Volume 9 Number 3

ACTA CYBERNETICA

Szeged, 1990

Information for authors: Acta Cybernetica publishes only original papers in English in the field of computer sciences. Review papers are accepted only exceptionally. Manuscripts should be sent in triplicate to one of the Editors. The manuscript must be typed double-spaced on one side of the paper only. For the form of references, see one of the articles previously published in the journal. A list of special symbols should be supplied by the authors.

Towards Languages that Support Program Derivation, or Control Modularity Considered Harmful*

REINO KURKI-SUONIO

Tampere University of Technology
Box 527, SF-33101 Tampere, Finland
e-mail: rks@tut.fi.

Abstract

Of current trends in programming languages, the paper concentrates on the need to support formal specification and derivation of software, mainly in the context of reactive systems that are in continual interaction with their environments. The non-programming facilities of operational specifications are briefly analyzed, and their inclusion in design oriented specification languages is considered. Early commitment to control-oriented decisions is found harmful, which leads to a language basis with implicit concurrency and no notion of control flow. The advantages of this approach for certain intuitively natural methods of program derivation are demonstrated. The paper ends with general comments about the diversification of languages along the dimension of specification, design, prototyping, and implementation.

1. Introduction

Software objects are artifacts that cannot be classified either as concrete objects or as pure abstractions. An executable machine language program might be considered a concrete object with the original source code as its abstraction. However, source programs are also executable, at least in principle, and therefore equally concrete. On the other hand, no matter which level of languages is considered, each program is an abstraction of something that gets concrete only in its physical execution. Therefore, as pointed out by Lamport [La89], every program is a specification, and some specifications are implementations of other specifications.

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

The evolution of programming languages shows continual rise in the *level of abstraction*, which means that the specification nature of programs becomes more and more obvious. The programmer needs to worry less about "concrete" or "efficient" representations, and can concentrate more on "abstractions" that are easier to reason about and can be automatically compiled into lower levels. Some perspective on this trend is provided by the remark by Parnas [Pa85] that the term "automatic programming" was probably used for the first time in the 1940s in a paper by Saul Gorn on the possibilities of building a simple assembler.

Another direction, which is evident especially in the current development of so-called object-oriented languages [SW87], is towards structures that facilitate *prototyping*, easy modification, and evolutionary development of systems. It is not entirely clear, however, whether the resulting language flexibility can be adequately reconciled with the security and provability requirements of many applications.

Thirdly, only very small programs can be written directly, and software maintenance involves changes in their specifications. The increasing significance of program reliability and correctness therefore requires explicit support for *software derivation from specifications*. Although most abstractions in programming languages, such as block structure, subroutines, and modules, support various methodologies for programming-in-the-small, explicit support for program derivation has usually not been considered a programming language issue. Instead, extra-linguistic tools have been provided in operating systems and programming environments for combining pieces of software, as well as for version control and related aspects of software maintenance. Even though some languages like Lisp and Smalltalk come with integrated programming environments, and Ada was claimed to extend the scope of programming languages towards programming methodologies and utilization of program libraries, only elementary language support is presently available for the derivation of software.

Any substantial support for program derivation requires the use of *formal specifications* instead of the informal and semiformal approaches that still dominate in programming practice. Like programs, formal specifications cannot be given directly, and their derivation is similar to that of programs. Existing components are used in this process, new properties are introduced in a stepwise manner, the level of abstraction is lowered for reasons like efficient implementability, and the results need to be verified and validated. Attempts to support software derivation have lead to experimental *wide spectrum languages* [Ba&89] within which specifications can be transformed into implementations through correctness-preserving transformations.

Notice that it is not only the result of the derivation process that is important, since the higher levels provide important abstractions and insight that are lacking in the final (high-level language) form. The development of tools to analyze finished designs, in order to recover the insight that was never made explicit in the first place, is a backward approach, and is no substitute for the derivation of programs in a manageable way from higher-level specifications. In particular this is currently a problem with concurrent systems, where most so-called specification languages have no better abstractions of concurrency than those available in programming languages.

The above trends in programming languages provide the background for this paper. We shall mainly focus on *reactive computations* [Pn86], in which the system is in continual interaction with its environment. Obviously, traditional input-output computations are a special case of these.

The structure of the paper is as follows. First we address the question of what distinguishes some specifications from programs. In Section 3 the problems of concurrency — or, rather, independence of sequentiality — lead us to a somewhat radical conclusion about the usefulness of traditional control flow oriented modularity. When the notion of control flow is abandoned, *statecharts* [Ha87] are shown to be suitable for the structuring of the global state. In Section 4 we demonstrate how independence of control decisions leads to a language basis that is suited for certain intuitively natural methods of program derivation. The paper ends with some general comments about the diversification of languages along the dimension of specification, design, prototyping, and implementation.

The paper is heavily influenced by and biased towards *joint action systems*, developed together with Ralph Back [BK83, BK88a, BK88b]. Case studies of such systems and design methods for their development have been investigated in [BK83, BK84, Ku86, KK88, Ku89], and [KJ89, Jä&89] report on an experimental specification language DisCo (for *Dis*tributed *Co*operation) that is based on these ideas. The reader is also reminded of the close similarity between joint action systems and the Unity language by K. Mani Chandy and Jay Misra [CM88a].

2. Operational Specifications and Programs

Formal specifications are commonly understood to express *safety* and *liveness* properties of programs [AS85]. Informally, the former state that nothing bad will ever take place, while the latter express the requirement that the desired good things will eventually happen. Temporal logics are well-established formalisms in which such properties can be expressed [MP83, Pn86]. Obviously, such specifications do not cover all formalizable requirements: statistical efficiency properties, for instance, remain inexpressible.

Programs are operational and executable specifications. In general, an operational specification consists of two components: a *generative* mechanism that is based on computational steps, and a set of *constraints* [Fe87]. The former generates a collection of potentially possible (finite or infinite) computations, which is then restricted to a subset by the constraints. Notice that operationality is more a viewpoint than a well-defined property. Temporal logic specifications, for instance, can be understood in these terms by viewing the temporal properties as constraints on the implicitly generated collection of all possible sequences of events.

In programs the emphasis is on the generative mechanism, which determines safety properties only. Liveness properties are given by implicit constraints that exclude those finite computations that have not yet terminated, as well as such infinite computations that do not satisfy certain *fairness* requirements [Fr86]. Since we are interested in programs as abstract specifications, we ignore here the practical non-constructivity and non-verifiability issues of fairness constraints [Di88, SL88, CM88b].

Since each program is a specification, and all specifications have an operational interpretation, the question arises whether there is any fundamental distinction between specifications and programs. Is the difference only in efficiency, whose significance changes rapidly with the development of hardware and software technology?

Based on Dijkstra's weakest precondition calculus of predicate transformers [Di76], Ralph Back was the first to introduce a mixed formalism in which specifications and programs coexist on equal basis [Ba78, Ba80, Ba88a]. For specifications, one of Dijkstra's healthiness conditions for programs had to be relaxed. The reason was in the need for *unbounded nondeterminism*. By this we understand the selection of a value that satisfies a given condition, when the number of potential alternatives is infinite. If several alternatives satisfy the condition, then the choice between them is nondeterministic.

Unbounded nondeterminism is equally non-constructive as fairness. Notice that describing the input of an arbitrary integer as a single event leads to unbounded nondeterminism, while replacing this by a sequence of separate input events for an arbitrary number of digits requires fairness instead.

Similarly to the bounded nondeterminism in [Di76] this nondeterminism is of the *demonic* variety, which means that each possible choice must lead to a correct computation.

Recently Back and von Wright [BW89] have extended this work to remove also the other healthiness conditions, except for monotonicity. The results are mathematically appealing, and they can be interpreted to lead to two further possibilities in specification languages: *angelic nondeterminism*, which only requires that at least one of the alternative choices leads to a correct computation, and *miracles*, which miraculously succeed in establishing conditions by impossible assignments. Similar generalizations have been found desirable also elsewhere [dB80, Mo87, Ne87, Mo88a, Mo88b], and their need for describing practical specification languages has been observed [Mo88b, Ba88b].

From a more practical viewpoint, at least the following quasi-executable facilities have been found useful in operational specifications:

- The generative mechanism may involve *unbounded nondeterminism* [BB87, KJ89]. Notice that unbounded nondeterminism is implicitly present in specifications that do not use an explicit generative mechanism, as is the case with temporal logic [Pn86] and algebraic specifications [Ba89].
- A computation may refer to its *past history* without having explicitly recorded it [BG79, Fe87, AL88].
- There can be "*oops conditions*" that are not allowed to become true [BG79, Fe87]. If the generative mechanism leads to such situations, the constraints are assumed to exclude those computations.
- A computation may contain *prophetic references* to its future [BG79, Fe87, AL88]. The constraints are then assumed to exclude computations where such predictions would turn out to be incorrect.

Of these facilities, references to past history are the least problematic for direct implementation, as further recording of history can always be added. Unbounded nondeterminism also looks rather innocent. Existential quantification provides, however, extremely poweiful possibilities for implicit solutions of problems for which no algorithms are known.

In the absence of unbounded nondeterminism, oops conditions cause no obstacles for classical input-output computations, and backtracking is a standard technique for their implementation. With unbounded nondeterminism this need not succeed, however, and with reactive computations the situation becomes totally different, as there is no way for an implementation to withdraw interactions that have already taken place between the system and its environment.

The situation with prophetic references is no simpler, as they can be understood as nondeterministic guesses about the future, combined with oops conditions to be evaluated later.

From this discussion we can conclude that all specifications are not programs, even if efficiency considerations are totally ignored, and that the differences are even more significant for reactive systems. A good specification language therefore needs facilities that do not satisfy the constructivity criteria for programming. In view of the problems in removing their use by systematic transformations [LF82] one should, however, be cautious in introducing them in design-oriented specification languages. Some form of unbounded nondeterminism seems to be a minimum that is required by reasonable design methodologies [Ku89].

3. On Concurrency, Control Modularity, and Structure of Global State

Concurrency is often thought of as an auxiliary feature that can be added afterwards to any description language. Its use seems to complicate matters, and one may therefore try to avoid it as far as possible. Even when concurrency is crucial for the design, one may resort to the backward approach of first designing a sequential solution and then parallelizing this. Notice that for the description of reactive systems concurrency is always essential, even when the system is to be implemented as a single process, since the environment works concurrently with the system itself.

Our view of concurrency is different: to us sequentiality or any particular choice of parallelism is an implementation-oriented design decision, of which specifications should be independent. We therefore argue that good support for deriving software from specifications cannot be provided by amending a sequential base language with additional constructs for concurrency. In fact, instead of having specific constructs for sequential and concurrent control, the base language should be independent of any such choices. In other words, it is not concurrency in itself that is important, but *independence of control decisions.*

In mathematics it is often the case that a more general formulation makes a problem easier to manage. The same has also been observed in programming. For instance, the advantages of nondeterminism over strictly deterministic descriptions were clearly demonstrated by Dijkstra [Di76], even in situations where the programmer would later restrict the design by purely deterministic choices. The situation with concurrency is similar, and we claim that it is the conventional control-oriented modularity that has prevented us from realizing this.

Any execution model of computing involves a *state* (memory, registers, variables) and *actions* (instructions, statements, transitions) that modify this state. Conventionally the state is partitioned into a *data* part (accumulators, variables) and *control* part (instruction counters, control states). In the light of sequential programming and von Neumann computer architecture this approach is natural, and the control-oriented modularity of structured programming may therefore seem inherent to any well-structured description of computations. It is this assumption that we challenge, and in doing so we need to abandon the early partitioning of state into data and control parts. For further discussion on why control modularity is especially harmful in the design of parallel programs, the reader is referred to Chandy and Misra [CM88a].

Without the special role of control state the conventional control flow oriented modularity becomes inapplicable. Even though independence of control decisions can easily be achieved, the result may be chaotic. Similar ideas have been used (with different motivation) in production system languages like OPS5 [FD77], and the drawbacks are well-known. With no notion of control flow there is no built-in structure in the collection of actions, and one is easily led to encode the missing control flow in unstructured collections of bits and flags. Obviously, such an unmodular system is even more difficult to understand than one where the control state has been explicitly separated from data.

In the following our purpose is to show that abandoning control flow does not imply lack of structuring and modularity. On the contrary, our conclusion will be that this can lead to another kind of modularity that is especially suited for software derivation. We start by inspecting how to impose structure on a state that has no dedicated control components.

We assume that the global state of a system is partitioned into components called *objects*. The state of a single object is called its *local* state. From an implementation point of view objects may be thought of as either data structures or processes. Avoiding the notion of control flow implies that no distinction is made between passive objects (data structures) and active agents (processes). Therefore, objects require structuring capabilities that are equally suited for both.

Harel's *statecharts* [Ha87] turn out to be an ideal visual formalism for this purpose. From the viewpoint of active agents, their hierarchical state structure generalizes the notion of ordinary finite-state systems, and can be interpreted as the nested control structures in conventional high-level languages. Associating data items with the states makes this analogy even more complete. On the other hand, from the viewpoint of passive objects, statecharts can be interpreted as record structures containing tagged unions of alternatives. The state transitions of a statechart correspond to the actions of the system, which are now separated from the structure of the global state.

As a simple example, Figure 1 gives a statechart description of database clients. On the outermost level a client object has three exclusive states: *idle, starting* a transaction, and *engaged* in one. When *engaged,* a client is either *ready* to issue another request or *waiting* for a response. A *waiting* client is expecting a response either to an end request *(ending)* or to a read or write request *(accessing)*. Transitions of the statechart are labeled by identifiers in italics, referring to actions whose descriptions have been omitted. For simplicity, the data items that are associated with the states have also been left out.

This simple example illustrates *or* decomposition of states, in which case the immediate substates of a state are exclusive alternatives to each other. Another useful possibility is *and* decomposition, which means that the actual state is a Cartesian product of states in each component.

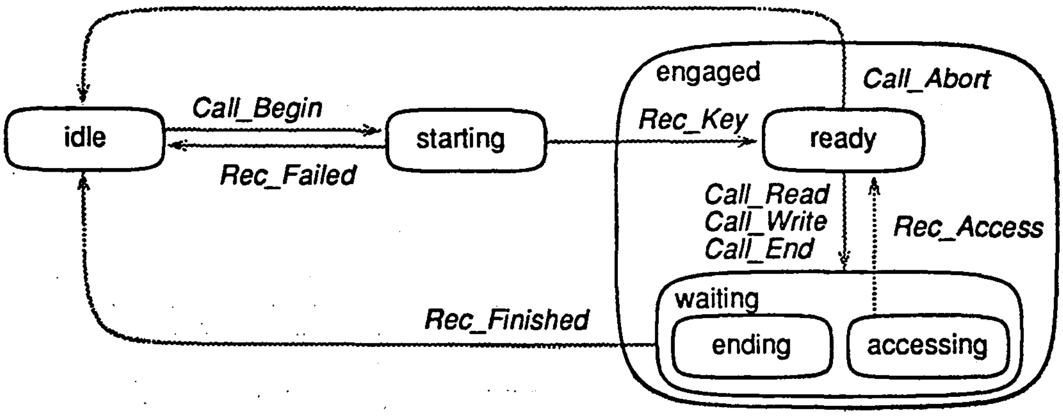


Figure 1. Statechart description of database clients

In the DisCo language the same information (except for the transitions) would be given in textual form as the following class declaration:

```
class Client is
    state idle, starting, engaged;
    extend engaged by
        state ready, waiting;
        extend waiting by
            state accessing, ending;
        end waiting;
    end engaged;
end Client;
```

The keyword **extend** is used to emphasize the possibility to abstract away internal structure of states, and to extend them later with further detail.

When the global state of a system is partitioned into objects with local states, the description of actions needs to be separated from objects. As there is no control flow to *enable* an action that can be executed next, each action requires a *guard* expression to determine its enabledness. Since any number of actions can be enabled at the same time (even though only one is selected for execution), nondeterminism is inherent in this kind of systems.

In principle, any number of objects may participate in an action in the sense that their local states are required and possibly updated in its execution. From the viewpoint of active agents the execution of an action can be interpreted as follows: first the participants determine by mutual communication that the action is enabled and perform a joint handshake to become committed to its execution; while committed to the action they exchange the data that are necessary for each participant to update its own local state appropriately; after updating its own local state and providing the other participants with the data they require, each object becomes free for another action. On the other hand, from the viewpoint of passive objects we can think in terms of a centralized scheduler that evaluates guards and triggers the execution of enabled actions. It is important that both views are equally possible, i.e., that the base language has no explicit constructs for concurrency or communication, and is therefore independent of any control decisions.

4. Support for Program Derivation

The main topic of this paper is the development of language support for program derivation. So far we have described a language basis that is independent of control decisions; in this section we shall demonstrate how such a base language is suited for such useful methods of program derivation that would be much more complicated to express with conventional control flow oriented languages.

The main design method to be considered here is superimposition or *superposition*. This is a layered approach where, starting from an initial solution that satisfies some basic requirements, further properties are imposed without violating those already established. Superposition has been mainly used in connection with distributed systems, and different formulations of it have slightly different properties. An early use of the technique and the term was in [DS80]; in [LS84] it was described as the reverse of a protocol verification method; recently it has been suggested as a control structure for concurrent and distributed programming [Ka87, BF88]; in [CM88a] it was introduced as one of the main facilities for designing Unity programs in a modular fashion; in connection with joint action systems the technique has been applied in [BK83, BK84, Ku86, Ku89].

Here we introduce the method by a simple example that has sometimes been used in comparing different specification and design methods. The problem is to describe a doctors' office, which involves patients that are cured by doctors, and a receptionist that organizes the free doctors to treat the waiting patients.

We start with the simplest possible projection of the system that exhibits complete behavior by itself. In this case such a system contains only one kind of objects, patients, with two possible states, *well* and *sick*, and two kinds of actions: each patient that is *well* may become *sick*, and a *sick* patient may become *well*. In DisCo this could be described as follows:

```
system Patients is

    class patient is
        state well, sick;
    end;

    action get_sick by p: patient is
    when p.well do
        → p.sick;
    end;

    action get_well by p: patient is
    when p.sick do
        → p.well;
    end;

end Patients;
```

Each of the two actions has just one participant, a patient *p*, and a guard indicating that *p* is *well* or *sick*, respectively; the effect of the actions is to change the state of *p* as indicated by the state transition statements (→). Only the class declaration for the patient objects is given here; the actual creation of patient objects is assumed to be given separately in system initialization.

This first approximation of the system is easy to understand: patients just get sick and are cured nondeterministically. (For simplicity we omit here fairness questions like whether each *sick* patient is eventually cured.) Another layer is now superposed on this system, introducing the property that no patient is cured without a doctor. Each doctor has two states: *free*, or *busy* with a patient *p*. The state *well* of patients needs to be extended with two substates at this stage, to distinguish whether a cured patient has already checked out from the office or not. Action *get_well* is also refined to indicate the need of a doctor in this action, and a third action is introduced for releasing a cured patient from the office:

```
system Doctors with Patients is

    class doctor is
        state free, busy(p: patient);
    end;

    extend patient.well by
        state released, hide*checking_out;
    end;

    refined get_well by ... d: doctor is
    when ... d.free do
        → d.busy(p);
        ...
    end;

    action release by p: patient; d: doctor is
    when d.busy.p=p ∧ p.well.checking_out do
        → d.free;
        → p.well.released;
    end;

end Doctors;
```

Ellipses (...) belong to the language and indicate parts taken directly from the previous level. The refinement of *get_well* introduces an additional participant *d* and another conjuct to the guard, indicating that *d* must be free, and makes *d* become busy with patient *p*. The state *patient.well* is extended in such a way that a cured patient always enters the default substate *p.well.checking_out* (indicated by the star). This substate is hidden (**hide**) from the previous level in the sense that *get_sick* cannot be enabled in it. Therefore *p* has to participate in the new action release before getting sick again, i.e., a cured patient cannot get ill before leaving the office.

Provided that some number of doctors are initially created, the system is again complete, although it still lacks some of the required properties. In the next step we introduce a receptionist that organizes the free doctors and the waiting patients. Again, the creation of the initial state is omitted:

```
system Office with Doctors is

    class receptionist is
        pq: sequence patient;
        dq: sequence doctor;
    end;

    refined get_sick by ... r: receptionist is
    when ... do
        ...
        r.pq:=r.pq & p;
    end;

    refined get_well by ... r: receptionist is
    when ... p=first(r.pq) ∧ d=first(r.dq) do
        r.pq:=tail(r.pq);
        r.dq:=tail(r.dq);
        ...
    end;

    refined release by ... r: receptionist is
    when ... do
        r.dq:=r.dq & d;
        ...
    end;
end Office;
```

Notice that although the receptionist is needed in all actions, it does not create a bottleneck for a concurrent implementation. In *get_well*, for instance, the role of the receptionist is only to remove the doctor and the patient from their respective queues, after which it can start participation in some other action, while the doctor and the patient still continue in action *get_well*.

This example gives rise to the following general observations about superposition:

- It is a top-down design method in which even partially specified systems are given as complete systems exhibiting well-defined behavior.
- The global state of a system can be extended by adding new objects and by extending the local states of old ones. Statechart structuring of objects is especially suited for the addition of new substructures and new data components.
- New functionality can be added and new properties can be introduced by providing new actions and by refining the old ones. Atomicity of actions and absence of control flow for individual objects are significant for doing this smoothly. Additions and refinements are restricted to ones that do not affect the old state components.
- Nondeterminism of the system can be restricted by strengthening the guards of actions. For instance, the design may utilize unbounded nondeterminism until a basis for deterministic selections has been superposed.
- With the notions of objects and actions, all modifications have good locality: one logical change does not lead to several small changes in different places.

