vee y_1 \vee \ldots \vee y_l \vee z_1 \vee \ldots \vee z_m \vee b_1^{\beta_1} \vee \ldots \vee b_r^{\beta_r} :$$

$$(\beta_1, \ldots \beta_r) \in \{0,1\}^r\},$$

i.e. $\mathbf{gen}(A_i) \cap \mathbf{gen}(A_j)$ consists of 2^r maxclauses and is not empty for every $r \geq 0$.

2) Suppose there exists propositional variable a so that $a \in A_i$ and $\bar{a} \in A_j$. In this case every clause from $\mathbf{gen}(A_i)$ contains a and every clause from $\mathbf{gen}(A_j)$ contains $\bar{a}$. Therefore $\mathbf{gen}(A_i) \cap \mathbf{gen}(A_j) = \emptyset$.

Algorithm 1: separate($A\&B$)

Let A and B be two nonseparated clauses, i.e.

$$A = x_1 \vee \ldots \vee x_k \vee y_1 \vee \ldots \vee y_l,$$

$$B = x_1 \vee \ldots \vee x_k \vee z_1 \vee \ldots \vee z_m,$$

where

$$\{\mathbf{prop}(y_1), \ldots, \mathbf{prop}(y_l)\} \cap \{\mathbf{prop}(z_1), \ldots, \mathbf{prop}(z_m)\} = \emptyset.$$

Suppose $m > l$.

If $l = 0$, *then* **separate**$(A\&B) = A$

else **separate**$(A\&B) = A\&B_1\& \ldots \&B_l$, where

$$B_1 = B \vee \bar{y}_1,$$

$$B_2 = B \vee y_1 \vee \bar{y}_2$$

$$B_3 = B \vee y_1 \vee y_2 \vee \bar{y}_3$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$B_l = B \vee y_1 \vee y_2 \vee \ldots \vee \bar{y}_l.$$

Theorem 2. **separate**$(A\&B)$ *is CNF, equivalent to* $A\&B$ *with pairwise separated clauses.*

Proof. 1) A is separated from B_i $(1 \leq i \leq l)$, because A contains y_i and B_i contains $\bar{y}_i$. B_i is separated from B_j $(1 \leq i < j \leq l)$, because B_i contains $\bar{y}_i$, and for every $j > i$, B_j contains y_i.

2) If $l = 0$, then A subsumes B (see [1]), and therefore **separate**$(A\&B) = A$ is equivalent to $A\&B$.

Suppose $l > 0$, and let $\sigma \in \{0,1\}^n$ be an arbitrary evaluation for which $A\&B$ is **true**. By construction of B_i, one can see that B subsumes B_I $(1 \leq i \leq l)$. Hence B_i is **true** for each evaluation for which B is **true**. Therefore $A\&B_1\&\ldots\&B_l$, is **true** for evaluation σ.

Let $\sigma \in \{0,1\}^n$ be an arbitrary evaluation for which $A\&B$ is **false**. If A is **false** for σ, then $A\&B_1\&\ldots\&B_l$, is **false** for σ. Suppose, that A is **true** for σ. Then B is **false** for σ, i.e all literals $x_1, \ldots, x_k, \ldots, z_1 \ldots, z_m$ are **false** for σ. At least one of literals y_i has to be **true** (remember that A is true for σ). Let i_0 be the minimal i such that y_i is **true**. Then $B_{i_0} = x_1 \vee \ldots \vee x_k \vee z_1 \vee \ldots \vee z_m \vee y_1 \vee y_{i_0-1} \vee \bar{y}_{i_0}$ is **false**, and hence $A\&B_1\&\ldots\&B_l$ is **false** for evaluation σ.

If we transform CNF A into CNF A', using **Algorithm 1**, some new clauses may be nonseparated. Consequently, if the number of clauses in A is p and the number of clauses in A' is p', then we have to check $(p')^2$ pairs of clauses. To be sure, that this method for establishing satisfiability does not violate the hypothesis P=NP, we have to find an example of CNF A for which number of clauses grows exponentially.

Example 1. $A = \{a_i \vee b_i \vee c_i, a_i \vee \bar{b}_i \vee \bar{c}_i, \bar{a}_i \vee b_i \vee \bar{c}_i, \bar{a}_i \vee \bar{b}_i \vee c_i : i = 1, \ldots, k\}$. The number of clauses in A is $4k$. After applying **Algorithm 1** we get A' with number of clauses $p' = \sum_{i=1}^{k} 4^i$.

Example 2. Let $G = (V, E)$ be a graph, where $V = \{a_1, \ldots, a_n\}$ is the set of nodes and E is the set of edges. We build CNF A_G with propositional variables $a_1, \ldots, a_n$, which consists of all clauses $\bar{a}_i \vee \bar{a}_j$ such that $(a_i, a_j) \notin E$. It is easy to see, that every satisfying evaluation for A_G represents one clique of G (including trivial ones). If we transform CNF A_G into separated CNF, we can compute the number of true evaluations of A_G, which is the number of all cliques of G. If we consider the graph $G = (N, E)$, where $N = \{a_1, b_1, \ldots, a_k, b_k\}$ and $E = (N \times N) \setminus \{(a_i, b_i) : 1 \leq i \leq k\}$, then CNF A_G will be $\bigwedge_{i=1}^{k} (\bar{a}_i \vee \bar{b}_i)$ and **Algorithm 1** gives the separated formula A_G', which consists of $2^k - 1$ clauses.

References:

[1] Davis M., Putnam H. A computing procedure for quantification theory. – J. Assoc. Comput. Mach., 1960, 7. pp. 201-215.

[2] Iwama K. CNF satisfiability test by counting and polynomial average time. – SIAM J. Comput., 1989, vol. 18 No.2, pp. 385-391.

TÜ 93.294.200.9,44.9,25.