- The preservation of all safety properties can be guaranteed by language rules; the restrictions enforced are similar to what has been called complete compatibility in connection with object-oriented programming [WZ88]. Because of guard strengthening and the potential possibility for takeover by new actions, liveness properties have to be checked.

By this we hope to have demonstrated that, once the self-evidence of control flow oriented modularity is given up, it is possible to support effectively such intuitively natural approaches to structured derivation of programs that are quite complicated to manage with conventional language bases. In an ordinary multi-process program, for instance, a simple modification of a single action would correspond to changes scattered in the codes of all processes involved.

For brevity we have described here only one design method, superposition, which is based on a top-down approach. Similar observations concern, however, the bottom-up design method that is dual to superposition in the light of the above notions. This method introduces modularity with *communication-closed layers* [EF 82]. In our language it uses a mechanism called *inheritance* [KJ 89] and is especially suited for the development and utilization of reusable modules. As described in [Jä & 89], the mechanisms for supporting these two design methods can be understood as two well-structured variants of object-oriented inheritance. In other words, these ideas can also be described as an object-oriented approach to specification. Notice, however, the fundamental departure from conventional object-oriented programming that objects are not assumed to have individualistic behavior; the methods of individual objects are replaced by roles in cooperative, multi-object actions.

5. Concluding Remarks

In this paper we have investigated some novel language directions to which we may be led by the need to support program specification and derivation effectively, especially in the context of reactive systems. In particular, we hope to have demonstrated that early commitment to decisions on control is harmful for certain natural approaches to software derivation. Therefore, languages with a possibility for independence of control decisions are foreseen, and some capabilities of a simple experimental specification language of this flavor have been presented.

More generally, with this direction of language development the practical significance of the following views are expected to be emphasized:

- The apparent need of better tools for program analysis is an indication of inadequate languages; ultimately the only way to reliable programs is by formal specifications with proper abstractions and by well-structured derivation of programs from them.
- In providing effective support for program derivation it is insufficient to restrict to constructive programming facilities; programs have to be considered as special cases of more general constructions.

In order to cope with different language requirements for program specification, derivation, prototyping, implementation, etc., a wide spectrum language would need a huge arsenal of capabilities. Various current trends in language development have different emphases in this respect, and we do not believe in the creation of languages that are very large along this axis. In spite of its size and ambitious objectives, Ada, for instance, extended the scope of programming languages only modestly towards supporting program development. Integrating effective support for program derivation with all the facilities of an efficient implementation language would necessarily lead to a language with even much greater complexity. To us this seems a hopeless direction, but, deciding from [Ga 89], the idea of such language dinosaurs has not been abandoned.

With a collection of different and more specialized languages the role of conventional high-level languages would change, which would also affect their requirements. Program specification and the initial design transformations could be carried out in languages with only little support for efficient executability, which was the area of the technical contributions in this paper. High-level languages that can be automatically compiled into efficient machine code would be needed as target languages for such design systems, and also as languages for efficiency-oriented transformations. However, the design motivations of current high-level languages have been quite different, and it would be instructive to evaluate them in the light of theses new uses.

Acknowledgments

The joint action approach was developed together with Ralph Back from Abo Akademi. Its use as a basis for a specification language, and the development of associated tools and design methodologies are the topics of project DisCo at Tampere University of Technology. This project is part of the FINSOFT programme of the Technology Development Centre of Finland (TEKES), and is supported by four industrial partners. Fruitful discussions with other project members and with a seminar group are gratefully acknowledged.

References

[AL 88] M. ABADI and L. LAMPORT, The existence of refinement mappings. Res. Rep. 29, Digital Equipment Corporation, Systems Research Center, Aug. 1988.

[AS 85] B. ALPERN and F. B. SCHNEIDER, Defining liveness. *Information Processing Letters 21*, (Oct. 1985), 181—185.

[Ba 78] R. J. R. BACK, On the correctness of refinement steps in program development. Report A—1978—4, Department of Computer Science, University of Helsinki, 1978.

[Ba 80] R. J. R. BACK, Correctness preserving program refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam 1980.

[Ba 88a] R. J. R. BACK, A calculus of refinements for program derivations. *Acta Informatica 25*, 6 (1988), 593—624.

[Ba 88b] R. J. R. BACK, Refining atomicity in parallel algorithms. Report A 57, Department of Computer Science, Åbo Akademi, 1988. To appear in *Conference on Parallel Architectures and Languages Europe*, 1989.

[BK 83] R. J. R. BACK and R. KURKI-SUONIO, Decentralization of process nets with a centralized control. *Distributed Computing 3*, 2 (1989), 73—87. An earlier version in *Proc. 2nd ACM SIGACT—SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug. 1983, 131—142.

[BK 84] R. J. R. BACK and R. KURI-SUONIO, A case study in constructing distributed algorithms: distributed exchange sort. In *Proc. Winter School on Theoretical Computer Science*, Lammi, Finland, Jan. 1984. Finnish Society of Information Processing Science, 1—33.

[BK 88a] R. J. R. BACK and R. KURKI-SUONIO, Serializability in distributed systems with handshaking. In *Proc. ICALP 88, Automata, Languages and Programming* (Ed. T. Lepistö and A. Salomaa), LNCS 317, Springer-Verlag, 1988, 52—66.

[BK 88b] R. J. R. BACK and R. KURKI-SUONIO, Distributed cooperation with action systems. *ACM Trans. Programming Languages and Systems 10*, 4 (Oct. 1988), 513—554.

[BW 89] R. J. R. BACK and J. VON WRIGHT, Duality in specification languages: a lattice-theoretical approach. Report A 77, Department of Computer Science, Åbo Akademi, 1989. To appear in *Mathematics in Program Construction*, LNCS, Springer-Verlag.

[dB 80] J. DE BAKKER, *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[BG 79] R. BALZER and N. GOLDMAN, Principles of good software specification and their implications for specification languages. In *Specification of Reliable Software*, IEEE Computer Society, 1979, 58—67.

[Ba & 89] F. L. BAUER, B. MÖLLER, M. PARTSCH and P. PEPPER, Formal program construction by transformations — computer-aided, intuition-guided programming. *IEEE Trans. on Software Engineering 15*, 2 (Feb. 1989), 165—180.

[BB 87] T. BOLOGNESI and E. BRINKSMA, Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN System 14*, (1987), 25—59.

[BF 88] L. BOUGÉ and N. FRANCEZ, A compositional approach to superimposition. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, San Diego, California, Jan. 1988, 240—249.

[CM 88a] K. M. CHANDY and J. MISRA, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CM 88b] K. M. CHANDY and J. MISRA, Another view of "fairness". *ACM Software Engineering Notes 13*, 3 (July 1988), 20.

[Di 76] E. W. DIJKSTRA, *A Discipline of Programming*. Prentice-Hall, 1976.

[Di 88] E. W. DISJKSTRA, Position paper on "fairness". *ACM Software Engineering Notes 13*, 2 (April 1988), 18—20.

[DS 80] E. W. DIJKSTRA and C. S. SCHOLTEN, Termination detection for diffusing computations. *Information Processing Letters 11*, 1 (Aug. 1980), 1—4.

[EF 82] T. ELRAD and N. FRANCEZ, Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming 2*, 3 (Dec. 1982), 155—173.

[Fe 87] M. S. FEATHER, Language support for the specification and development of composite systems. *ACM Trans. Programming Languages and Systems 9*, 2 (April 1987), 198—234.

[FD 77] C. FORGY and M. C. DERMOT, OPS, a domain independent production system language. In. *Proc. Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., Aug. 1977, Morgan Kaufmann, 1977, 933—939.

[Fr 86] N. FRANCEZ, *Fairness*. Springer-Verlag, 1986.

[Ga 89] R. P. GABRIEL (ED.), Draft report on requirements for a common prototyping system. *ACM Sigplan Notices 24*, 3 (March 1989), 93—165.

[Ha 87] D. HAREL, Statecharts: a visual formalism for complex systems. *Science of Computer Programing 8*, 3 (June 1987), 231—274.

[Jä & 89] H.-M. JÄRVINEN, R. KURKI-SUONIO, M. SAKKINEN and K. SYSTÄ, Object-oriented specification of reactive systems. *Proc. 12th International Conference in Software Engineering*, Nice, France, March 1990, IEEE Computer Society Press, 63-71.

[Ka 87] S. KATZ, A superimposition control construct for distributed systems. Microelectronics and Computer Technology Corporation, Report STP—268—87, Aug. 1987.

[Ku 86] R. KURKI-SUONIO, Towards programming with knowledge expressions. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, Jan. 1986, 140—149.

[Ku 89] R. KURKI-SUONIO, Operational specification with joint actions: serializable databases. To appear in *Distributed Computing*.

[KJ 89] R. KURKI-SUONIO and H—M. JÄRVINEN, Action system approach to the specification and design of distributed systems. In *Proc. 5th International Workshop on Software Specification and Design, ACM Software Engineering Notes 14*, 3 (May 1989), 34—40.

[KK 88] R. KURKI-SUONIO and T. KANKAANPÄÄ, On the design of reactive systems. *BIT 28* ,3 (1988), 581—604.

[LS 84] S. S. Lam and A. U. Shankar, Protocol verification via projections. *IEEE Trans. on Software Engineering SE—10,* 4 (July 1984), 325—342.

[La 89] L. Lamport, A simple approach to specifying concurrent systems. *Comm. ACM 32,* 1 (Jan. 1989), 32—45.

[LF 82] P. E. London and M. S. Feather, Implementing specification freedoms. *Science of Computer Programming.2,* 1982, 91—131.

[MP 83] Z. Manna and A. Pnueli, How to cook a temporal proof system for your pet language. In *Proc. 10th ACM Symposium on Principles of Programming Languages,* Austin, Texas, Jan. 1983, 141—154.

Mo 88a] C. Morgan, Data refinement by miracles. *Information Processing Letters 26,* (Jan. 1988), 243—246.

Mo 88b] C. Morgan, The specification statement. *ACM Trans. Programming Languages and Systems 10,* 3 (July 1988), 403—419.

[Mo 87] J. Morris, A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming 9,* 3 (Dec. 1987), 287—306.

[Ne 87] G. Nelson, A generalization of Dijkstra's calculus. Res. Rep. 16, Digital Equipment Corporation, Systems Research Center, April 1987.

[Pa 85] D. L. Parnas, Software aspects of strategic defense systems. *Comm. ACM 28,* 12 (Dec. 1985), 1326—1335.

[Pn 86] A. Pnueli, Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency* (Ed. J. W. de Bakker, W.-P. de Roever and G. Rozenberg), LNCS 224, Springer-Verlag, 1986, 510—584.

[SL 88] F. B. Schneider and L. Lamport, Another position paper on "fairness". *ACM Software Engineering Notes 13,* 3 (July 1988), 18—19.

[SW 87] B. Shriver and P. Wegner (ed.), *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[WZ 88] P. Wegner and S. B. Zdonik, Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proc. European Conference on Object-Oriented Programming '88,* Springer-Verlag, 1988, 55—77.

Techniques for Modular Language Implementation*

KAI KOSKIMIES

Department of Computer Science, University of Tampere
Box 607, SF—33101 Tampere, Finland
e-mail: koskimie@ondake.uta.fi

Abstract

It is argued that the modularization of language implementation software should be based on the concepts of the source language rather than on certain implementation techniques: this would lead to more maintainable and reusable software components. Various techniques supporting source language oriented modularization are explored, covering both syntactic and semantic issues. For scanning and parsing, a lazy LL(1) method based on independent nonterminal modules is proposed; in this method the scanner and the parser are partially constructed during parsing according to the needs of a particular input. For semantic aspects, an object-oriented approach is suggested in which the source program is viewed as a collection of objects. The classes are derived systematically on the basis of a disciplined syntactic specification of the language.

1. Introduction

A crucial question of any software development is how to divide the software into managable pieces, modules, with simple mutual relationships. The answer can vary considerably, depending on the way a system designer thinks about the system. There are at least two basic approaches. In the implementation-oriented approach the system is viewed as a hierarchy of abstract machines; then the modules provide services required by the abstract machines. In the task-oriented approach the system is divided into pieces according to the logical task of the system, so that different modules implement different subtasks. An important advantage of the latter approach is that if the task is slightly changed, the system can be relatively easily updated by

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

replacing the corresponding modules with new ones. The latter view is normally taken also in object-oriented programming.

As a software product, a language system (analyzer, compiler, translator, interpreter) is perhaps one of the most studied. The structure of such a system has become practically standard, and the components can be usually developed using well-known systematic techniques, often supported by automatic generation tools. The standard structure of a language system follows basically the implementation-oriented approach: typical modules are an input buffer, a scanner, symbol table, code generation services (see e.g. [WeM 80]). These modules can be understood as an abstract implementation machine for a particular language.

It is somewhat surprising that alternative modularization techniques, in particular the task-oriented approach, have not been applied in practical language implementation. The task-oriented approach (which can be called *language-oriented* approach in the context of language systems) has obvious advantages over the implementation-oriented approach:

- the number of modules depends on the size of the language, implying that the sizes of the modules remain small;
- during language development, some part of the language can be easily changed by replacing the module corresponding to that part with another module;
- components of existing language systems can be reused in the development of new languages;
- system maintenance becomes easier because of fine-grained modularization that can be understood on a high conceptual level (i.e., on the level of the source language).

Although the language-oriented modularization principle has not been applied in practical implementations of programming languages (to my knowledge), it is not a completely new idea in the research. From a theoretical point of view, the subject has been studied by Watt [Wat 85]. Some experimental language implementation systems provide a modular specification language (e.g. [Toc 88]). In some implementation systems ([Gro 84], [HeR 75]) a language implementation can be developed in a step-wise way that is ideologically close to the language-oriented modularization.

In this paper we study the language-oriented modularization on the level of a general-purpose modular implementation language (say, Modula-2 or Oberon [Wir 88]). Our results can be applied to writing modular language systems by hand, but they can equally well be used in the design of a generator producing (modular) implementations on the basis of high-level specifications. We feel that even in a system providing a modular specification language the generated code should also be modular: otherwise a small change in some of the specification modules requires a complete recompilation of the generated code (even though the other specification modules perhaps need not be reprocessed).

We proceed as follows. In the next section we introduce a notation for describing the construction of programs; we will use this notation throughout the paper. Section 3 is an informal introduction to a parsing technique supporting modular implementations; this part is essentially a summary of the results given in [Kos 89]. In

Section 4 we present a method for constructing an abstract representation of a program in a modular way. Section 5 discusses briefly the problems in modularizing the dynamic semantics. Sections 4 and 5 are mostly extensions of the ideas presented in [Kos 88]. Finally, in Section 6 we present some concluding remarks.

2. Program generation tools

We make use of static statements enclosed in brackets; these statements can be regarded as advanced "macro" facilities that can be used within normal program text. They are assumed to be executed by a preprocessor (or by the compiler) to produce the actual source code to be inserted in their places. Hence, the information on which the execution depends must be static. We use three kinds of static statements. The ***let*** statement allows the value of a static expression to be inserted in the code:

$$[\textbf{\textit{let}}\ X = E{:}\beta]$$

where E is a (static expression) and β is an arbitrary string. As a result, the string obtained from β by replacing every occurrence of X with the string representation of the value of E is inserted into the source code at this point. A short-hand notation can be used for nested ***let*** statements:

$$[\textbf{\textit{let}}\ X_1 = E_1{:}\,[\textbf{\textit{let}}\ X_2 = E_2{:}\ \dots{:}\ \beta]]$$

can be written in the short form:

$$[\textbf{\textit{let}}\ X_1 = E_1,\ X_2 = E_2,\ \dots{:}\ \beta].$$

Static ***if*** statement is given in the form:

$$[\textbf{\textit{if}}\,E{:}\ \beta]$$

where E is a condition (Boolean expression) and β is an arbitrary string. The condition must be a static expression; if it yields true β is included in the program; otherwise the entire statement is ignored by the compiler. Similarly, a static ***for*** statement

$$[\textbf{\textit{for}}\ X\ \textbf{\textit{in}}\ S{:}\ \beta]$$

denotes a sequence of strings, each obtained from β by replacing the occurrences of X with one element in the ordered set S. The above statement then generates:

$$\beta_1 \beta_2 \dots \beta_k$$

where β_i is obtained from β by replacing every occurrence of X with the ith element of S. We assume that all sets discussed here are ordered; if the order is not explicitly given some arbitrary order is assumed. Static statements may be nested, in which case the outermost statements are executed first. Note that we use italic bold for the keywords of static statements to distinguish them from the keywords of the normal program text.

We use these static statements mainly to express the generation of programs in a compact way: a program containing static statements can be understood as an algorithm for producing a normal program.

3. Scanning and parsing

We define first some basic concepts. A context-free grammar (CFG) is a 4-tuple (V_T, V_N, S, P), where V_T is the set of terminal symbols, V_N is the set of nonterminal symbols, S is the start symbol and P is the set of productions of the form $A \to \beta$, where A is a nonterminal and β is a possibly empty string of terminals and nonterminals. A production of the form $A \to B$, where B is a nonterminal, is called a *chain production.* A CFG is *reduced* if every nonterminal is used in the derivation of a terminal string. A CFG in *non-circular* if there is no nonterminal that can produce a string consisting of this nonterminal only.

Assuming that the modularization is based on language concepts that are (mostly) represented by certain syntactic structures, we decide that for each nonterminal of the language there is a separate module that implements this nonterminal. It might be argued that this decision leads to a huge number of modules in some cases (say, several hundreds), but on the other hand it allows very fine-grained reuse of language structures. We do not regard the possibly great number of modules as a serious problem, assuming that the module library is organized in some sensible way.

The natural way to proceed is then to introduce a handling procedure for each nonterminal module, taking care of the processing of the structure generated by that nonterminal in the well-known recursive descent style. However, the basic modularization principle requires that when writing one module we are not allowed to make use of the detailed knowledge of the tasks of the other modules. This principle guarantees that the modules are interchangeable, as long as the interfaces remain the same. When applied to nonterminal modules, this means that we must be able to replace the implementation part of a nonterminal module into another one without affecting the implementation of the other modules. In language terms, if we change the productions of a nonterminal, it must be sufficient to change the implementation part of that nonterminal only. Note that if the processing procedures are written in the traditional way, this does not hold because the starter and follower symbols of nonterminals are assumed to be known globally, and because all the terminal tokens of the language are assumed to be known by a scanner.

Hence, our problem is the following: how can we write the analysis procedure for one nonterminal module on the basis of the productions of that nonterminal only, without using any knowledge about the productions of other nonterminals and the tokens appearing in them? This implies that the global information about the entire language cannot be embedded statically into the program code, but it must be computed at run-time. The key question is when and how to collect this information. In a nondeterministic top-down analyzer (e.g. [Gro 84], [HeR 75]) the necessary information is essentially recomputed every time it is needed. The other extreme is to compute all the information before the analysis of an input begins. In both cases some loss of efficiency is expected: the former method involves backtracking (which is unpleasent also for the semantic processing), the latter method implies that the analysis time of every program is increased by the time required for parser and scanner construction which is particularly unsatisfactory for small programs of a large language.

Our choice is a method which is between these two extremes. We find out the necessary information about the grammar on the fly during parsing, and store it so that it need not be recomputed when the same parsing situation occurs later. This means that we construct the parser during parsing, but only as far as is needed to analyze a particular input text. This approach can be called *lazy* in the sense that the analyzer is constructed in a lazy manner. The lazy approach has been previously taken in the context of LR parsing by Heering, Klint and Rekers in [HKR 88].

In a traditional recursive descent parser, global grammar information is used only to select the alternative production of a nonterminal, when the procedure of the nonterminal has been activated. Hence, this part of an analyzer procedure must be removed so that the selection can be based on some global data structure that is built during parsing. The analyzer will therefore be partly table-driven (the global data structure for selecting the alternative), partly hard-coded (the code for analyzing the right-hand sides of productions).

Obviously it is possible to give each nonterminal module a procedure that computes the starter symbols of that nonterminal (say A), using the corresponding procedures of those nonterminals appearing on the left-hand sides of the productions of A. Then a straightforward way to construct a lazy recursive descent parser would be to augment each analyzer procedure with an initial action that computes and stores the starter symbols of that nonterminal, together with information that indicates which production must be selected for each starter symbol, if they have not been already computed. By matching the current input symbol with one of the starter symbols the correct alternative can be selected, and the parsing proceeds in the normal way. If none of the starter symbols matches with the input, and there is an alternative that produces the empty string (assuming this can be decided), this alternative can be safely selected. If there is no such alternative, a syntax error must be reported. Obviously this works at least for LL(1) grammars: the fact that the parser makes a "default" move corresponding to the derivation of an empty string does not essentially change the behaviour of the parser. However, this scheme leads to an unnecessarily inefficient parser because the same current input symbol will be matched with a starter symbol many times on different nonterminal levels when the right-hand side of a production begins with another nonterminal. Note that when a starter symbol is matched with the current input in a nonterminal procedure, all the subsequent productions that are applied next to expand the leading nonterminal symbols on the right-hand sides of productions are in fact known in LL parsing. We call these productions the *left-corner productions* of the nonterminal in that context. So, our aim is a global data structure that supplies for each nonterminal not only pairs (a, p) where a is a starter symbol and p is the production to be applied, but sequences of the form $(a, p_1, \ldots, p_k)$, where $p_1, \ldots, p_k$ is the sequence of the left-corner productions of the nonterminal in the parsing situation determined by the starter symbol a. The analysis procedures will then select the alternatives of the nonterminals according to this sequence, without consulting any more the current input symbol. The required data structure will be a labelled directed graph called the *start tree* of the nonterminal.

Suppose that the productions of each nonterminal are numbered $1, \ldots, n$; i.e. the alternative production rules of a nonterminal are given by unique numbers. The leaves of the start tree of A will be the starter tokens of A, and some additional special symbols for handling empty derivations. The essential property of the start tree of A is the following: if there is a leaf labelled t (terminal symbol), then the labels of the arcs on the path from this leaf node to the root give the (numbers of the) left-corner productions when an A produces something that begins with t. In addition, if there is a leaf labelled $\langle A\rangle$, the labels of the arcs on the path from this leaf to the root give the (numbers of the) left-corner productions when an A produces the empty string.

Assuming that we know how to build the start trees we can parse as follows. When the analyzer procedure of the nonterminal A is called, we first check whether the start tree of A is already constructed. If not, we construct it. Then the current input symbol is compared with the leaves of the start tree of A. If it matches with one of the leaves, the production numbers found on the path from the leaf to the root (in the reverse order) are applied in the subsequent activations of analyzer procedures of other nonterminals without consulting the current input symbol, until all these production numbers are consumed. If there is no match, but one of the leaves is $\langle A\rangle$, we know that A produces the empty string and this is the only possible correct choice in this context. Hence we use the production numbers on the path from this leaf to the root as before. If there is no match and no $\langle A\rangle$ leaf, we report a syntax error. When all the production numbers have been consumed, the parser switches its "mode" and starts to process the right-hand side of the last selected rule in the normal way.

Here we will not discuss the construction of the start trees in detail (see [Kos 89]), but instead we show how to write the analyzer procedures. For that purpose we use some notations:

PrimaryStarters(A)	= $\{t$ in $V_T \mid$ there are productions $X_0 \rightarrow X_1 \ldots, \ldots, X_{k-1} \rightarrow X_k \ldots$, where $k > 0$, $X_0 = A$, $X_k = t\}$;
Path(A, x)	= the sequence of numbers associated with the arcs from the (leaf) node x to the root in the start tree of A, in the reverse order;
Variants(A)	= the number of alternative productions for the nonterminal A;
RhsLength(A, i)	= the number of terminal and nonterminal occurrences on the right-hand side of A's production i;
RhsItem(A, i, j)	= the jth terminal or nonterminal occurrence on the right-hand side of A's production i;
Sym(A, i, j)	= the terminal or nonterminal symbol corresponding to RhsItem(A, i, j).

Further, we use the following procedures that are assumed to be provided by a general support module called MLI:

```
procedure Rule(): Integer;
var ProdNumber: Integer;
begin
    ProdNumber:=Head(LeftCorners);
    LeftCorners:=Tail(LeftCorners);
    if LeftCorners is empty then Mode:=Examine; end;
    return ProdNumber;
end Rule;
```

```
procedure Scan(t: Token);
begin
    if t is in the current input position then
        advance the input pointer past token t;
    else SyntaxError;
    end;

end Start;
```

For each nonterminal A we construct procedure Create as follows (the choice of the name will become understandable later), to be included in the module of the nonterminal:

```
procedure Create;
begin
    if MLI.Mode=MLI.Examine then
        if there is no start tree for A then construct the start tree of A end;
        if there is terminal t such that
            a) t is in the current input position, and
            b) t is a leaf node of the start tree of A
        then
            if t belongs to Primary Starters(A) then
                advance the input position past t;
            end;

            MLI.LeftCorners:=Path(A, t);
        else
            if there is ⟨A⟩ in the leaves of the start tree of A then
                MLI.LeftCorners:=Path(A, ⟨A⟩)
            else
                MLI.SyntaxError;
            end;

        end;
          MLI.Mode:=MLI.Parsing;
    end;
    case MLI.Rule() of
        [for i in 1..Variants(A):
        i: [for j in 1..RhsLength(A, i):
            [if RhsItem(A, i, j) is terminal and j>1:
                [let S=Sym(A, i, j): MLI.Scan(S);]]
            [if RhsItem(A, i, j) is nonterminal:
                [let S=Sym(A, i, j): S.Create;]]]]
    end;

end Create;
```

Here LeftCorners and Mode are global variables provided by the general support module MLI, initially Mode = Examine. Note that we pay no attention to error recovery. Traditional error recovery techniques are in general not applicable, because there is no global grammar information that could be used e.g. to skip tokens after an error.

A modular recursive descent parser has some interesting properties. The fact that we make a default move leading to empty derivation implies that even some non-LL(1) grammars can be parsed succesfully. For example, the classical dangling else problem is solved simply by parsing according to the productions

IfStatement→"if" Expr "then" Statement ElsePart
ElsePart→"else" Statement |

which makes the grammar ambiguous. In this case the parser will always try to recognize a non-empty else part for the innermost preceding "if" instead of an empty one, if possible.

Another interesting feature is due to the fact that there is no global scanner, but the scanner is distributed in the start trees. This leads to "syntax-directed" scanning: only those tokens are considered in the scanner that are possible in the syntactic context. For example, consider the well-known Pascal subrange problem: the scanning of a subrange definition, say "1...10", fails because the principle of maximal length forces the scanner to expext a real constant after reading "1.". The problem does not appear in our method because a real constant cannot start a subrange and will not be considered at all.

Our method introduces also some new problems. Note that in principle the LL(1)-ness of the grammar is never known in advance, when the parsing begins (indeed, as shown above, the grammar need not be LL(1) in some cases.) Since only those parts of the grammar are examined that are actually used in analyzing a particular input, the LL(1)-ness cannot be decided at all. The method guarantees a correct parse for all LL(1) grammars, and the parser cannot accept an invalid input for any grammar, but it can a) produce a correct parse for some non-LL(1) grammars and b) report a syntactic error for a correct input of some non-LL(1) grammars. Problem b) is of course unpleasent: it would be more appropriate to report a grammar error than a syntactic error. Although most of the non-LL(1) cases must be eliminated during parsing in order to construct the start trees in a sensible way, some cases remain undetected. For a discussion, see [Kos89].

The syntax-directed scanning scheme implies certain problems, too. Because the scanner is distributed in the start trees, there may be conflicts between the tokens that are not known by the scanning process. For example, it is in general impossible to prevent a keyword belonging to one part of the grammar to be interpreted as an identifier when processing another part. Our method supports the convention that keywords are not reserved symbols but can be used e.g. as identifiers, as long as the left context determines uniquely the identity of the token.

The method described above has been implemented and some preliminary experiments have been carried out [Kos89]. The results show — somewhat surprisingly — that a modular scanner/parser is as fast as a traditional recursive descent one, reaching the speed of 300 000 tokens/min. It turns out that in practice the construction of start trees takes very little time: only for very small programs a difference in the running time was observed, when compared to a traditional recursive descent parser. The start trees tend to be rather small: for a subset of Pascal the average depth of the start trees was less than 2, and the average number of leaves was 2.7.

It is interesting to note that the behaviour of the modular parser is sensitive to the properties of the grammar, and even to the properties of a particular input. If the grammar is "modular" in the sense that it consists of several relatively independent subgrammars, and only one or some of them are typically used in one input text, the start trees need to be constructed only for a small part of the grammar.

4. Construction of an abstract representation

We consider a program as a set of interrelated objects that, when put into a particular environment, behaves in a certain way implied by the language semantics. Consequently, there are two kinds of concepts involved in language implementation: *program concepts* that have more or less obvious concrete counterparts in the syntax of the language, and *environmental concepts* that are not represented in the program text, but belong to the "abstract machine" that executes the program. Our intention is to view both kinds of concepts in the object-oriented setting; correspondingly, instead of concepts we will speak of *program classes* and *environmental classes*. The program classes will be implemented by regarding nonterminals as classes, and by adding certain parts into the nonterminal modules constructed in Section 3. We shall use the term *nonterminal class* as a synonym for program class. The environmental classes could be provided by some general implementation support module (like MLI, see Section 3), or they could be implemented by additional modules; we use the latter approach in the sequel. The connection between these two class categories is established by the fact that some nonterminal classes are considered as subclasses of the environmental classes.

Let us first consider the problem of constructing an abstract representation of a program. To establish a sensible class hierarchy for the classes represented by nonterminal symbols we require that the syntactic specification is given in a certain form.

A context-free grammar is *structured*[1] if for each nonterminal A, either

(i) there is only one production that has A on the left-hand side, or
(ii) all the productions that have A on the left-hand side are chain productions;

but not both. Further, we say that a grammar is *well-structured*, if it has the following properties:

(i) it is structured;
(ii) it is reduced and non-circular;
(iii) there is no nonterminal A such that the only production having A on the left-hand side is a chain production;
(iv) each nonterminal appears on the right-hand side of a chain production at most once.

The basic idea is to interpret chain productions as presentations of class hierarchies. This is a natural interpretation: the fact that a nonterminal A has the productions $A \rightarrow B_1, \ldots, A \rightarrow B_k$ is just another way of saying that a B_1 is an $A, \ldots,$

[1] This grammar form has been used (independently) by Jürgen Uhl [Uhl 86]. However, he used this form for establishing equivalence relations between nonterminals rather than class hierarchies. We adopt his term ("strukturierte Grammatik").

a B_k is an A. A production that is not a chain production expresses only the constituent parts of a concept that is "basic" in the sense that it does not have subclasses, whereas a chain production $A \rightarrow B$ expresses the relation "B is a subclass of A".

We say that the nonterminals having only chain productions are ***superclass nonterminals***, and the nonterminals having no chain productions are ***basic nonterminals***. The properties listed above for well-structuredness guarantee that

(a) each nonterminal is either a superclass nonterminal or a basic nonterminal, but not both;
(b) there are no needless or circular classes;
(c) there are no identical classes;
(d) the class structure is purely hierarchical (i.e. there is no multi-inheritance).

The properties (a), (b), (c) ,and (d) are implied by (i), (ii), (iii), and (iv), respectively. In the following we assume that a grammar is well-structured. Note that the well-structuredness of a grammar is easy to check using well-known techniques, and that an arbitrary context-free grammar can be automatically transformed into a well-structured one in a straightforward way, without affecting the essential properties of the grammar (like the generated language or the parsing properties); this requires only the introduction of some new nonterminals and possibly some renaming of the nonterminals. Also note that although class nonterminals cannot be circular they can be (and often are) recursive: there is no reason why a class nonterminal could not appear on the right-hand side of a production of one of its subclass basic nonterminals. The reader is invited to confirm that no nonterminal can appear both on the left-hand side and on right-hand side of some production (i.e. there are no directly recursive nonterminals).

Basically, the existance of an instance of a basic nonterminal in a syntax tree implies the existance of an object of the class corresponding to the nonterminal. In contrast, an instance of a superclass nonterminal merely establishes a new class level for an object that corresponds to the basic nonterminal instance somewhere below the superclass nonterminal.

To express classes in a program, we assume an Oberon-like [Wir88] type extension facility[2]: a record type may be extended with additional fields to create a new record type (subclass) that is upwards compatible with the original record type (superclass). Type extension is given as

type T = **record** (U)... fields...**end**;

where U is the superclass type that is extended with the new fields, yielding the subclass type T. As in Oberon, if a record type is given in the definition part of a module, it can be extended in the implementation part; this is only a means to introduce "invisible" fields for a visible record type. This minor feature turns out to be very useful in our method.

Consider the nonterminal modules constructed following the method described in the previous chapter. For each nonterminal module we specify a record type that provides all the local data for objects of the nonterminal class; we call this the *instance*

[2] The new object-oriented extension of Oberon [MTG 89] might have been even more suitable for our purposes; but we stick to a presentation language that is close to Oberon because we assume it is widely known.

type of the nonterminal. If the nonterminal has a superclass, this type is defined as an extension of the corresponding type in the module of the (*immediate*) superclass nonterminal. We will modify the "Create" procedure introduced in Section 2 so that it will return as its value a reference to the instance object. In the case of a superclass nonterminal the reference to be returned is provided directly by a call of a "Create" procedure of a subclass nonterminal; in the case of a basic nonterminal the instance object is explicitly created.

In an abstract representation of a program, certain fields of the instance type refer to objects whose (instance) type is provided by the modules of the constituent nonterminals. These fields can be declared only in the body of the module: the fact that nonterminals may be recursive prohibits the declaration of these fields in the definition part (otherwise there would be circular importing between the definition parts). This is natural also because these fields are internal knowledge of the objects that should be used only by the methods of the corresponding class (i.e. by the procedures of the module). However, the instance type itself must be declared in the definition part of the module, because it is needed by other modules. This contradiction can be nicely solved using the Oberon-like feature which allows the adding of new "invisible" fields in the module body into a record given in the definition part.

In addition to the special notations introduced in Section 3, we use the following notation:

Super(A) denotes the immediate superclass nonterminal of A, if it exists; otherwise A;

In the following we give a scheme for generating a nonterminal module together with the parts that are needed for constructing an abstract representation of the program.

```
definition NontName;
    import MLI
        [if NontName has a superclass:, [let S=Super(NontName): S]];
    type Class=pointer to InstanceType;
    type Instance Type=record
        [if NontName has a superclass:
            ([let S=Super(NontName): S.InstanceType])]
    end;

    var Descriptor: MLI.DescriptorType;
    procedure Prepare;
    procedure Create(): Class;
end NontName;

module NontName;
    import MLI
    [for N in {A | there is a production NontName→ ... A ...}: ,N];
    [if NontName is a basic nonterminal:
    type InstanceType=record
        [for j in 1..RhsLength(NontName, 1):
            [if RhsItem(NontName, 1, j) is nonterminal:
                [let Y=Sym(NontName, 1,j), Z=j:
                comp_Z_Y: Y. Class;]]]
        end;]

        procedure Prepare;
            ... for constructing the start tree, see Section 3 ...
        end Prepare;
        procedure Create(): Class;
            var NewObj: Class;
        begin
            [if NontName is a superclass nonterminal:
            if MLI.Mode = MLI.Examine then
                ... as in Section 3 ...
            end;

            case MLI.Rule() of
                [for i in 1..Variants(NontName):
                    [let X = Sym(NontName,i, 1):
                        i: return X.Create();]]
            end;]

            [if NontName is a basic nonterminal:
                New(NewObj);
                [for j in 1..RhsLength(NontName,1):
                    [if RhsItem(NontName,1,j) is terminal:
                        [let t = Sym(NontName,1,j): MLI.Scan(t);]]
                    [if RhsItem(NontName,1,j) is nonterminal:
                        [let Y = Sym(NontName,1,j), Z = j:
                            NewObj^.comp_Z_Y:= Y.Create();]]]
                return NewObj;]
        end Create;
    end NontName;
```

Note that we have slightly modified the parsing scheme presented in Section 3 to make use of the well-structuredness of the grammar. Since a basic nonterminal has only one syntactic alternative, there is no need for a **case** statement and for the preceding **if** statement in the Create procedure. Hence these statements can be omitted, provided that the arcs corresponding to the productions of basic nonterminals are removed from the start trees as well.

The above scheme produces a structure which is exactly the abstract syntax tree of the program. However, we are aiming at a more elaborated structure that would be more amenable to further processing. For this purpose we need new environmental classes.

As an example, suppose that we have an environmental class providing the abstract concept of a general list. To be able to conveniently specify the sequential execution of a statement list we would like to represent a statement list as a list rather than as a tree structure. Hence, we say that the nonterminal class "StatementList" is a subclass of the environmental list class. Consequently, the nonterminal class that gives the element of the list ("Statement") must be a subclass of another environmental class that gives the abstract concept of a list element. Since these environmental classes are obviously closely related, they are provided by the same module:

```
definition List;
    type List=record end;      (* only invisible fields *)
    type Elem=record end;
    type ListClass=pointer to List;
    type ElemClass=pointer to Elem;
    procedure CreateList(): ListClass;
    procedure Insert(L: ListClass; E: ElemClass);
    ... other procedures ...
end List;
```

We make use of these classes in the instance types of StatementList,

```
type InstanceType=record (List.List) end;
```

and Statement,

```
type InstanceType=record (List.Elem) end;
```

Note that in principle the environmental superclasses are treated in the same way as nonterminal superclasses. However, in StatementList there are no (invisible) fields of the instance type that would contribute to the abstract representation; the structure of a statement list is implicitly accessible through the operations provided by the list module. An element of a list (Statement) is created normally using New in the creation operation of the basic nonterminal (e.g. IfStatement), and then inserted into the list using the appropriate list operation. In contrast, a list (StatementList) must be created using directly the creation operation provided by module List because this requires certain initializing actions that cannot be given in the nonterminal module.

Note that the class hierarchy must be consistent in the sense that all the instances of a nonterminal class X have the same class levels, independently of the context. The class levels of the objects do not depend on the syntactic context, but only on the existance of certain chain productions. Hence, even though X is not produced by its superclass nonterminal Y in a particular context, the object created for the instance of X has a Y-level. This holds for environmental superclasses as well: for example, a statement has to be a list element in every context, even though it is (syntactically) not an element of a statement list.

Since we regard a list element as a superclass of a statement, this must be true for every instance of a statement: the class hierarchy must be consistent in this sense. Hence a statement should always appear in a list of statements.

Let us consider a more complicated example, the implementation of name environments (i.e. symbol tables). Again we may assume the existance of an additional module providing certain environmental superclasses. For example, we could have:

```
definition NameEnv;
    import ... ;
    type Decl=record name: String; end;
    type Region=record ... end;
```

```
        type DeclClass=pointer to Decl;
        type RegionClass=pointer to Region;
        procedure CreateRegion(): RegionClass;
        procedure DeleteRegion(X: RegionClass);
    ...
    end NameEnv;
```

Suppose that we have the grammar fragment

```
Declaration = VariableDeclaration|TypeDeclaration|...
VariableDeclaration = "var"...
TypeDeclaration = "type"...
```

Nonterminal Declaration (or its instance type) should then be a subclass of Decl, and the nonterminals generating visibility regions like modules or blocks should be subclasses of Region; VariableDeclaration and TypeDeclaration are subclasses of Declaration as usual. The creation operation of VariableDeclaration (a basic nonterminal) creates a new object in the normal way, and then inserts it to the region using an appropriate operation provided by NameEnv. The creation operation of a region nonterminal (say, Block) also creates the region object using New, but it must also apply other operations provided by NameEnv to "enter" and "exit" the region.

It should be noted that above we have only sketched the basic guidelines that could be followed in the implementation. The details depend on the source language, and it is possible that even the basic principles may have to be adjusted to fit a particular language.

5. Semantics

The (dynamic) semantics of a language is essentially more irregular than the parts discussed previously. Hence it is difficult to develop techniques that would be generally applicable. The basic principle, however, should be that the dynamic semantics of the instances of nonterminal classes should be based on the methods of the classes. We illustrate this by an example.

Consider the following fragment of a language:

```
Statement→AssStatement|IfStatement|...
AssStatement→VariableDenotation :=" Expression
IfStatement→"if" Expression "then" Statement
```

Here Statement is a superclass nonterminal, while AssStatement and IfStatement are basic nonterminals. Each statement has the property that it can be executed; hence a semantic field of the instance type of Statement provides a procedure (method) for executing a statement object.

```
definition Statement;
    import MLI;
    type Class=pointer of InstanceType;
    type StatEx=procedure(X: Class);
    type InstanceType=record
        execute: StatEx;
    end;
    ...
end Statement;
```

Module Statement does not provide any value for field "execute"; using the object-oriented terminology this is a virtual method of the class Statement. The value of "execute" is given at the lower level where the kind of the statement is known:

```
definition IfStatement;
    import MLI, Statement;
    type InstanceType=record (Statement.InstanceType) end;
    type Class=pointer to InstanceType;
    procedure Create(): Class;
    ...
end IfStatement;
module IfStatement;
    import MLI, Statement, Expression;
    type InstanceType=record
        comp_1_Expression: Expression.Class;
        comp_2_Statement: Statement.Class;
    end;

    procedure ExecuteIf(S: Statement.Class);
    begin
        with S: Class do
            if S^.comp_1_Expression^.evaluate()=1 (* true *)
            then S^.comp_2_Statement^.execute
        end
    end ExecuteIf;
    procedure Create(): Class;
        var NewObj: Class;
    begin
        New(NewObj);
        NewObj^.execute:=ExecuteIf; (* determine the execution method *)
        Scan("if");
        NewObj^.comp_1_Expression:=Expression.Create();
        Scan("then");
        NewObj^.comp_2_Statement:=Statement.Create();
        return NewObj;
    end Create;
    ...
end IfStatement;
```

In this way every creation of a statement instance, carried out by the basic statement nonterminals like IfStatement, must assign an appropriate value for the execution operation. Hence, when the execute-field of a statement object is called, the actual routine will depend on the kind of the statement. We have followed here the Oberon conventions which require that the actual procedure has the same parameter types as the virtual one; therefore IfStatement's parameter has to be of type Statement. Class (and not Class which would be more natural). Explicit subclass checking (**with** statement) guarantees that the parameter statement is really an **if** statement.

6. Discussion

The starting point of this work has been the observation that so far the modularization of language implementation software has been based on very implementation-oriented thinking. Implementation aspects have always had deep effect on the way we design and view programming languages. We argue that the conventional modularization technique which treats the source language as a black box has led to the view that languages are in principle indivisable, and that it is not sensible to try to reuse parts of existing language implementation software in the development of other languages. It is characteristic that programming languages are often regarded as a means to communicate with a computer, as a "formal language", suggesting a close relationship with natural languages. However, programming languages are not like natural languages: they are most of all technical tools to build systems. Like other complex industrial tools they should be composed of relatively specialized parts that can nevertheless be used as such in many kinds of system building tools. This would give us the same benefits that are now regarded as self-evident in other engineering branches: new production (i.e. programming) systems could be rapidly developed for different purposes using existing building blocks, old systems could be modernized by replacing certain parts with more advanced parts, and system maintenance would be easy because the system consists of small modules with clean interfaces.

References

[Gro 84] Grossmann R., Hutschenreiter J., Lampe J., Lötzsch J., Mager K.: Depot 2a — Metasystem für die Analyse und Verarbeitung Verbundener Fachsprachen. Anwenderhandbuch, Sektion Mathematik, Technische Universität Dresden, 1984.

[HKR 88] Heering J., Klint P., Rekers J. G.: Incremental Generation of Parsers. Report CS—R8822, Centre for Mathematics and Computer Science (CWI), Amsterdam 1988 (also in the Proc. of Sigplan '89 Symposium on Design and Implementation of Programming Languages).

[HeR 75] Heindel L. E., Roberto J. T.: Lang-Pak — An Interactive Language Design System. Elsevier, New York—London—Amsterdam 1975.

[Kos 88] Koskimies K.: Software Engineering Aspects in Language Implementation. In: Proc. of Workshop on Compiler-Compiler and High-Speed Compilation, Berlin, Oct. 1988.

[Kos 89] Koskimies K.: Lazy Recursive Descent Parsing for Modular Language Implementation. Arbeitspapiere der GMD 376, Gesellschaft für Mathematik und Datenverarbeitung, Forschungsstelle Karlsruhe, April 1989.

[MTG 89] Mössenböck H., Templ J., Griesemer R.: Object Oberon — An Object-Oriented Extension of Oberon. ETH Zürich, Institut für Computersysteme, Report 109 (June 1989).

[Toc 88] Toczki J., Gyimothy T., Horvath T., Kocsis F.: Generating Modular Compilers in PROF-LP. In: Proc. of Workshop on Compiler-Compiler and High-Speed Compilation, Berlin, Oct. 1988.

[Uhl 86] Uhl J.: Spezifikation von Programmiersprachen und Übersetzern. GMD-Bericht Nr. 161, Gesellschaft für Matematik und Datenverarbeitung, 1986.

[Wat 85] Watt D. A.: Modular Description of Programming Languages. Report A-81-734, Computer Science Division — EECS, University of California, Berkeley 1985.

(WeM 80] Welsh J., McKeag M.: Structured System Programming. Prentice-Hall 1980.

(Wir 88] Wirth N.: The Programming Language Oberon. Software Practice and Experience 18 (7). 671—690 (July 1988).

An Error-Recovering Form of DCGs*

JUKKA PAAKKI

Nokia Research Center
P.O.Box 156, 02101 Espoo, Finland

KARI TOPPOLA

Department of Computer Science, University of Helsinki
Teollisuuskatu 23, 00510 Helsinki, Finland

Abstract

In this paper an alternative implementation of Prolog's Definite Clause Grammars (DCGs) is presented. The DCG variant is based on the context-free grammar class LL(1) and it solves some of the problems with parsing programming languages using conventional DCGs, such as nondeterminism and intolerance to syntax errors.

1. DCGs and Context-Free Grammars

The programming language Prolog has been connected to parsing right from its very birth: the first real implementation of the logic programming idea [Col 73] was actually developed for processing (i.e. parsing) *natural languages.* Since then, several special notations especially for parsing have been introduced in Prolog, the most popular one being the Definite Clause Grammars (DCGs) [PeW80]. DCGs can be considered as an executable form of context-free grammars that have traditionally been the leading notation in specifying the syntax of *programming languages.*

Informally, a context-free grammar consists of a finite set of *nonterminal symbols,* a finite set of *terminal symbols,* and a finite set of *productions* of the form

$$A \dashrightarrow S_1, S_2, \ldots, S_n \, (n >= 0)$$

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

where A is a nonterminal symbol, and each S_i is either a nonterminal or a terminal symbol. A context-free grammar represents all the syntactically legal sentences (programs) of the language. A sentence can be derived from the grammar by beginning with a symbol string consisting of the designated *start symbol* and by repeatedly replacing a nonterminal in the symbol string with the right-hand side of a production for that nonterminal, until the string contains only terminal symbols; that terminal string is a sentence of the language. The language defined by the context-free grammar consists of exactly those sentences that can be derived from the start symbol.

As an example, simple arithmetic expressions can be defined with the following context-free grammar where the set of nonterminal symbols is {expr, term, factor, number}, the set of terminal symbols is {“+”, “*”, “(“ , ”)”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}, and the start symbol is expr:

```
expr    ---> expr, “+”, term
expr    ---> term
term    ---> term, “*”, factor
term    ---> factor
factor  ---> “(“,expr,”)”
factor  ---> number
number  ---> “0”
number  ---> “1”
number  ---> “2”
number  ---> “3”
number  ---> “4”
number  ---> “5”
number  ---> “6”
number  ---> “7”
number  ---> “8”
number  ---> “9”
```

DCGs, as a notation, resemble much context-free grammars. In a DCG, nonterminal symbols are represented by Prolog terms and terminal symbols by Prolog lists. For example, the context-free grammar given above can be modified into Quintus Prolog [Qui 86] simply by terminating each production with a period.

2. DCGs and Language Processing

The DCG facility is in most Prolog dialects implemented with a transformation from DCG into ordinary Prolog. The transformation is straightforward: each nonterminal is translated into a predicate with two extra arguments (representing the input symbol list before and after processing the corresponding nonterminal), and each terminal is translated into a call for a special built-in predicate (corresponding to advancing the input pointer to the next input symbol).

As an example, the DCG for simple arithmetic expressions outlined in chapter 1 would be translated into the following Prolog program (for clarity, we present the terminals explicitly, instead of using their ASCII codes):

```
expr(S0, S)    :- expr(S0, S1),shift(S1, '+', S2),term(S2, S).
expr(S0, S)    :- term(S0, S).
term(S0, S)    :- term(S0, S1), shift(S1, '*', S2), factor(S2, S).
term(S0, S)    :- factor(S0, S).
factor(S0, S)  :- shift(S0, '(', S1), expr(S1, S2), shift(S2, ')', S).
factor(S0, S)  :- number(S0, S).
number(S0, S):- shift(S0, '0', S).
number(S0, S):- shift(S0, '1', S).
number(S0, S):- shift(S0, '2', S).
number(S0, S):- shift(S0, '3', S).
number(S0, S):- shift(S0, '4', S).
number(S0, S):- shift(S0, '5', S).
number(S0, S):- shift(S0, '6', S).
number(S0, S):- shift(S0, '7', S).
number(S0, S):- shift(S0, '8', S).
number(S0, S):- shift(S0, '9', S).
```

Here shift is the built-in scanning predicate:

$$\text{shift}([X|S], X, S).$$

It can be interpreted as "removing symbol X from input stream $[X|S]$, producing stream S".

Sentences of a language are recognized by a parsing process. Most parsing strategies lay some restrictions on the underlying context-free grammar of the language: for instance ambiguous grammars are usually forbidden. Conventionally a DCG is applied, i.e. the input program is "parsed", by executing the corresponding ordinary Prolog program. The operational semantics of Prolog thus implies that a DCG implemented this way produces a top-down, left-to-right, recursive descent, backtracking parser. This characterization in terms of normal Prolog brings out some problems with DCGs when considering practical parsing of programming languages:

(1) the order of alternative productions for a nonterminal has great significance on the speed of the parser (parsing is nondeterministic),
(2) left-recursive grammars cannot be handled,
(3) no recognition or recovery of syntax errors is provided, and
(4) lexical analysis cannot be interleaved with parsing (since the source program is represented as a list of symbols); this leads to two passes over the source program for parsing it.

On the other hand, reducing DGCs into ordinary Prolog makes them more general than context-free grammars:

(i) grammar symbols can have an arbitrary number of arguments, and
(ii) procedure calls can be embedded within productions.

These additional features make DCGs closely related with attribute grammars [Knu 68]: arguments can be considered as "attributes" and procedure calls as "semantic rules".

As an example, our DCG for arithmetic expressions can be revised in such a way that the value of an expression is evaluated during parsing. Note that the original version is left-recursive; we have to remove left-recursion and for instance replace it with right-recursion in order to make the DCG correctly executable. The arguments represent the values of the subexpressions, and procedure calls are enclosed in {...}.

```
expr(Val)    ---> term(V1), "+", expr(V2), {Val is V1+V2}.
expr(Val)    ---> term(Val).
term(Val)    ---> factor(V1), "*", term(V2), {Val is V1*V2}.
term(Val)    ---> factor(Val).
factor(Val)  ---> "(", expr(Val),")".
factor(Val)  ---> number(Val).
number(0)    ---> "0".
number(1)    ---> "1".
number(2)    ---> "2".
number(3)    ---> "3".
number(4)    ---> "4".
number(5)    ---> "5".
number(6)    ---> "6".
number(7)    ---> "7".
number(8)    ---> "8".
number(9)    ---> "9".
```

3. A More Practical Form of DCGs

We have implemented the DCG formalism in a way that is more related to the parsing theory of context-free grammars. Most notably, we have tried to remove the shortcomings (1)—(3) of the conventional DCG implementation strategy discussed in the previous chapter. The initial idea was to support primarily syntax error handling, but the resulting system was expected to contribute to other parsing aspects as well, such as efficiency. In the sequel we shall briefly present the main characteristics of the system.

Determinism

Since the normal execution model in Prolog is a complete depth-first traversal of the search tree, it was a natural choice to retain the top-down parsing strategy in our DCG facility as well. However, the general backtracking mechanism of Prolog contradicts the standard parsing principles in language processing: conventional DCGs parse the input program nondeterministically, while traditionally deterministic parsing is preferred. Nondeterministic parsing also torpedos syntax error handling since it makes hard to connect a recognized error to the erroneous grammar symbol. Moreover, nondeterministic parsing (although being a more general approach than deterministic one) is rarely actually needed in the context of programming languages because most programming languages are designed to be deterministically parsable.

Because of these reasons, we have based our DCG implementation on the context-free grammar class LL(*1*), i.e. parsing is a top-down left-to-right process using a lookahead of length *1*. This choice makes our notation more restricted than the conventional one; the resulting formalism is rather related to one-pass attribute grammars or affix grammars [Kos 71] than to general attribute grammars.

Left recursion

Since our system can only process LL(*1*) grammars, left recursion is still forbidden. However, the system provides some relief in this restriction by automatically eliminating left-recursion from the original grammar, when asked. It also provides two other grammar transformations: left factoring, and elimination of useless productions. All these transformations have been implemented according to [ASU 86].

One shortness in these grammar transformations is that they are applied merely to the context-free part of the DCG; if the original grammar makes use of symbol arguments or procedure calls, these have to be updated on the transformed grammar by the user. The reason for excluding the semantic aspects from the DCG transformations is that a well-known result with attribute grammars shows that in general it is impossible to transform even an L-attributed grammar into an equivalent LL-attributed form [GiW 78] (preserving the level of semantic information during the transformation); thus an automatic semantic conversion would be doomed to failure.

Error recovery

Because we have based our implementation on deterministic parsing, we can employ standard syntax error handling techniques instead of just giving up, as is the case with the conventional DCG implementation. Our error recovery method is a combination of *panic mode* and *phrase-level* methods, as described in [WeM 80].

The idea is to always keep the parser in synchron with the input stream. This means that when detecting an error, the parser skips symbols in the input, until a symbol is found that matches the current state of the parser. The parser and the input are synchronized both at entry and at exit of each nonterminal under parse. Synchronization is based on the FIRST and FOLLOW sets of nonterminals (see e.g. [ASU 86]).

The principle of error handling can be illustrated by giving as an example a procedure for parsing nonterminal A with production $A \rightarrow B$:

```
procedure A (Followers);
begin
    if not (Next in FIRST(A)) then begin
     Error ...;
     Skipto(FIRST(A)+Followers);
    end;
    if Next in FIRST(A) then begin
        B; — parse the right-hand side
        if not (Next in Followers) then begin
         Error ...;
         Skipto(Followers);
        end
    end
end,
```

Here the set Followers includes all the symbols in
FOLLOW(A)+(FOLLOW(X_1)+FOLLOW(X_2)+...+FOLLOW(X_n)), $n>=0$,
where the symbols X_i represent the nonterminals on the path from A to the root in the underlying parse tree, i.e. all the nonterminals which have been entered but not yet exited. The FOLLOW(X_i) sets guarantee that within any underlying parse tree a lower-level nonterminal cannot inadvertently skip over a token which a higher-level nonterminal expects to deal with.

Next represents the current input token, Error emits an appropriate error message, and Skipto(S) skips the input stream until a token in set S is found.

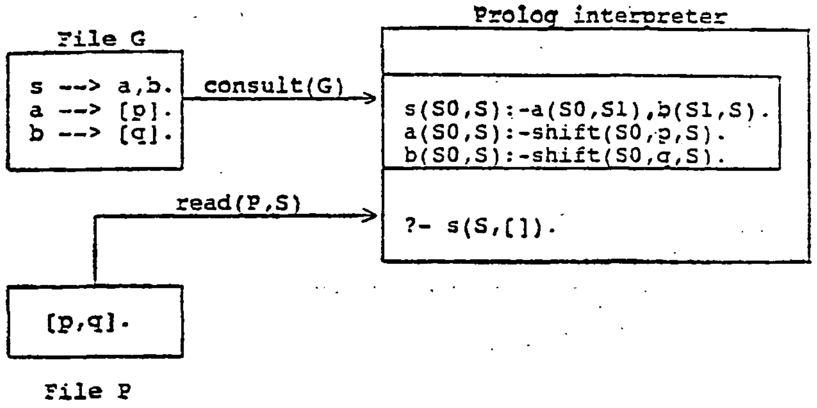
Because of the interactive nature of working with a Prolog interpreter, we have enriched this automatic form of recovery with the possibility for *local correction*: if requested, the parser always halts when detecting an error and asks the user to correct the current erroneous token. The available operations are replacement, insertion, and deletion.

We demonstrate the system by giving in the Appendix an example session.

4. Implementation

Our deterministic error-recovering DCG notation has been implemented using a meta-interpreter (see e.g. [StS 86]) that "interprets" the input grammar. Thus the solution is different from the conventional implementation where a DCG is first translated into ordinary Prolog and after that executed by a standard Prolog interpreter or compiler. The difference can be characterized more explicitly by sketching in Figures 1 and 2 the conventional implementation strategy and the metainterpreter strategy, respectively.

In our implementation the grammar is transformed into an internal representation of the DCG interpreter. This interpreter (a Prolog program) parses the source program by recursively applying a universal parser predicate with the current grammar symbol as parameter. The interpretation follows the principles discussed in chapter 3.



```
s --> a,b.
a --> [p].
b --> [q].
```

```
s(S0,S):-a(S0,S1),b(S1,S).
a(S0,S):-shift(S0,p,S).
b(S0,S):-shift(S0,q,S).
```

```
?- s(S,[]).
```

```
[p,q].
```

Figure 1. Conventional implementation of DCGs

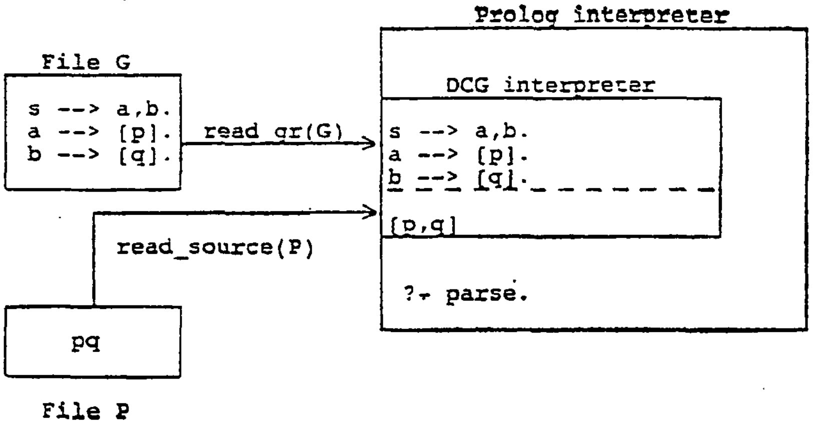


Figure 2. Meta-interpreter implementation of DCGs

In order to support lexical analysis, the system includes a standard scanner (read_source) that can be used for reading the source program and for converting it into a list of tokens. The lexical analyzer makes the conversion assuming "normal" patterns of "ordinary" token classes, such as identifiers, numbers, and operators. In case the lexical form of the source language does not match the assumptions made by the system, the user must either modify the standard analyzer or supply an analyzer of her/his own.

The system is embedded in Quintus Prolog [Qui 86], and it is described in more detail in [Top 89].

5. Experiences

Our DCG variant has been applied to several toy examples, such as arithmetic expressions. In these simple cases the system is superior to the conventional implementation: all the syntactic errors can be uncovered quite rapidly and even corrected on-the-fly. The automatic transformations free the user to some extent from artificial grammar constructions, such as right recursion.

Since the design of the system stems from practical problems with using Prolog for parsing, we have tested it in a more realistic case as well. The syntax of the programming language Edison [Bri 82] was specified as a DCG which was then executed both using our system and using Quintus Prolog. The efficiency of these parsers was analyzed and the results are given in Tables 1 and 2. The length of the source programs is indicated by lines, our system by Meta-DCG, and Quintus Prolog by Quintus-DCG. For Quintus Prolog we have assigned two figures, the first one being for the compiled parser and the second one for the interpreted parser. All the figures for our system are for the compiled parser. The tests have been carried out in a VAX/8800 under VMS.

Table. 1. Execution time of DCGs (seconds of cpu time)

lines	Quintus-DCG	Meta-DCG
10	0.01/0.1	7.8
60	0.06/0.7	88.4
100	0.08/0.8	101.8

Table 2. Memory consumption of DCGs (kbytes)

lines	Quintus-DCG	Meta-DCG
10	594/660	1368
60	654/899	4742
100	654/899	4875

As can be noticed, the meta-interpreter implementation unfortunately resulted in drastic loss of efficiency when compared to the conventional implementation by translation into ordinary Prolog. Even for relatively small Edison programs (less than 100 lines) the meta-interpreter was far too slow for practical consideration, and for source programs larger than 100 lines the Quintus Prolog system might run out of memory. Also parser initialization (loading the meta-interpreter, reading the DCG, checking the LL(*1*) property) took clearly more time than in the conventional case (reading the DCG, converting it into Quintus Prolog).

The main reason to this unfortunate inefficiency lies certainly in meta-interpretation. On one hand the program is quite complex and on the other hand the DCG is represented as data; thus no optimizations on the grammar can be done by the Prolog system as is the case with the conventional implementation. One part of the difference can be explained by the fact that our system has to check for syntactic correctness of the source program which task is totally outside the normal DCG model.

The primary goal of the system, automatic syntactic error recovery, has been reached to the extent that seems to be normal for this technique ([Har 77], [Pem 80]). The quality of error handling was analyzed by parsing syntactically erroneous Edison programs with the system. In ordinary cases the parser was able to find most of the actual errors, but on the other hand it reported quite many nonexistent errors (in some extreme cases the number of extraneous error messages was even larger than the number of actual error messages). When recovering from an error, the parser also skips some portion of the source program which in Edison's case is typically the whole incorrect structure (expression, statement, etc.). In the correction mode the amount of omitted text is usually smaller since user-supplied corrections can locally turn an invalid structure into a legal one.

6. Discussion and Future Work

This work shares some of the contributions with previous research on parsing and Prolog. Deterministic parsing with Prolog based on LL(*1*) grammars is discussed in [Abr 86], some systems circumvent the problems with left-recursion by employing bottom-up parsing (e.g. BUP [MTK 86], AID [Nil 86]), etc. However, as far as we know the automatic error handling mechanism is unique in our system. Also the DCG transformations (albeit merely on the context-free part of the grammar) are something new. We emphasize the methodological aspect in our system; of course the same tasks could be carried out by the user as well (we have produced yet another Edison parser as a DCG with explicit error handling [Paa 89]) but that would significantly lower the conceptual level of the DCG notation.

Restricting the implementation on LL(*1*) grammars with FIRST and FOLLOW sets imposes some problems compared with the conventional implementation (besides reducing the set of accepted grammars). In our DCG variant it is not sensible to make use of terminal variables, as in

number → [C], {is_number(C)}.

This would include variable C in FIRST(number), and the consequence would be that a syntactically erroneous number symbol would not be detected by the parser (since each possible token *t* would be considered valid through unification C=*t*). Another problem of similar nature is that a grammar with the following alternative productions is not LL(*1*) in our sense:

factor → [C], {is_number(C)}.
factor → [id].

The reason to this is that the sets FIRST([C]) and FIRST([id]) are not considered disjoint (again since C always unifies with id). A general solution to these problems is hard to find. In both example cases we could and actually should use procedure is_number to generate all the possible ground patterns for C and make use of this pattern set instead of C in computing the FIRST and FOLLOW sets, but in general such lexical auxiliary procedures are rather hard to automatically locate in a DCG.

One interesting problem to be solved in the future is to integrate lexical analysis with parsing in DCGs. As noted in chapter 2, the traditional DCG formalism does not support such an integration, and we also have excluded it from our implementation. Another topic for the future is to base parsing and error recovery on the translation from DCG into ordinary Prolog, as is done in conventional implementations. This strategy would certainly be more efficient than our current one: besides that meta-interpretation as the implementation method was shown to be rather inefficient, conceptually the relation between a translation-based implementation and the meta-interpreter-based implementation clearly bears an analogy to the relation between (faster) parser programs and (slower) table-driven parsers. In the translation mode it would also be easier for the user to correct a non-LL(*1*) grammar or to retain the semantics during context-free transformations, since all the implementation-dependent information (such as the FIRST and FOLLOW sets) that is currently hidden within the meta-interpreter would be explicitly available in terms of Prolog.

Acknowledgements. We appreciate Prof. Esko Ukkonen's participation in a number of fruitful discussions on the topic.

References

[Abr 86] Abramson H.: Sequential and Current Deterministic Logic Grammars. In: Proc. of the 3rd International Conference on Logic Programming, London, 1986. Lecture Notes in Computer Science 225, Springer-Verlag, 1986, 389-395.

[ASU 86] Aho A. V., Sethi R., Ullman J. D.: Compilers — Principles, Techniques and Tools. Addison-Wesley, 1986.

[Bri 82] Brinch Hansen P.: Programming a Personal Computer. Prentice-Hall, 1982.

[Col 73] Colmerauer A.: Les systemes-Q ou un Formalisme pour Analyser et Synthesizer des Phrases sur Ordinateur. Publication Interne No. 43, Dept. d'Informatique, Universite de Montreal, 1973.

[GiW 78] Giegerich R., Wilhelm R.: Counter-One-Pass Features in One-Pass Compilation — A Formalization Using Attribute Grammars. Information Processing Letters 7, 6, 1978, 279—284.

[Har 77] Hartman A. C.: A Concurrent Pascal Compiler for Minicomputers. Lecture Notes in Computer Science 50, Springer-Verlag, 1977.

[Knu 68] Knuth D. E.: Semantics of Context-Free Languages. Mathematical Systems Theory 2, 2, 1968, 127—145.

[Kos 71] Koster C. H. A.: Affix Grammars. In: Peck J. E. L.: Algol 68 Implementation, North-Holland, 1971, 95—109.

[MTK 86] Matsumoto Y., Tanaka H., Kiyono M.: BUP: A Bottom-Up Parsing System for Natural Languages. In: Logic Programming and Its Applications (van Caneghem M., Warren D., eds.), Ablex Publishing Co., 1986.

[Nil 86] Nilsson U.: Alternative Implementation of DCGs. New Generation Computing 4, 4, 1986, 383—399.

[Paa 89] Paakki J.: Comparison of Compiler Writing Methods: An Experiment. In: Proc. of the 13th Information Technologies Conference, Sarajevo, 1989. Science and Research Council of Bosnia and Hertzegovina, 1989, R 122.

[Pem 80] Pemberton S.: Comments on an Error-Recovery Scheme by Hartmann. Software Practice and Experimence 10, 3, 1980, 231—240.

[PeW 80] Pereira F., Warren D.: Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence 13, 1980, 231—278.

[Qui 86] Quintus Computer Systems, Inc: Quintus Prolog Reference Manual, Version 6, 1986.

[StS 86] Sterling L., Shapiro S.: The Art of Prolog. The MIT Press, 1986.

[Top 89] Toppola K.: An Error-Recovering DCG Feature (in Finnish). Report C—1989—23, Department of Computer Science, University of Helsinki, 1989.

[WeM 80] Welsh J., McKeag M.: Structured System Programming. Prentice-Hall, 1980.

Appendix

An example session, starting with automatic error recovering and finishing with user-supplied local correction. The commands by the user are given in **bold**.

```
? — consult(dcg).
yes
? — read_grammar(g1).
The grammar is not LL(1).
yes
?— list_grammar.
 (1) expr    —→ expr,"+",term.
 (2) expr    —→ term.
 (3) term    —→ term, "*",factor.
 (4) term    —→ factor.
 (5) factor   ---→ "(",expr,")".
 (6) factor   ---→ "id".
yes
? — transform(e).
Eliminating left recursion ...
yes
? — list_grammar.
 (1) expr     ---→ "(",expr,")",term1,expr1.
 (2) expr     ---→ "id",term1,expr1.
 (3) term     ---→ "(",expr,")",term1.
 (4) term     ---→ "id",term1.
 (5) factor   ---→ "(",expr,")".
 (6) factor   ---→ "id".
 (7) expr1    ---→ "+",term,expr1.
 (8) expr1    ---→ " ".
 (9) term1    ---→ "*",factor,term1.
(10) term1    ---→ " ".
yes
? — parse("id+id").
Parsing completed, 0 errors detected.
yes
? — parse("id*(id—id)+id").
id*(id
--- Error 1 ---
Unexpected symbol(s) met and skipped:
— id
Parsing completed, 1 errors detected.
yes
? — correction(on).
yes
? — parse("id*(id—id)+id")
id*(id
--- Error 1 ---
** __ **
One of the following expected:
* + )
Replace(r)/insert(i)/delete(d) token:
==→ r(+).
Parsing completed, 1 errors detected.
yes
```

Uniform approach to parameter transmission mechanisms, coercions, optional parameters and patterns*

MATTI JOKINEN

Computer Center
University of Turku
SF—20500 Turku
Finland

Abstract

The formal parameter part of a procedure can be regarded as a mapping from the set of arguments into the set of environments. If environments and environment-valued functions are treated as first-class objects, a number of useful linguistic features can be constructed from a small set of elementary building blocks; such features include the most parameter transmission mechanisms, implicit conversions, conditional clauses based on pattern matching, and optional, repeatable and variable-type parameters.

1. Introduction

Programming languages use numerous variants of mappings of the general form $S \to V$, where S is a finite set of character strings and V is the universe of data objects. Such mappings can be divided into three main categories:

- *Evaluation environments* bind the free identifiers of programs into data objects. Although they are defined by declarations embedded in the program text, they tend to belong to the meta universe outside the domain of data objects. Most programming languages provide no method of identifying them by name or referring to them as entities.
- *Packages* are used as library modules, and their components are mostly types and procedures. They are often used for information hiding. They are typical second-class objects which may have names but must be completely defined at compile time.
- *Records* are designed for storing runtime data. In most modern programming languages they are first-class objects which can be created and modified at runtime.

The distinction between the three concepts makes implementation simpler, but conceptually it is more or less arbitrary. Advantages of a uniform approach are obvious [1, 4, 9]. The idea of combining the concepts is not new: Simula classes [2] are used in all three roles.

* Lecture presented at the 1st Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

2. Language

Records, packages and evaluation environments are treated uniformly in this paper and they are all called environments. We shall design a programming language that uses environments extensively as first-class objects. Details of syntax and semantics are of minor interest here, and the language will not be defined rigorously; it is solely a tool for discussing various cases where environment-valued functions prove to be useful.

2.1. Environments

An environment can be created with a clause

$$\{e_{11} \mapsto e_{12}, \ldots, e_{n1} \mapsto e_{n2}\}$$

where e_{ij}s are arbitrary expressions. Clauses e_1 and e_2 can be evaluated in any order or interleaved; this allows some extra freedom in optimization. Each e_{i1} must evaluate to a string. The resulting environment binds the strings to the values of expressions e_{i2}. Standard procedure *select* can be used to find the value of an identifier in an environment. The value of *select*$[e, x]$ is the value bound to string x in environment e. Both operands can be arbitrarily complex expressions. Procedure *econcat* concatenates two or more environments. Clause *econcat*$[e_1, \ldots, e_n]$ returns an environment that contains the combination of bindings from environments $e_1, \ldots, e_n$. If an identifier is bound in more than one e_i, its value is taken from the last one. An environment can also be used in a clause

$$\textbf{with } e_1 \textbf{ do } e_2$$

where clause e_1 evaluates to an environment. The value of this clause is the value of e_2, whose free identifiers are bound as in the environment yielded by e_1. The whole program is implicitly embedded in an environment that contains the definitions of standard identifiers.

2.2. Procedures

The definition of a procedure usually looks something like

$$p = \textbf{proc}\,(x_1{:}t_1, \ldots, x_n{:}t_n)e$$

where e is the body of the procedure. The call of this procedure is written as

$$p(e_1, \ldots, e_n)$$

where the result of clause e_i is of type t_i. In the simplest case the effect of the call is that the body e of p is evaluated in an environment in which each x_i is bound to the value of e_i. But in many programming languages parameter transmission is more complicated. The values may undergo various conversions before they are bound to formals. Parts of the data objects may be copied. Sometimes the conversion process may involve more than a single formal-actual pair and the number of actuals may be different from the number of formals. There may be optional parameters which get certain default values if omitted in the call, or a single actual may define the values of several formals, as conformant array parameters in Pascal [6]. Implicit actions allow a more compact notation and their proper use may thereby improve readability. Unfortunately the rules are usually built into the language, and although modern languages allow the definition of application-specific types they rarely [8] provide any way to extend implicit actions to user-defined types.

The FEXPR feature of Lisp [7] is one method to give the programmer more control over the actual parameters. The list of actual parameters is passed as such and can be freely manipulated in the called routine. The method relies on the representation of programs as list structures and the existence of a user-callable EVAL function. Another approach to handle optional, repeatable and variable-type parameters has been suggested by Ford and Hansche [3] and Prasad [5]. Their methods include syntax extensions to specify formal and/or actual parameters with such properties, and special statements or standard functions to test the existence of optional parameters, the number of repeatable parameters and the actual type of variable-type parameters. These mechanisms, unlike the FEXPR feature, were designed as extensions to strongly typed languages.

If parameters are passed by value, the call is equivalent to the following with-clause:

$$\textbf{with}\ \{"x_1"\mapsto e_1, \ldots, "x_n"\mapsto e_n\}\ \textbf{do}\ e.$$

Thus the formal parameter part $(x_1: t_1, \ldots, x_n: t_n)$ can be regarded as a function that maps the argument tuple into an environment. Since environments are first-class objects, it is natural to consider also the formal parameter part as an ordinary procedure. Any environment-valued procedure can then be freely used as a formal parameter part of another procedure. A procedure object is created with a clause

$$\textbf{proc}\ e_1 \Rightarrow e_2$$

where e_1 is an arbitrary clause that evaluates to an environment-valued procedure (from now on all such procedures will be called formals).

It is convenient to reduce multi-argument procedures into single-argument procedures by treating the argument list as a tuple. A tuple object is created with a clause $[e_1, \ldots, e_n]$. A one-element tuple is *not* identical with its element. Expressions e_i can be evaluated in an arbitrary order, or interleaved. Procedures with no parameters formally take an empty tuple as an argument. Procedure invocations are written as

$$e_1\ e_2$$

where clause e_1 evaluates to a procedure and e_2 evaluates to its argument. If the value of e_1 is **proc** $f \Rightarrow b$, the invocation is equivalent to **with** $f e_2$ **do** b. For convenience, certain operators will be written in their familiar infix or postfix notation. For example, we shall write $x := y$ instead of $:= [x, y]$.

2.3 Basic formal generators

The language must contain a set of standard formals, or formal generators, as elementary building blocks for user-defined procedures. We shall first introduce a procedure named *atomf*, which generates "atomic" formals. It accepts as an argument a 2-tuple $[s, t]$, where s is a string and t is a type. The value of the clause

$$atomf\,[s,t]$$

is a procedure that maps an object x (of type t) into an environment that binds s to x.

$$atomf[s,t]\,x = \begin{cases} \{s \mapsto x\}, & \text{if } x \text{ is of type } t. \\ \textbf{abort} & \text{otherwise} \end{cases}$$

If x is not of type t, the call causes a failure, a termination without any result, represented by the clause **abort**. Failures can be trapped in case-clauses, as will be seen later; untrapped failures are propagated to upper-level clauses. Note that s may be an arbitrary string-valued expression, and it is the value of s (rather than the identifier s) that becomes bound in the environment. For example,

$$atomf[\text{"}n\text{"}, int]\,4 = \{\text{"}n\text{"} \mapsto 4\}.$$

To make formals look more familiar, the following sugared syntax is defined for calls of *atomf*:

$$x{:}t = atomf[x, t].$$

For tuple arguments we first introduce a procedure, denoted by *nullf*, that accepts an empty tuple as its argument and returns an empty environment. Thus

$$\begin{aligned} &nullf[\;] = \{\;\} \\ &nullf\,x \;= \textbf{abort}, \quad \text{if } x \neq [\;]. \end{aligned}$$

Next we introduce a procedure, denoted by *fconcat*, that maps 2-tuples of formals to formals. The value of the clause $fconcat[f_1, f_2]$ is a formal that accepts as its argument a nonempty tuple whose first element is accepted by the formal f_1 and whose tail is accepted by the formal f_2. The result of the concatenated formal is an environment which is constructed by combining the environments yielded by f_1 and f_2.

$$\begin{aligned} &fconcat[f_1,f_2][e_1, \ldots, e_n] = econcat\,[(f_1\,e_1), f_2\,[e_2, \ldots, e_n]] \\ &fconcat[f_1,f_2][\;] = \textbf{abort} \\ &fconcat[f_1,f_2]\,x = \textbf{abort}, \quad \text{if } x \text{ is not a tuple.} \end{aligned}$$

For convenience, we shall often use an additional formal generator *tuplef*, which can be defined in terms of *nullf* and *fconcat*:

$$\begin{aligned} &tuplef[\;] = nullf \\ &tuplef[f_1, f_2, \ldots, f_n] = fconcat\,[f_1,\, tuplef[f_2, \ldots, f_n]]. \end{aligned}$$

2.4. Types

Since type checks occur at runtime, there must be a sensible action taken when a type check fails. A failing type check is defined equivalent to the execution of **abort**. In the examples to follow we will use standard types *int*, *real*, *string*, *anyenv*, *anytuple*, *any* and *type*, and type constructors **ref**, **union**, **tuple** and $\rightarrow$. Type **ref** t is the type of pointers to t-typed cells. Type **union**$[t_1, \ldots, t_n]$ is a coalesced union of types $t_1, \ldots, t_n$. The value space of a union type is the set-theoretic union of the value spaces of component types. Type **tuple**$[t_1, \ldots, t_n]$ is the type of tuples $[x_1, \ldots, x_n]$, where x_i is of type t_i. Clause $t \rightarrow u$ denotes the type of functions with domain t and range u. Identifier *anyenv* denotes the type of all environments, *anytuple* denotes the union of all tuple types and *any* denotes the union of all (nonunion) types. Identifier *type* denotes the type of all types (including or excluding *type*).

Union types (either the **union** constructor or *any*) are essential to the expressive power of the abstraction mechanism. Other types are more or less optional, replaceable by each other, or required only in specific examples.

2.5. Case-clause

Many modern languages have union types and a conditional clause that allows a safe access to the contents of a union. Such a clause will be needed in all the examples below. The syntax and semantics of the clause can be defined elegantly with generalized formals. The syntax is

$$\textbf{case}\ e\ \textbf{in}\ f_1 \Rightarrow e_1,\ \ldots, f_n \Rightarrow e_n\ \textbf{else}\ f_{n+1} \Rightarrow e_{n+1}$$

where the values of clauses f_1 to f_{n+1} are formals. The else-part is optional. The clause is evaluated by first evaluating the clause e and then invoking formals f_1 to f_n (in an unspecified order) using the value of e as the argument. If the invoked formal f_i returns an environment, the clause e_i is evaluated in that environment and the value of e_i becomes the value of the case-clause. If f_i fails, the next formal is tried. If all the formals f_1 to f_n fail, the optional formal f_{n+1} is invoked and the clause e_{n+1} is evaluated in the resulting environment. If f_{n+1} fails, or if there is no else-part, the case-clause fails.

3. Applications

3.1. Implicit type conversions

As a simple example, let us define a generator for formals that accept either a real or an integer as their actual argument and convert it into a real in the latter case. Standard procedure *inttoreal* performs the conversion explicitly.

```
intreal = proc ("id": string)⇒
                  proc ("x": union [int, real])⇒
                          (id: real) (case x in ("r": real)⇒r,
                                                ("n": int)⇒inttoreal n)
```

Here the case-clause is used to compute the argument of (*id*: *real*). Type **union** [*int*, *real*] could be replaced with the type *any*. Formal *intreal*[*x*] would normally be used in definitions of arithmetic functions. However, *atomf* [*x*, *real*] could be used in cases where an integer argument makes no sense. For example, assume that we need a procedure that computes the integral of a given function f over a closed interval $[a, b]$ in the accuracy *eps*. The header of the procedure might look like this:

```
proc tuplef ["f": real→real,
             intreal "a",
             intreal "b",
             "eps": real]⇒ ...
```

As an analogous but more specialized example, let us define a generator for formals that accept as an argument a month represented either as an integer or as a string:

```
proc ("id": string)⇒
     proc ("x": union [int, string])⇒
        (id: int) (case x in
                           "n": int⇒
                               if n<1 or n>12 then abort else n,
                           "s": string⇒
                               if s="January" then 1
                               else if s="February" then 2
                               ...
                               else if s="December" then 12
                               else abort)
```

3.2. *Parameter transmission mechanisms*

Transmission mechanisms are closely related to types. If the type system of the language is rich enough, transmission by various mechanisms can be reduced to transmission of various types of data [10]. Call by reference is equivalent to transmission of a parameter of type **ref** *t*. Call by name is equivalent to transmission of a parameter of type $void \to t$, where $void$=**tuple**[ ]. Call by need is equivalent to transmission of a recipe, an object of type **ref union**[*t*, $void \to t$]. However, the programmer may still want to think in terms of transmission mechanisms rather than in terms of types. To make the underlining type system transparent, an argument should undergo an implicit type conversion when it is transmitted further by a different method.

We shall first define two auxiliary procedures. *Rcp_value* generates procedures that compute values of recipes:

```
rcp_value=proc ("t": type)⇒
     proc ("x": ref union[t, void→t])⇒
        case x↑ in
             ("y": t)⇒y,
             ("f": void→t)⇒(with {z⇝f[ ]} do (x:=z; z)).
```

Here *x*↑ denotes the contents of the cell pointed to by *x*. Components of the serial clause ($x:=z$; z) are evaluated from left to right, and the value of the clause is the value of the last component. The other auxiliary procedure *rcpdefs* just generates two shorthand notations, *rcp* and *u*:

```
rcpdefs=proc ("t": type)⇒
     {"rcp"↦ref union[t, void→t],
      "u"↦union[t, void→t, ref t, rcp]}.
```

Call by value, name, need and reference, and all the required conversions, can now be defined with the following procedures:

```
value=proc tuplef["id": string, "t"; type]⇒
     with rcpdefs t do
          proc("x": u)⇒
```

```
            {id↦case x in  ("y": t)⇒y,
                           ("p": ref t)⇒p↑,
                           ("f": void→t)⇒f[ ],
                           ("r": rcp)⇒rcp_value t r}

name=proc tuplef["id": string, "t": type]⇒
    with rcpdefs t do
        proc("x": u)⇒
            {id↦case x in  ("y": t)⇒(proc nullf⇒y),
                           ("p": ref t)⇒(proc nullf⇒p↑),
                           ("f": void→t)⇒f,
                           ("r": rcp)⇒(proc nullf⇒rcp_value t r)}

need=proc tuplef["id": string, "t": type]⇒
    with rcpdefs t do
        proc("x": u)⇒
            {id↦case x in  ("y": t)⇒new rcp y,
                           ("p": ref t)⇒new rcp (p↑),
                           ("f": void→t)⇒new rcp f,
                           ("r": rcp)⇒r}
```

where clause (*new rcp e*) allocates a new cell of type *rcp*, initializes its contents to *e* and returns a pointer to the cell.

```
reference=proc tuplef["id": string, "t": type]⇒
    with rcpdefs t do
        proc("x": u)⇒
            {id↦case x in  ("y": t)⇒new t y,
                           ("p": ref t)⇒p,
                           ("f": void→t)⇒new t (f[ ]),
                           ("r": rcp)⇒new t (rcp_value t r)}.
```

Call by result cannot be implemented in this way because it involves implicit actions at the termination rather than at the start of the called procedure.

3.3. Procedures with varying number of parameters

Procedures with optional parameters can be constructed by treating the list of arguments as a tuple. One possibility is to define a fixed number of normal arguments and bind the rest of the argument tuple to one identifier. For example, in the following formal the length of the fixed part is one:

fconcat["head": t, "tail": anytuple]

Another possibility is to define optional arguments that receive default values if omitted in the call. The following procedure takes a list *L* of *3*-tuples [*name, type, default_value*] and returns a formal that accepts a tuple *A* whose i^{th} element corresponds to the i^{th} element of the tuple *L*. The length of *A* may be smaller than the length of *L*, in which case the missing elements are given default values from *L*.

```
optlist = proc ("L": anytuple) ⇒
    case L in
        nullf ⇒ nullf,
        fconcat[tuplef["name": string, "t": type, "default": any],
                "tail": anytuple] ⇒
            proc ("A": anytuple) ⇒
                case A in
                    nullf ⇒ defaults L,
                    fconcat["x": t, "rest": anytuple] ⇒
                        econcat[(name: t) x, optlist tail rest]
```

where

```
defaults = proc ("L": anytuple) ⇒
    case L in
        nullf ⇒ {},
        fconcat[tuplef["name": string, "t": type, "default": any],
                "tail": anytuple] ⇒
            econcat[(name: t) default, defaults tail].
```

If there are many optional parameters, it is more convenient to identify them by name than by position. In the list of actual arguments, an optional argument is specified as a (sub)tuple [*name, value*] in the argument list. The following procedure takes the specification of optional arguments in the same form as above, but the resulting formal accepts a list of 2-tuples in an arbitrary order:

```
optset = proc ("L": anytuple) ⇒
    proc ("T": anytuple) ⇒ econcat [defaults L, values [types L, T]].
```

Procedure *types* computes an environment that maps the names of the formal arguments to their types. This environment is used in the other auxiliary procedure to check the types of actual arguments:

```
types = proc ("L": anytuple) ⇒
    case L in
        nullf ⇒ {},
        fconcat[tuplef["name": string, "t": type, "default": any],
                "tail": anytuple] ⇒
            econcat[(name: type) t, types tail]

values = proc tuplef["ttable": anyenv, "T": anytuple] ⇒
    case T in
        nullf ⇒ {},
        fconcat [tuplef["name": string, "value": any],
                "tail": anytuple] ⇒
            econcat[(name: select[ttable, name]) value, values[ttable, tail]]
```

3.4. Patterns

In recent years it has become popular to write the formal parameter part as a pattern. A pattern is a data structure in which certain elements denote variables to be bound in an invocation. Patterns can be easily defined in our system. Below is a generator for patterns of possibly nested tuples. Variables are denoted by strings that begin with a capital letter.

```
pattern=proc ("p"): any)⇒
    case p in
        nullf⇒nullf,
        ("s": string)⇒if s[1]≧'A' and s[1]≦'Z'
                        then (s: any)
                        else (proc("t": string)⇒
                                    if s=t then {} else abort),
        fconcat["head": any, "tail": anytuple]⇒
                        fconcat [pattern head, pattern tail]
```

For example, the value of the clause
pattern ["*f*", ["*X*", "*Y*"], ["*g*", "*Z*"]]

is a formal that accepts all tuples that can be constructed by replacing "*X*", "*Y*" and "*Z*" with any objects in the tuple "*f*", ["*X*", "*Y*"], ["*g*", "*Z*"]]. Patterns for other data types can be defined in an analogous way.

In a more realistic program the types of the variables would be included in patterns and the formal generator would take care of multiple occurrences of a variable. A quotation mechanism is also desirable to permit arbitrary constant terms in patterns (for example, strings beginning with a capital letter). These features can be defined in the language without difficulty.

4. Implementation

The programming language designed in the preceding sections is based on late binding and runtime type checks. That is typical of interpreted languages, and the reader may wonder whether the ideas presented in this paper are of any use in compiled languages where efficiency is considered more important. Fortunately the quality of the code can be greatly improved with relatively simple optimization methods.

General environments can be represented as association lists, hash tables, binary trees, or combinations of these (and possibly other) structures. However, in the special case in which the bound identifiers are known at compile time, an environment can be represented exactly like a conventional record: the components of the environment can be stored in consecutive memory locations and the value of an identifier is found by adding a static offset of the base address of the environment. A single-element environment $\{x \rightsquigarrow v\}$ is represented exactly as the object v. Assume that in an invocation (p e) the value of p is completely known at compile time and defined by

$$p = (\textbf{proc}\ (x\colon t) \Rightarrow u)$$

If, in addition, x is a string constant and e is guaranteed to be of type t, the environment produced by the formal can be used as the lower part of the activation record of a procedure as in conventional languages and the invocation can be translated into the instruction sequence

$$\mathit{code}(e);\ \mathrm{jsub}(u)$$

where $\mathit{code}(e)$ evaluates e and leaves its value on the top of the stack, and $\mathrm{jsub}(u)$ saves the program counter and transfers control to the body u of the procedure.

Next assume that p is defined by

$$p = (\textbf{proc}\ \mathit{tuplef}\,[f_1, \ldots, f_n] \Rightarrow u)$$

where each f_i is completely known at compile time, e_i is known to be of type suitable as an argument for f_i, and the result of f_i is a mini-environment $\{x_i \rightsquigarrow v_i\}$ where x_is are string constants and $x_i \neq x_j$ whenever $i \neq j$. The invocation can now be translated into the instruction sequence

$$\mathit{code}(f_1\ e_1);\ \ldots;\ \mathit{code}(f_n\ e_n);\ \mathrm{jsub}(u).$$

Procedure calls involving more complicated formals can usually be optimized with partial evaluation. From the semantics of the language the following evaluation rules can be derived:

1. Clause (**proc** $e_1 \Rightarrow e_2)e_3$ can, by definition, be reduced to (**with** e_1 e_3 **do** e_2).
2. Clause (**if** *true* **then** e_1 **else** e_2) reduces to e_1, and (**if** *false* **then** e_1 **else** e_2) reduces to e_2.
3. Clause (**case** e **in** $f_1 \Rightarrow e_1, \ldots, f_n \Rightarrow e_n$ **else** $f_{n+1} \Rightarrow e_{n+1}$) reduces to (**with** f_i e **do** e_i), where f_i is the first such formal that (f_i e) does not fail. If all invocations (f_i e) fail, the case-clause reduces to (e; **abort**). In the latter case the clause e can be eliminated if the compiler can conclude that e has no side effect. Note that the actual value of the clause e need not be known.
4. Clause (**with** $\{x_1 \rightsquigarrow e_1, \ldots, x_n \rightsquigarrow e_n\}$ **do** e) can, under certain conditions, be reduced by substituting the occurrences of x_i with e_i in e; the substituted e replaces the with-clause. This reduction rule can always be applied if clauses e_i have no side effects. But even if e_i does have a side effect, the substitution is legal if x_i *occurs in e exactly once*. If left to right evaluation is to be guaranteed, an additional constraint is be required: identifier x_i can be replaced by e_i in e only if there is no subclause in e that precedes the occurrence of x_i and may have a side effect. This additional constraint is actually satisfied in most cases that occur in practice, but the compiler may have difficulties in verifying it. The rule becomes simpler and more general if the requirement of left-to-right evaluation is relaxed.
5. The first component of a serial clause (e_1; e_2) can be moved into the front of a structured clause in the following cases:

$[\ldots, (e_1;\ e_2), \ldots]$
proc $(e_1;\ e_2) \Rightarrow e_3$
$(e_1;\ e_2)e_3$
$e_3\ (e_1;\ e_2)$
$\{\ldots(e_1;\ e_2) \rightsquigarrow e_3, \ldots\}$
$\{\ldots e_3 \rightsquigarrow (e_1;\ e_2), \ldots\}$

```
with $(e_1; e_2)$ do $e_3$
if $(e_1; e_2)$ then $e_3$ else $e_4$
case $(e_1; e_2)$ in $e_{11} \Rightarrow e;_{12}, \ldots$
```

The reader is encouraged to apply the rules to the formals defined in the preceding section.

Rules 1 and 4 together may lead to a nonterminating sequence of reductions. Since compilers have difficulties in recognizing the diverging clauses, it is probably better to let the programmer specify which clauses shall be evaluated at compile time. Abstract formals could then be regarded as sophisticated macros rather than ordinary procedures.

Acknowledgments

The author wants to thank Reino Kurki-Suonio and Robert Johnson for their helpful comments.

References

[1] BURSTALL R. and LAMPSON B. W., 'A kernel language for abstract data types' and modules, *Proceedings of the International Symposium on Semantics of Data Types*, Sophia-Antipolis, France, 1—50 (1984).

[2] DAHL O-J., MYRHAUG B. and NYGAARD K., *Common Base Language*, Norwegian Computing Centre (1970).

[3] FORD G. and HANSCHE B., 'Optional, repeatable and varying type parameters', *SIGPLAN Notices* **17:2**, 41—48 (1982).

[4] GELENTER D., JAGANNATHAN S. and LONDON T., 'Environments as first class objects', *Proceedings of the 14th conference on Principles of Programming Languages*, Munich, West Germany, 98—110 (1987).

[5] PRASAD V. R., 'Variable number of parameters in typed languages', *Software—Pratice & Experience*, **10**, 507—517 (1980).

[6] *Specification for Computer Programming Language Pascal*, International Organization for Standardization, Switzerland (1983).

[7] STOYAN H., *Lisp-programmierhandbuch*, Akademie-Verlag, Berlin (1978).

[8] STROUSTRUP B., *C+ + Programming Language*, Addison-Wesley (1986).

[9] WEGNER P., 'On the unification of data and program abstraction in Ada', *Proceedings of the 10th conference on Principles of Programming Languages*, Austin, Texas, 257—264 (1983).

[10] VAN WIJNGAARDEN A. et al., *Revised Report on the Algorithmic Language Algol 68*, Springer-Verlag (1976).

Generation of test cases for simple Prolog programs*

PEKKA KILPELÄINEN, HEIKKI MANNILA

University of Helsinki, Department of Computer Science
Teollisuuskatu 23, SF-00510 Helsinki

Abstract

We describe a general method for producing complete sets of test data for Prolog programs. The method is based on the classical competent programmer hypothesis from the theory of testing, which states that the program written by the programmer differs only slightly from the correct one. The nearness is expressed by postulating a class of possible errors, and by assuming that the written program contains only errors from this class. Under this assumption the test cases produced by the method are enough to ensure the correctness of the program. The method is based on a result showing that it is sufficient to consider programs from which the written one differs by a single error. Test cases are produced by forming a path condition consisting of equations and universally quantified inequations, and solving the condition. The method is particularly easy to implement for the class of iterative programs; for general programs it can be used as a component of an interactive tool.

1. Introduction

One of the attractive properties of the Prolog programming language is that testing is quite easy. Each predicate can be tested as soon as it has been written. One does not have to write separate test programs, as in many conventional programming languages and environments.

Although testing Prolog programs is easy, the problem of choosing sets of test data is as difficult as in conventional languages. How do we know that the inputs we use for testing really test the program adequately?

There is a fairly large amount of research on testing of programs and systems written in conventional programming languages (see, e.g., the books by Beizer

* This work is supported by the Academy of Finland.

** Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

[Be84] and Myers [My79]). Also the theory of testing has been developed [BA82, GG75, DMLS78, MR89, Ho86]. This paper shows how these ideas can be applied to the generation of test cases for Prolog programs.

We describe a method for producing sets of test cases which are complete in a certain exact sense. For example, for the program

```
member(A, [A|_]).
member(A, [_|X]) :- member(A, X).
```

our method generates the test queries

```
?-member(a, [b]).
?-member(a, [a]).
?-member(a, [b, b]).
?-member(a, [b, a]).
```

These queries can be seen to test the member predicate in a quite natural way.

Our method is based on the competent programmer hypothesis [DMLS78, GG75], which assumes that the program P written by the programmer is fairly close to the intended program Q. We formalize this along the lines of [DMLS78] and [Br80] by assuming there is a class M of possible modifications, and that P and Q differ only by one or more modifications from M. Intuitively the modifications are inverses of possible programming errors. In Prolog the class M can contain modifications like exchanged variables, missing or extraneous functors or missing arguments. The competent programmer hypothesis is used by generating test cases which show the difference between the written program and any other nonequivalent program which differs from it only by modifications in class M. Hence the test cases show the difference of the written and the intended program (if there is any). The equivalence criterion used is based on the box model trace of the predicates. Hence top level tracing must be used in running the test cases to observe all their properties.

The above test cases for member were produced by considering errors in variable names. If the class of modifications contains also missing functors, then the test case set would include the additional query

```
?-member(a, a).
```

The choice of the class of modifications M is fairly important for the usefulness of the method: the larger M is, the more likely it is that the formalization of the competent programmer hypothesis holds. On the other hand, a large set M tends to produce more test cases. Fortunately, our method is not very dependent on the properties of M, so that variety of choices can be used.

Our method faces the difficulties inherent in every test data generation method. If the predicate p to be tested calls some other predicate, say q, we have to be able to generate inputs satisfying q. No system can automate this for all possible predicates q, as determining whether q ever can succeed is an unsolvable problem. Therefore our system is particularly well suited for simple programs, e.g., the *iterative* programs defined in [SS86]. Our work is fairly close in spirit to the mutation testing approach [DMLS78] and especially to the work of Brooks [Br80] on generating test cases for Lisp programs.

Work on generating test data is in a sense complementary to the interesting work done on debugging Prolog programs by Shapiro and others (see [Sh82, Pe86]). Algorithmic debugging aims at methods for finding the cause of an erroneous test output; test data generation tries to help in the process of finding the inputs showing the presence of an error. Our work can also be seen to be complementary to the work done on program synthesis [Sh82, MCM83, MCM86], which tries to move from illustrative examples to programs. We move in the opposite direction.

The rest of this paper is organized as follows. Section 2 describes the general framework by defining the concept of complete test data. It also describes a naive method for producing test data and points out its deficiencies. Section 3 contains the theoretical result showing that instead of all programs which can be obtained from the original program P by one or more modifications it suffices to consider those programs which differ from P by one modification only. This is crucial in obtaining reasonably sized sets of test cases.

Section 4 discusses how we find a test case which illustrates the difference of program P and a program obtained from P by applying one modification. We show that the existence of such an input can be characterized by giving a formula containing equations and universally quantified inequations [LMM86]. In Section 5 we discuss how inputs satisfying these formulas are found for simple programs. Section 6 is a short conclusion. For reasons of space some straightforward technical definitions have been omitted.

2. Framework

We consider the testing of a predicate p which has been defined by giving a program P for p, consisting of a list of clauses for p and the definitions of other predicates. We want to generate queries of the form

$$?\text{-}p(d_1, d_2, \ldots, d_k).$$

where k is the arity of p and $d_1, \ldots, d_k$ are terms, such that these queries test predicate p completely in some sense. We call $d = d_1, \ldots, d_k$ an *input* for the program P to achieve compatibility with the usual terminology in the theory of testing. To discuss testing one has to specify the properties of program one is interested in. The basic choice is to consider input-output relations, i.e., the function/relation computed by the program. However, this gives little information about the program. Therefore in the theory of testing it is usual to consider traces of program execution, which give more information about the computations (see, e.g., [Br80] for a discussion of the usefulness of traces).

Given a program P, we define the *top-level trace* of P on input d, denoted by $P(d)$. This is, intuitively, the box model trace of the query

$$?\text{-}p(d).$$

limited to the definition of p and with the unification attempts explicitly represented. For example, let P denote the following definition of append

```
append([], X, X).
append([A|X], Y, [A|Z]):- append(X, Y, Z).
```

Then the top-level trace of P on $([1, 2, 3], [4, 5], U)$ is

```
CALL append([1, 2, 3], [4, 5], U)
UNIFY with append([], X, X) FAILED
UNIFY with append([A|X], Y, [A|Z]):- ...SUCCEEDED
    CALL append([2, 3], [4, 5], Z)
    EXIT append([2, 3], [4, 5], [2, 3, 4, 5])
EXIT append([1, 2, 3], [4, 5], [1, 2, 3, 4, 5]).
```

Note that the recursive call is not traced; only the CALL and EXIT of it are represented. We omit the straightforward formal definition of the top-level trace.

Two programs P and Q for predicate p are *equivalent*, if for all inputs d the traces of P and Q are equal, i.e., $P(d)=Q(d)$ for all d. If P and Q are equivalent, we write $P \equiv Q$.

Example 1. Let P be the program

```
p(X) :- sort(X, Y),process(Y).
sort(X, Y) :- ...
```

with sort implemented by merge sort, and Q the same program, but sort implemented by quicksort. Then P and Q are equivalent, since the traces $P(d)$ and $Q(d)$ do not include any details of the sort computation. □

The above definition of equivalence is rather strict. Two equivalent programs not only have to compute the same relation for predicate p, but they have to compute it in the same way at the level of p's definition.

Equivalence of programs is undecidable, as, e.g., the halting problem can be reduced to it. Therefore one cannot expect too much from a test method which tries to generate instances separating the given program from all nonequivalent programs in a given class. If the class is wide enough, even recognizing the (non)equivalent programs cannot be done.

As mentioned in the introduction, our work is based on the competent programmer hypothesis [DMLS78]. That is, we assume that the program written by the programmer is reasonably close to the one he/she meant to write.

Following Brooks [Br80] and DeMillo, Lipton, and Sayward [DMLS78], we use this assumption by postulating a set of possible modifications M.

Example 2. We use mainly list processing programs in our examples. For such programs, a suitable class of modifications consists of the following.

- replace an occurence of variable x by variable y, for all variables x and y
- replace term $[t|t']$ by term t or t'
- replace variable x by term $[t|x]$ where t is a variable or an atom

We omit the formal definition of this class. □

We assume, that the programmer has made only errors which are inverses of some modifications in the class M. That is, the intended program differs from the written one only by one or more modifications from the class M. We assume the errors occur only in the clauses of predicate p, i.e., the testing concentrates on this one predicate.

The *neighbourhood* $M^*(P)$ of P consists of programs Q such that P can be transformed to Q by application of zero or more modifications from M. We denote the result of applying a single modification m to a program P by $P.m$. The neighbourhood $M^*(P)$ of P is defined as follows.

$$M^0(P) = \{P\}$$
$$M^i(P) = \{Q.m | Q \in M^{i-1}(P), m \in M\}, i > 0$$
$$M^*(P) = \bigcup_{i \geq 0} M^i(P)$$

Formalized in this framework the competent programmer hypothesis states that the intended program P' belongs to $M^*(P)$.

Let D be a set of inputs for program P. D is an (M-)*complete* test data set for P, if for all programs $Q \in M^*(P)$, such that Q is not equivalent with P, there is an input $d \in D$ such that $P(d) \neq Q(d)$. If the program works correctly on such a set D, then by the competent programmer hypothesis the program is the intended one.

Example 3. The five queries for member given in the introduction form a complete set of test data for member under the errors of Example 2. □

Let C be a set of programs and P a program, and D a set of test cases for P. We say that D separates P from C, if for each $Q \in C$, $Q \not\equiv P$. there is an input $d \in D$ such that $P(d) \neq Q(d)$. Thus a complete set of test cases for P separates P from $M^*(P)$.

How does one generate complete sets of test data? A naive approach would be to generate for each $Q \in M^*(P)$, $Q \not\equiv P$, an input d_Q such that $Q(d_Q) \neq P(d_Q)$, and to collect these inputs into the test data set. However, $M^*(P)$ may be infinite and usually is very large. The naive method takes time proportional to the size of $M^*(P)$ and probably produces test sets having about as many elements as $M^*(P)$ has, which is unacceptable. In the next section we show how this problem can be avoided by considering only a small subset of $M^*(P)$.

The second problem is: given programs P and Q, how do we decide whether they are quivalent or not, and if they are not, how do we generate a test revealing the nonequivalence (i.e., an input d for which $P(d) \neq Q(d)$)? These questions are considered in Sections 4 and 5.

We close this section by giving another example of a complete test data set.

Example 4. Let P be the familiar append program:

```
append([], X, X).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

Let M be the class of modifications consisting of variable name changes and errors in list functors. Then the following queries are a complete test data set for P.

```
?-append([], a, a).
?-append([a], [], [a]).
?-append([a], a, [a, a]).
?-append(a, a, a).
```
□

3. Local neighbourhoods

Let a class M of modifications and a program P be given. We want to generate a test case set D separating P from each program in $M^*(P)$. Given a program $Q \in M^*(P)$, we know that Q can be obtained from P by applying the modifications from M. That is

$$Q = P.m_1.m_2.\ldots.m_n$$

for some modifications $m_1, \ldots, m_n \in M$.

Example 5. Let the member program be as in the introduction; denote it by P. Let Q be the program

member(A).
member(A, [A|X]):- member(X, X).

Then $Q = P.\ m_1.\ m_2.\ m_3$, where m_1 drops the second argument of the fact and m_2 and m_3 change variable names in the second clause. Thus $Q \in M^*(P)$. □

Our goal is to avoid considering the entire neighbourhood $M^*(P)$. For this, we define the *local* (*M-*)*neighbourhood* $M(P)$ of program P by

$$\begin{aligned} M(P) &= M^1(P) \\ &= \{P.m | m \in M\} \end{aligned}$$

That is, $M(P)$ consist of those programs obtained from P by applying one modification from M; thus $M(P) \subset M^*(P)$.

Example 6. The program Q in Example 5 belongs to $M^*(P)$, but not to $M(P)$. □

The class $M(P)$ is far smaller than the entire neighbourhood $M^*(P)$. If there are k subgoals in the program and for each subgoal there are on the average b modifications that can change it, then $M(P)$ contains about kb elements, whereas $M^*(P)$ has about b^k elements.

In order to be able to consider only the local M-neighbourhood of P, we need to restrict the possible sets of modifications. A set of modifications M is *term-closed* for P if all programs $Q \in M^*(P)$ differing from P only with respect to a single term belong to $M(P)$. In other words, we require that all sequences of modifications to one term can be expressed as one modification. We also assume, that modifications are *shape preserving* in the sence of Brooks [Br80]. A modification is shape preserving, if it does not change the ordering and number of heads and goals in the program. Thus a shape preserving modification may change argument lists, but it preserves the overall structure of the program. These assumptions are realistic, if we think modifications as inverses of typing errors, for example.

The next theorem shows that under our assumptions a set of test cases separating the program from its local environment is enough to completely test the program.

Theorem 1. Let P be a program for predicate p, M a term-closed set of shape preserving modifications to the clauses of predicate p, and D a set of test cases that separates P from $M(P)$. Then D is a complete set of test cases for P and M, i.e., D separates program P from $M^*(P)$.

Proof. Let Q be a program in $M^*(P)$, $Q \not\equiv P$. We need to show that there is a test case d in D for which the traces $Q(d)$ and $P(d)$ differ from each other. Let d' be a test case, for which $Q(d') \neq P(d')$. Consider the first differing lines in the traces $Q(d')$ and $P(d')$; let them be the ith ones. We have the following lemma.

Lemma 1. The first differing lines in $Q(d')$ and $P(d')$ are of one of the following forms.

1. UNIFY with h' and UNIFY with h, where h' is a clause head in program Q, h a clause head in P, and $h \neq h'$,
2. CALL $r(t_1', \ldots, t_n')\theta$ and CALL $r(t_1, \ldots, t_n)\theta$, where $r(t_1', \ldots, t_n')$ is a subgoal of program Q, $r(t_1, \ldots, t_n)$ a subgoal of P, and $t_i'\theta \neq t_i\theta$ for some $i \in 1, \ldots, n$, or
3. the difference in the traces follows immediately calls to a predicate r (i.e., one trace contains an EXIT or a FAIL line, and the other contains adifferent line or no line at all). Note that because only the definition of predicate p may be modified, the call to r must be a direct or indirect call to p. □

The proof of the theorem uses induction on the number of the recursive calls of p that must be executed before the first differing lines in the traces $Q(d')$ and $P(d')$ are encountered.

For the base case, only the first two cases of the lemma are possible. In the first case the ith line of $Q(d')$ is

$$\text{UNIFY with } h' : \!-\ldots \tag{1}$$

and the corresponding line in trace $P(d')$ is

$$\text{UNIFY with } h : \!-\ldots \tag{2}$$

Let m be a modification that changes the head h. Then $P.m \in M(P)$, and $P \cdot m \not\equiv P$, since the ith lines in the traces $P.\ m(d')$ and $P(d')$ differ. Therefore there is an input d in D such that $P(d) \neq P.\ m(d)$. Let the first lines where the traces $P(d)$ and $P.\ m(d)$ differ from each other be the jth ones. Now, if the traces $Q(d)$ and $P(d)$ differ before line j, $Q(d) \neq P(d)$. Otherwise the jth lines are (1) and (2), i.e., they are different, and again $Q(d) \neq P(d)$.

In the second case the ith line in trace $Q(d')$ is

$$\text{CALL } r(t_1', \ldots, t_n')\,\theta$$

and the corresponding line in $P(d')$ is

$$\text{CALL } r(t_1, \ldots, t_n)\,\theta$$

and there is a $k \in 1, \ldots, n$ such that $t_k'\theta \neq t_k\theta$. By the term-closedness of M there is a modification $m \in M$ for which $t_k \cdot m = t_k'$. Now the ith line in the trace $P.\ m(d')$ is

$$\text{CALL } r(t_1, \ldots, t_k.m, \ldots, t_n)\,\theta$$

and the corresponding line in $P(d')$ is

$$\text{CALL } r(t_1, \ldots, t_k, \ldots, t_n)\,\theta$$

Because the traces differ, $P.\,m \not\equiv P$. Since $P.\,m \in M(P)$, there is an input $d \in D$ such that $P(d) \neq P.\,m(d)$. Assume that these traces are the same up to the jth line. Let the jth line of $P(d)$ be

$$\text{CALL } r(t_1, \ldots, t_k, \ldots, t_n)\,\gamma$$

and the corresponding line in $P.\,m(d)$

$$\text{CALL } r(t_1, \ldots, t_k.m, \ldots, t_n)\,\gamma$$

with a substitution γ, for which $t_k\gamma \neq t_k.\,m\gamma$. If the traces $Q(d)$ and $P(d)$ differ before the $j$tb line, then $Q(d) \neq P(d)$. Otherwise the jth line of $Q(d)$ is

$$\text{CALL } r(t_1', \ldots, t_k.m, \ldots, t_n'),$$

which is different from the corresponding line in $P(d)$, since $t_k\gamma \neq t_k.\,m\gamma$.

The induction assumption is that for all d'' such that $Q(d'')$ and $P(d'')$ differ and the number of recursive calls to p before producing the first different lines in $Q(d'')$ and $P(d'')$ is smaller than the number of recursive calls preceding the first difference in $Q(d')$ and $P(d')$, there exists a $d \in D$ such that $Q(d)$ and $P(d)$ differ.

Assume that there are recursive calls of p in the execution preceding the first differing lines in top-level traces $Q(d')$ and $P(d')$. Now all the three cases of the lemma are possible. The first two cases are as in the base case. The third case is that the first line where the traces $Q(d')$ and $P(d')$ differ is immediately after a line

$$\text{CALL } r(u_1, \ldots, u_m),$$

Because the programs Q and P can differ at the definitions of predicate p only, the execution of the call $r(u_1, \ldots, u_m)$ must contain a recursive call $p(t_1, \ldots, t_n)\theta$, and the result of this call differs in Q and P. Let $d'' = t_1\theta, \ldots, t_1\theta$. Now $Q(d'') \neq P(d'')$, and the number of recursive calls of p before the first difference of $Q(d'')$ and $P(d'')$ is smaller than the corresponding number before the first difference between $Q(d')$ and $P(d')$. By the induction assumption there exists a $d \in D$ such that $Q(d) \neq$ $\neq P(d)$. □

Now we have reduced the problem of generating complete sets of tes cases for P to the problem of separating P from the class $M(P) = \{P.\,m \mid m \in M\}$. Our test data generation method can thus be formulated as follows:

```
$D := \{\}$;
for each modification $m \in M$ do
    if $P \not\equiv P.m$ then
        generate a $d$ such that $P(d) \neq P.m(d)$;
        $D := D \cup \{d\}$;
```

The method outlined above has connections to the classical testing method known as path testing [Be84]. This method requires that every path in the program is traversed by execution of some test case. We have the following simple result.

Theorem 2. Let a class of modifications M and a program P be given. If for each subgoal in the clauses of predicate p there is a modification m altering that subgoal so that $P.\,m \not\equiv P$, then the execution of all the queries in a complete set of test data for P traverses every subgoal of the predicate p. □

4. Path conditions and separation

Suppose we are given a program P and a modification m. How do we test whether P and $P.m$ are equivalent and if they are not, generate an input d such that $P(d) \neq P.m(d)$? Such a d (if one exists) must cause the execution of P to proceed to the subgoal c altered by the modification m; additionally, d must be such that it causes different trace lines to be output by subgoals c and $c.m$.

Suppose p is defined by the clauses

$$\begin{gathered} p(t_1) \text{ :- } q_{11}(t_{11}), \ldots, q_{1a_1}(t_{1a_1}). \\ \vdots \\ p(t_k) \text{ :- } q_{k1}(t_{k1}), \ldots, q_{ka_k}(t_{ka_k}). \\ \vdots \\ p(t_u) \text{ :- } q_{u1}(t_{u1}), \ldots, q_{ua_u}(t_{ua_u}). \end{gathered}$$

Here $t_1, \ldots, t_u, t_{11}, \ldots, t_{ua_u}$ are parameter lists. Let m alter the goal $q_{kh}(t_{kh})$ in clause k $(1 \leq k \leq u,\ 1 \leq h \leq a_k)$. Then an input d such that $P(d) \neq P.m(d)$ must satisfy at least the following conditions.

1. d unifies with t_k; let θ be the unifying substitution,
2. the subgoal $(q_{k1}(t_{k1}), \ldots, q_{kh-1}(t_{kh-1}))\theta$ succeeds
3. the goals $q_{kh}(t_{kh}\theta)$ and $(q_{kh}(t_{kh}).m)\theta$ produce different trace lines.

Here (1) states that execution of the altered clause must be able to start; (2), that this execution proceeds to subgoal q_{kh}; and (3), that the resulting traces are different.

These conditions are not enough, however. We must know that unification of clause k in p's definition is attempted. There are two ways to guarantee this: we can require that no previous clause is applicable, or that no previous clause succeeds (or meets a cut). A third possibility arises when the clauses $1, \ldots, k-1$ contain no cuts. Then clause k can be reached by backtracking.

The first alternative of the condition (4) is formally expressed as

(a) t_i and d do not unify for each $i = 1, \ldots, k-1$,

and the second

(b) for each $i = 1, \ldots, k-1$, either t_i and d do not unify, or, if they do (with substitution θ_i), the execution of the resulting subgoal $(q_{i1}(t_{i1}), \ldots, q_{ia_i}(t_{ia_i}))\theta_i$ does not succeed.[1]

The conjunction of (1), (2), (3), and (4a) is called the *(strong) path condition* for modification m. The conjunction of (1), (2), (3), and (4b) is the *regular path condition* for m, and the conjunction of (1), (2) and (3) the *weak path condition* for m.

Example 7. Let P be the append-program of Example 4 and let $P' = P.m$ be the program where the second clause is

append($[A|X]$, Y, $[A|Z]$):- append(X, Y, Y).

The strong path condition for the modification m is that input d unifies with $([A|X], Y, [A|Z])$, d does not unify with $([], X, X)$, and the tracing of append

[1] For simplicity we do not discuss cuts here.

(X, Y, Y) and append(X, Y, Z) is different on d. A d satisfying these conditions is, e.g., ([1], [1], [1]). □

To generate a d such that $P(d) \neq P.m(d)$, we form the path condition for m and try to generate an input satisfying it. For this, we need a formal way of describing path conditions. This is easy to do using the concepts of unification and universally quantified inequalities [LMM86, LM87]. An example should explain how this is done.

Example 8. The path condition in the previous example can be formalized as follows.

$$U = ([A|X], Y, [A|Z]) \wedge \forall X' : U \neq ([\,], X', X') \wedge (X \neq X \vee Y \neq Y \vee Y \neq Z).$$

Here U stands for the input d. The last conjunct comes from the requirement that the traces of append(X, Y, Z) and append(X, Y, Y) differ. □

The reason for introducing regular path conditions is that sometimes the strong path condition is unsatisfiable. For example, let p be defined as follows.

$$\begin{array}{l} \mathrm{p}(A', B') :\text{-}A' < B', q(A', B'). \\ \mathrm{p}(A, B) :\text{-}r(A, B). \end{array}$$

The path condition for subgoal $r(A, B)$ altered to $r(A, A)$ is

$$(\forall A', B' : U \neq (A', B')) \wedge U = (A, B) \wedge \ldots$$

which clearly cannot be satisfied. An input reaching the end of the second clause necessarily unifies with the first clause, but fails in its body. The regular path condition contains the subformula

$$U = (A', B') \wedge (\neg(A' < B') \vee \neg q(A', B')),$$

which can be satisfied by letting A' and B' be integers and $A' \geqq B'$.

The technique we use is first to try the strong path condition. If that cannot be satisfied, we move towards the regular path condition by allowing some previous unifications to succeed but requiring that some subgoal in that clause fails. In our example this would mean including the subformula

$$U = (A', B') \wedge \neg(A' < B')$$

in the path condition. In this fashion the path condition is weakened, until it can be satisfied.

Weak path conditions arise in situations where we want to check backtracking behaviour of a program. We omit the discussion on this; in the sequel we concentrate on strong path conditions.

5. Generating inputs satisfying the path condition

Given a path condition, how do we generate an input satisfying it? Here we have to restrict our class of Prolog programs. A path condition can contain conditions of the form $q(t)$, where q is an arbitrary predicate. Generating inputs satisfying q is an unsolvable problem, as, e.g., the halting problem is reduced to it.

There are two ways out of this problem. We could try to generate inputs satisfying the subgoals just by running the subgoals. This alternative seems to be feasible in practice, but it is hardly amenable to an exact analysis.

The second way is to restrict ourselves to programs where the subgoals are easily analyzable. One such class (but by no means the only) is the class of *iterative programs*, defined in [SS86]. An iterative program for predicate p consists of clauses, where the last subgoal in each clause can be a recursive call and all other goals are calls of system predicates. For example the programs for append and member are iterative.

Given a conjunction of equations and universally quantified inequations, we collect first the equations and solve them by using (Robinson's) unification algorithm. This gives us a structure, possibly with free variables, representing the most general solution of the equations.

We then process the universally quantified inequations one by one. If for the inequation $\forall X\colon\ U \neq t(X)$ the right hand side matches the structure formed so far, we develop the structure by adding functors or atoms so that the inequation holds.

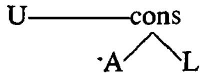
Example 9. Let p be defined by the clauses

$$\begin{aligned}&\mathrm{p}([\,]) \mathrel{:-}\ldots\\ &\mathrm{p}([A', B'|L']) \mathrel{:-}\ldots\\ &\mathrm{p}([A|L]) \mathrel{:-} B \text{ is } A+1, \mathrm{p}(L).\end{aligned}$$

and let m be the modification changing the last L in the third clause to a new variable, say X. The path condition for m is

$$U \neq [\,] \wedge (\forall A', B', L' : U \neq [A', B'|L']) \wedge B = A+1 \wedge U = [A|L] \wedge L \neq X^2$$

We start by considering the equations in this conjunction: $U=[A|L]$, $B=A+1$. We form a structure representing the value of U:



Next we check the inequations for U. The right hand side of the equation $U \neq [\,]$ does not match the above partially instantiated value of U, so we proceed to the next inequation, $\forall A', B', L'\colon\ U \neq [A', B'|L']$. The right hand side of this matches the above structure, so we have to ensure that $\forall B', L'\colon\ L \neq [B'|L']$. This can be done by instantiating L to the empty list.

The method outlined above is fairly easy to implement by brute force: we use a Prolog query, which first generates the structure from the equations and then tries possible alternatives for the free variables until a suitable case is found. A more refined method would also be quite simple to implement.

[2] Note that this inequation is not universally quantified; it comes from the requirement that the altered subgoal behave differently from the original one.

6. Concluding remarks

We have described a general method for producing complete sets of test data for Prolog programs. The method is based on the competent programmer assumption and on a theoretical result showing how one can concentrate on the local neighbourhood $M(P)$. The test cases were produced by forming path conditions for each modification and by solving them.

Several open problems remain. One is the exact class of programs for which the method can be made fully automatic. For iterative programs this can be done, but they probably do not form the largest such class. Another such class might be the programs without function symbols (Datalog).

Another problem is dealing with general programs. For those interaction with the user is necessary for succesful generation of test cases. How should the interaction be organized?

7. Acknowledgements

We wish to thank Tiina Häkkä for implementing a prototype of the system.

References

[BA82] BUDD, T. & ANGLUIN, D., Two Notions of Correctness and Their Relation to Testing. Acta Informatica 18 1982, pp. 31—45.

[Be84] BEIZER, B., Software System Testing and Quality Assurance. Van Nostrand Reinhold. USA 1984.

[Br80] BROOKS, M., Determining Correctness by Testing. Report No. STAN—CS—80—804, Computer Science Department, Stanford University. May 1980.

[DMLS78] DEMILLO, R. & LIPTON, R. & SAYWARD, F., Hints on Test Data Selection: Help for the Practicing Programmer. Computer Vol. 11 No. 4 April 1978, pp. 34—41.

[GG75] GOODENOUGH, J. & GERHART, S., Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering Vol. SE—1 No. 2 (1975), pp. 156—173.

[Ho86] HOWDEN, W., A Functional Approach to Program Testing and Analysis. IEEE Transactions on Software Engineering Vol. SE—12 No. 10 (1986), pp. 997—1005.

[LM87] LASSEZ, J-L. & MARRIOT, K., Explicit Representation of Terms Defined by Counter Examples. Journal of Automated Reasoning, 3 (1987), pp. 301—317.

[LMM86] LASSEZ, J-L. & MAHER, M. & MARRIOT, K., Unification Revisited. RC 12394. IBM-T. J. Watson Research Center Yorktown Heights, NY. November 1986.

[MCM83] MICHALSKI, R. & CARBONELL, J. & MITCHELL, T. (eds.), Machine Learning: an Artificial Intelligence Approach, Morgan Kaufmann, 1983.

[MCM86] MICHALSKI, R. & CARBONELL, J. & MITCHELL, T. (eds.), Machine Learning: an Artificial Intelligence Approach, Vol. II, Tioga 1986.

[MR89] MANNILA, H. & RÄIHÄ, K.-J., Automatic Generation of Test Data for Relational Queries. Journal of Computer System Sciences 38, 2 (April 1989), 240—258.

[My79] MYERS, G., The Art of Software Testing. Wiley. New York 1979.

[Pe86] PEREIRA, L., Rational Debugging of Prolog programs. Proceedings of the Third International Conference on Logic Programming, London 1986, pp. 203—210.

[Sh82] SHAPIRO, E., Algorithmic Program Debugging. Ph. D. Thesis, Yale University, 1982.

[SS86] STERLING, L. & SHAPIRO, E., The Art of Prolog. The MIT Press. Massachusetts 1986.

CONSTRUCTOR: A Natural Language Interface Based on Attribute Grammar*, **

Z. ALEXIN[1], J. DOMBI[1], K. FABRICZ,[2] T. GYIMOTHY,[1] T. HORVATH[1]

[1] *Research Group on Theory of Automata*
Hungarian Academy of Sciences, Somogyi u. 7., H—6720 Szeged
[2] *Attila József University, Egyetem u. 2., H—6722 Szeged*

Abstract

The paper gives an overview of a natural language interface currently being developed at the Research Group on Theory of Automata at the Hungarian Academy of Sciences in collaboration with the Attila József University, Szeged. The interface supports natural language communication between the user issuing commands as steps in plane geometry constructions and the actual graphical presentation. The Natural Language Interface named CONSTRUCTOR is described and the experiences of the authors are outlined with a view to generalizing the results thus obtained.

1. Introduction

Natural Language Interfaces [NLIs] are one of the most common applications of natural language processing. The majority of such interfaces have been developed for manipulating databases [Cliff 88].

The literature on the methodology for NLI evaluation is by and large restricted to interfaces to databases [Schr 88]. Other kinds of NLIs do not only lack general principles for objective evaluation; their value is rather hard to assess due to the fact that they are usually oriented to some special task with a microcosm of words, rules, and knowledge.

We can roughly distinguish two types of Natural Language Interfaces. Less complicated Natural Language Interfaces are based on a sentence-by-sentence analysis. As a rule, they extract information from the main constituents of sentences.

*The computer facilities provided by the Alexander von Humboldt Foundation have greatly contributed to the completion of this project.

** Lecture presented at the 1st Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

This approach allows for skipping different parts of the input while restricting parsing to finding the words of direct semantic relevance. In case of a simple NLI, knowledge representation does not go hand in hand with natural language analysis. In fact, here natural language understanding is replaced by pattern matching and pre-wired procedures. These interfaces are relatively easy to construct and they can even be made portable. An example of such a system is JAKE [JAKE 88].

On the other hand, more detailed analysis along with deeper understanding is achieved by interfaces which do not omit parts of the input considering them irrelevant. Rather, they are designed to capture the overall content of the input, therefore they are suitable for analyzing intersentential relations. Their construction, however, presupposes global understanding of what the input is about. Therefore, they can provide insight into what representing knowledge or understanding natural language means. Although, in principle, their transportability is per se precluded, later in this paper some considerations suggest that this should not be the case.

2. CONSTRUCTOR — an NLI for plane geometry constructions

CONSTRUCTOR, the NLI we are currently working on belongs to this latter group of interface systems. It has been designed to accept English sentences used as instructions for plane geometry constructions. The basic idea is to let the user issue commands whose output is a step in producing a more or less complicated geometrical construction. (A prototype NLI for plane geometry constructions is described in [Arz 85], where an interface of the simpler kind is presented.)

With CONSTRUCTOR, the main steps to be taken are:

(i) analyze the input in its entirety,
(ii) translate the result into a semantic representation,
(iii) produce, on the basis of (ii), a visualized construction,
(iv) keep track of the sequence of inputs in order to:

a) maintain control over the whole procedure of construction,
b) supply the user with a means of feed-back (evaluation).

Thus, CONSTRUCTOR is part of a program package that consists of the following basic modules.

CONSTRUCTOR itself consists of the following parts:

a lexical analyzer
a syntactic parser
an attribute evaluator
a semantic interpreter
a construction creator

These moduls can be briefly described as follows.

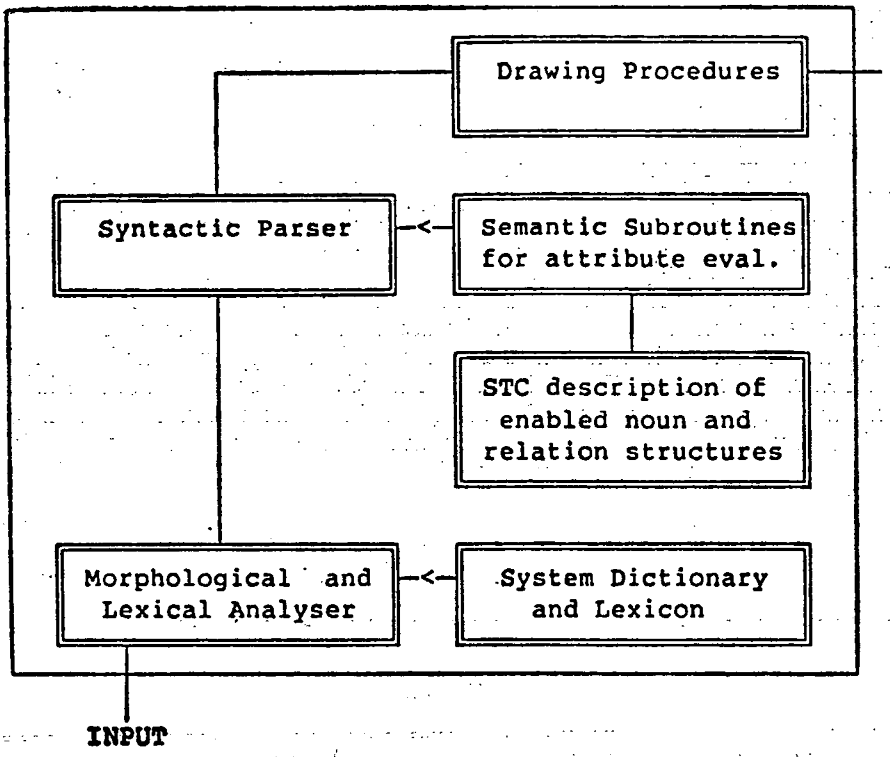

Drawing Procedures
Syntactic Parser
Semantic Subroutines for attribute eval.
STC description of enabled noun and relation structures
Morphological and Lexical Analyser
System Dictionary and Lexicon
INPUT

Figure 1.

2.1. The lexical analyzer

The lexical analyzer consists of a machine dictionary, a scanner and a morphological analyzer.

The dictionary of CONSTRUCTOR contains a lexicon of more than 300 items necessary for issuing commands (a typical set of instructions is provided with the description of the syntactic analyzer). The word stock incorporated in the lexicon makes it possible to maintain ease of reference. Thus the information contained in one instruction is related to other pieces of information from a sequence of commands.

A lexical entry consists of a word-stem (canonical form) and a set of codes that are necessary for subsequent analysis. In fact, it is in the lexical entry that basic morphological, syntactic, and semantic information is stored. Morphological features involve data for the derivation of inflected forms. From the point of view of syntax, the lexicon helps the parser assign a token to a particular word. The semantic information contained in the entry is the basic synthetised attribute underlying the final process of analysis, i.e. attribute evaluation.

The lexicon holds synonymous words to account for the fact that there is a difference in word usage among students depending on age and/or level of training (see, e.g. selection from synonymous verbs like "name", "mark", "label", "denote", or "designate" according to the above factors).

The lexical module is virtually extended by a morphological analyzer. Its function is to trace all the word-forms not found in the lexicon to their canonical lexeme. Its work is based on a tagging algorithm for the derivation of inflected word-forms. The relevant information for finding the base form (stem) of a lexeme is encoded in the lexical entry. The algorithm facilitates the derivation of all the four major parts of speech: verbs (past tense, past participle, third person in the present tense, and gerund), nouns (plural forms), adjectives (comparative and superlative degree), and adverbs (degrees and "-ly" traced back to adjectival canonical forms with the derivational path recorded). The inclusion of a morphological analyzer reduces the size of the lexicon to a minimum, while the overall amount of actual word-forms is potentially well over a thousand. In some cases actual word-forms appear in the lexicon along with their canonical form. This is due to matters of conversion, that is, some inflected forms may represent products of a change in linguistic status. For instance, the word-form "circumscribed" appears to be an adjective rather than a past participle. In this case the algorithm would yield a false result in the sense that it would analyze the form as the past participle of "circumscribe" instead of assigning it to the class of adjectives in base form. Here we create a lexical entry for "circumscribed" with the codes for an adjective. As dictionary look-up takes place prior to derivation, a match for "circumscribed" is found with no conflict with the analyzer. Table 1 shows the main morphological derivations handled by the analyzer.

Table 1.

MORPHOLOGICAL CATEGORIES	STEMS	VARIATIONS	INFLECTIONS
VERBAL MORPHEMES:	apply		-ing
		applie-	-s
(Canonical form: 'apply')		applie-	-d
NOMINAL MORPHEMES:	copy		
		copie-	-s
(Canonical form: 'copy')			
ADJECTIVAL/ADVERBIAL MORPHEMES:	big		
(Canonical form: 'big')		bigg-	-er
(Canonical form: 'great')	great		-er
(Canonical form: 'equal')	equal		-ly

2.2. Syntactic parsing

The input to the syntactic parser is a string of tokens and terminals to be processed into a sentence (or a list of sentences) with some structure assigned to the input on the basis of about two hundred re-writing rules. The parser is a hypothesis-driven (top-down) depth-first left-to-right syntactic analyzer. The syntactic rules represent a context-free grammar description. As for the type of look-ahead, the syntax is basically of the LL(1) type. The only exceptions are conjunctions together with a conjunction and more or less optional commas (","), and a few minor constructs.

The sentences that can be processed by the parser may be fairly complex. The only significant restriction imposed is that one sentence may refer to but one step of construction. It means that issuing commands like "Draw and move a right triangle..." is prohibited. On the other hand, nested sentences for object specification can be used. It means that sentences like "Draw a segment that connects points ˜*A* and ˜*B*." or "Label the line that crosses circle ˜*C* at points ˜*A* and ˜*B* by ˜1." can be freely used. To get a grasp of the range of sentences accepted by CONSTRUCTOR, consider the table below:

Table 2.

FRAME SENTENCE STRUCTURES

1. Draw two parallel lines.
   (Verb Phrase)(Noun Phrase)
2. Construct a triangle inside the circle.
   (Verb Phrase)(Noun Phrase)(Prep Phrase)
3. From point ˜A, drop a vertical line.
   (Pre-Specifier) (VP) (NP)
4. Label by ˜e a straight line that is above the circle.
   (VP) (NP) (Specifier)
5. Label by ˜J a point that divides segment ˜O˜B into parts with a proportion of 1:3.
   (VP) (NP) (Specifier within Specifier)
6. By measuring off the length of segment ˜A˜B, draw two circles with radius ˜A˜B at a distance equal to the difference between the base of the triangle and side ˜U˜V of the heptadecagon.
   (PreVP) (VP) (NP) (Specifier within Specifier)

2.3. Attribute evaluation for the basic grammatical structures

The system uses an L-attribute evaluation strategy. Its task is to compute the basic features of grammatical structures. For example, the attributes of the verbal object can be computed from the attributes synthetized for the noun phrase, the adjectival phrase, and the apposition. This computation mostly involves synthetized attribute evaluation, whereas further specification of the object (localization, relatedness etc.) requires the use of inherited attributes.

Facing the complexity of the sentences above does not appear to be a simple enterprise. Nevertheless, there do seem to be clues to semantic interpretation.

For one thing, there are some observations that can be made use of for a more thorough understanding of the semantic relations involved.

Prepositions, for example, correspond to markers of localization. Localization is taken to refer to either a place or a direction in the plane cf.:

Mark a point on the circle.
Move the triangle up.

Adjectives and nouns enter into relations of selectional restrictions, cf.:

* Draw a parallel circle.

Verbs appear to invoke one or more of the following actions:

drawing,
marking,
measuring,
manipulating.

These action types often result in overlapping actions due to the vagueness present in natural languages, cf.:

Label by $\bar{e}$ an arbitrary line.
1. Draw an arbitrary line.
2. Label it by $\bar{e}$.

Drop a vertical line $\bar{e}$.
1. Drop a vertical line.
2. Label it by $\bar{e}$.

2.4. Semantic interpretation

The result of syntactic parsing, attribute evaluation, and the observations all serve as input for semantic interpretation. The main bulk of analysis at this stage, however, is done through a metalevel description for building complex noun phrases. A part of the metalevel description is shown in Figure 2 below:

```
Objects are
                          {POSITION data structures}

IntegerPosition      is X (alias X~ coordinate): IntegerNumber;
IntegerPosition      is Y (alias Y~ coordinate): IntegerNumber;
RealPosition         is X (alias X~ coordinate): RealNumber;
RealPosition         is Y (alias Y~ coordinate): RealNumber;

                        {DESIGNATION Data structures}

Designation          is  ObjectName (alias Name);
Designation          has IntegerPosition (alias Locus);
Designation          has RealPosition (alias Locus);

Dot                  is  IntegerPosition;
Dot                  is  RealPosition;
Dot                  has Designation (alias Name);

                          {TRIANGLE data structures}

TriangleBySideType   = (EquiAngular (alias EquiLateral),
                        Isosceles, Scalene (alias General));
TriangleByAngleType = (Acute, Right, Obtuse);

Triangle             is  Dot;
Triangle             is  Dot;
Triangle             is  Dot;
Triangle             is  Designation (alias Name);

Triangle             is  of SizeType;
Triangle             is  of TriangleBySideType;
Triangle             is  of TriangleByAngleType;

Triangle             has Edge [3] (alias Side): Line;
Triangle             has Angle [3];
Triangle             has Center ~Line [3]: Line;
Triangle             has MidPoint [3]: Dot;

Triangle             has Circumscribed ~Circle: Ellips;
Triangle             has Inscribed ~Circle: Ellips;
Triangle             has Circumference: Length;
Triangle             has Area: RealNumber;
```

Figure 2.

Another difficulty is computing the relations between the objects involved in some construction. Different kinds of specifiers get evaluated by way of logical expressions and mathematical functions and equations. For example, the location "on the triangle" is computed from the equations relating to the three sides of the triangle and defining a set of points to be found "on" the triangle.

From the nodes of a given triangle we can compute the equations for the edges of the triangle. If the coordinates of the point are within the sets of points defined by the equations, then the relation "on the triangle" holds.

2.5. *The execution of commands*

The action creator receives as an input a complete specification of the object to be created and it defines the procedure to be executed with all the parameters set. The definition does not involve arriving at a possible solution of the specifications but also questions of a suitable appearance are of relevance.

Although the set of sentences presented above may seem impressive as far as syntactic and semantic complexity are concerned, the most prominent feature of CONSTRUCTOR is most likely its ability to handle reference to some previously defined object or action. This feature of CONSTRUCTOR does not simply imply a syntactic sugar of using words like "it" or "its" but, from a broader perspective, it opens the way to picturing a series of instructions as related steps of some geometrical construction. Keeping a record of what has been done makes it possible to resolve or, at least, detect, cases of ambiguity.

3. Summary

The kind of natural language interface under consideration appears to be a perspective candidate for a large scale of applications from CAD through text editors to intelligent database query languages. Our aim has been to develope a software tool for generating NLIs of this kind.

Since a software generator is considered the right tool in case

a) it can generate a major part of the software, and
b) it can provide some high level user friendly means for the description of the variable parts (cf. [Mart 83]),

we have tried to find the more or less readily standardizable parts of CONSTRUCTOR and provide a metalanguage for the specification of the variable parts.

In the case of CONSTRUCTOR that has basically been generated by a generator based on attribute grammars, the following modules seem to have been apt to generation:

— its lexical analyzer is highly suitable for generation.
— the algorithm for morphological derivation appears as a standard procedure of the lexical analyzer. We have constructed a convenient tool for dictionary maintenance.
— the syntactic parser is easily generated by PROF—LP [Gyim88] as long as the number of LL(1) conflicts is kept to a minimum. In other cases, procedures defined by the user can be implemented (this has only partly been carried out in the present version). Slight modifications in the syntactic description of CONSTRUCTOR might be sufficient for applications in syntactically related domains.
— a considerable amount of attribute evaluation can be standardized. In cases where linguistic structure shows significant variation (e.g. the structure of objects), the metalevel description can be used for object definition. This description is the basis of the procedures that handle the object table. There are several parts of the specification which are suitable for generaliza-

tion, but others are problem-specific. Here, again, the metalevel description can provide a possible way-out by defining clues for establishing relations between objects.

— although the implementation of relations depends on the very application, it seems probable that a natural language interface connected to some CAD or data-base would have much in common with CONSTRUCTOR,

— at present we cannot give a positive answer as to whether the actions invoked by CONSTRUCTOR could be straightforwardly transferred to some other Natural Language Interface, if at all, but a deeper insight in the semantic configuration of the class of verbs might lead to some result in the future.

4. Further research

Our further research in the area of Natural Language Interface generation will mainly be oriented to developing a generator that generates Natural Language Interfaces in a unique framework (now PROF—LP and metalevel object descriptions are separate entities). Another field of interest would be developing further methods for generalization.

References

[Arz85] Arz, J.: TRICON Ein System für geometrische Konstruktionen mit natürlichsprachlicher Eingabe, Technische Bericht, Universität des Saarlandes, Saarbrücken, KI—LABOR.

[Cliff88] Clifford, J.: Natural Language Querying of Historical Databases, Computational Linguistics 1988, 14 (4), 10—34 pp.

[Gyim88] Gyimóthy, T., Horváth, T., Kocsis, F., Toczki, J.: Incremental algorithms in PROF—LP, Lecture Notes in Computer Science Vol. 371, 93—102 pp.

[JAKE88] JAKE The application-independent natural language user interface, English Knowledge Systems, Inc. Scotts Valley, California, 1988.

[Mart83] Martin, P., Appelt, D., Pereira, F.: Transportability and Generality in a Natural Language Interface System, Proceedings of IJCAI—83. Vol. 1, 573—581 pp.

[Schr88] Schröder, M.: Evaluating User Utterances in Natural Language Interfaces to Databases, Computers and Artificial Intelligence 1988 (7), 317—337 pp.

Data Pictures on the Desktop*

ÁGNES HERNÁDI, ALADÁR HEPPES and ELŐD KNUTH

Computer and Automation Institute
Hungarian Academy of Sciences
Kende u. 13—17., H-1052 Budapest, Hungary

Abstract

An overview of the coDB database management system is presented, focusing on the system substratum and the facilities provided to build and manage data contents called Pictures. The experimentally implemented system provides an unusual database interface which makes multi-contextual data dialogues intersession-resident and responsive to appropriate changes of the database. Contextual access to the data provides an important degree of functional separation. The information base splits into two parts: the database and the so-termed Gallery, the repository of Pictures.

1. Introduction

Single user graphical workstations which provide a multiwindow environment are becoming more and more common. Whereas a plenty of application programs are offered which deal skilfully with data like Excel [1], Cardfile (SAPANA) or even HyperCard [2], none of these pursue real database functions.

The idea is appealing: a kind of visual "data object editor" having the power for performing all database functions formerly associated with entry forms, query specifications and pieces of the schema separately. The appeal is to the inexperienced end-user who manipulates heterogenous, ill-structured data and not to the professional database person.

The challenge facing us, then, is to provide a highly flexible user friendly man-machine interface facility which not only allows for concealing the traditional concepts of schemas, data definition languages and data manipulation languages but also provides a single unifying tool serving simultaneously various purposes for entering and updating data, query interpretation, report generation and schema manipulation.

* Lecture presented at the 1st Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

A database interface inspired by office-, management- and personal information systems recently coming into view (such as Hyper-systems, the remarkable phenomenon of Macintosh etc. [2], [3], [4]) has been experimentally developed at our institute for AT & T UNIX PCs. This interface facility [5], [6], [7], [8], [9,] [10], [11] supports simultaneous usage of multiple views or data contexts, moreover makes these contexts intersession resident, sensitive to the current alteration of the database and is the only tool for accessing the database. Last but not least this facility is easy to understand, to learn and its putting to use is quick, requiring no programming knowledge.

2. On the Data Model of the Experimental System

We have developed an experimental system called cooperative Databases (co DB) [12]. This relies on an ultimately simple data scheme. In order to allow us to keep our attention intently fixed on the problems of that interface facility we have chosen a completely unsophisticated and practicable data model. Namely

- We apply a binary relationship model [13] with no subtyping, however, relationships are non-directed many-to-many.
- Types and relationships are maintained automatically by entering their instantiation, and destroyed utterly on deleting their last instantiation. Instances of types are called atoms, and that of relationships are called connections.
- Two constituents are put together to form an atom that is to say a ⟨type⟩ class-description appointing the type to which the atom belongs and a ⟨value⟩:

  ⟨type⟩⟨value⟩.

  Both constituents are character strings although implementations may have restrictions laid on them. There are no arithmetical operations interpreted on values and for the time being we don't even plan to introduce them.
- A connection is an unordered pair of atoms belonging to a particular relation, so three constituents are put together to form a connection like a **⟨relname⟩** and an unordered pair (**⟨type 1⟩⟨value 1⟩**, **⟨type 2⟩⟨value 2⟩**). So it seems a relation is made up of a **⟨relname⟩** taking the place of role and an unordered pair of existing types (**⟨type 1⟩**, **⟨type 2⟩**).

  Relation names are — they may even be empty ones — character string objects which are unique for any given pair of types. Two binary relationships are considered identical if and only if all three of their corresponding constituents are identical (disregarding the order of types).

  A type might as well be related to itself, and an atom might be a constituent of any number of connections within a given relation. Two connections are considered to be identical if and only if all three of their corresponding constituents are identical (neglecting the order of atoms). Accordingly the same pair of atoms might be connected in as many relations as one could desire and still appearing once at most in a given relationship.

3. The Way of Displaying Data and Context

We believe it helps to view a binary relationship as a paragraph of two lines marked by the indentation of the second one.

Accordingly the first line displays an atom or a type and the second one the related atom or type preceded of course by the relation name if it is not actually an empty string. So a relation ⟨**relname**⟩(⟨**type1**⟩, ⟨**type2**⟩) or a connection ⟨**relname**⟩(⟨**type1**⟩⟨**value1**⟩, ⟨**type2**⟩⟨**value2**⟩) may appear in either form shown by Figure 1.

```
(i)    type 1:
         [relname] type 2:
    or
       type 2:
         [relname] type 1:

(ii)   type 1: value 1
         [relname] type 2: value 2
    or
       type 2: value 2
         [relname] type 1: value 1
```

Figure 1. Equivalent ways of displaying (i) a relation and (ii) a connection

As an example let us explore a database that records the main features of the twelve animal categories in the ancient Chinese lunar calendar. To represent this we select three types such as "*category*", "*in_the_cycle*" and "*year*", and two sorts of relations *R*1 for the relationship between "*category*", and "*in_the_cycle*", and *R*2 for the relationship between "*category*", and "*year*". Information about the Goat should appear in one of those forms in Figure 2, depending on our taste or purpose. We will see later, that any number of indentations are allowed.

Notice that no matter how we name *R*1 and *R*2 they remain redundant and even disturb us in understanding the represented connections. If both relation names were empty strings it would be quite similar to the traditional way of jotting. That's the reason why we allow relation names to be empty strings.

```
(i)    category: the Goat
         [R1] in_the_cycle: 8th
         [R2] year: 1907

(ii)   category: the Goat
         [R2] year: 1907
         [R1] in_the_cycle: 8th

(iii)  in_the_cycle: 8th
         [R1] category: the Goat
           [R2] year: 1907

(iv)   year: 1907
         [R2] category: the Goat
           [R1] in_the_cycle: 8 th
```

Figure 2. Alternative reflections of the same information

4. Pictures as the unified tools of interaction

A *picture* consists of an arbitrary number of hierarchically indented lines displaying atoms, types and relation names occasionally. More formally a picture is a forest of *picture lines* with the definition of

⟨picture line⟩ : = [⟨relname⟩:] ⟨type⟩: ⟨domain⟩
⟨domain⟩ : = BLANK | ⟨value⟩ | ⟨expression⟩

where **⟨expression⟩** is a selection criteria (e.g. a regular expression in terms of UNIX), and **BLANK** stands for *any* or *all* (no selection criteria).

In root lines no **⟨relname⟩** may appear. In non-root lines however theoretically a **⟨relname⟩** always appears at the very most it is empty (theoretically present but invisible). In a manner consistent with the above definition each picture line has a unique parent line unless it is a root line.

4.1. Validity and other characteristic properties of pictures

Informally speaking a picture is called *valid* if all the types, atoms, relations and connections referred to by any of its lines exist.

A valid picture is *filled* if each indented line in it possesses the following property: if its parent line contains a value, then it enumerates all the values connected to the atom displayed in its parent line by the named relationship.

A valid picture is *saturated* if each line in it which has at least one indented line, also has all possible indented lines in this very picture.

An empty picture trivially possesses all of these characteristic states.

5. Operations on Pictures

Any kind of user action can be carried out by using the appropriate operations.

5.1. Property Enforcing Transformations

To enforce picture properties and to carry out report generation we provide four transformations each of which acts on the whole picture.

VALIDATE

All the types, atoms, relations and connections appearing in the picture spring into existence if they have not existed in the database (see definition of valid picture).

FILL

The picture is to be converted into a filled one (see definition of filled picture). All the values fitting into a given place will be listed. In case of domain expression only values satisfying the expression will be included.

SATURATE

The picture is to be completed to become saturated (see definition of saturated picture).

EVALUATE

All feasible valid sequences satisfying every single domain specifications are to be determined.

Each value displayed on the picture is regarded as a restriction. Feasible pathes between two lines displaying atoms must fit on both ends. Reports are typically generated by this operation.

5.2. Operation modes

In order to reduce the number of *depicting* operations operation modes were introduced. These serve as distinctive marks of the particular classes of effects. There are three operation modes:

FREE	mode serves for temporal depicting with no effect on the database.
CHECK	mode is the default mode for all valid pictures. This mode serves the purpose to develop our view on data. Therefore in this mode no operation has update effect and operations violating the validity constraint are refused.
ENFORCE	mode provides the only way to alter the database. In this mode the VALIDATE transformation is called after each depicting operation the result of which violates the validity constraint.

The FREE→CHECK mode transition is refused whenever the picture is not valid. Other mode transitions are never refused. The FREE→ENFORCE mode transition is equivalent to a call of the picture state transformation VALIDATE, and as such must be confirmed.

5.3. Depicting Operations

These operations act on the selected part, that is on a subtree, a line or a token of a picture. They may or may not change the database itself depending on the current operation mode, however, according to the *What You See Is What You Get* paradigm no invisible change may occur. The whole interaction is supported with forms and icons requiring no syntactic knowledge of the user.

On selecting a subtree, we speak about a weak subtree if no restriction applies to the selection. But if the remainder must still constitute a picture, we speak about a strong subtree, see Figure 3.

```
category: the Ox
    in_the_cycle: 2nd
    year: 1913
    [best_marriage_partners] category: the Snake
        year: 1905
    [should_avoid] category: the Monkey
        in_the_cycle: 9th
        [best_marriage_partnars] category: the Rat
            in_the_cycle: 1st                              (i)          (ii)
            [should_avoid] category: the Horse
                year: 1906
        year: 1908
    business: the are good in business being very shrewd and
              completely reliable
```

Figure 3. Example of (i) a weak and (ii) a strong subtree

REMOVE (strong subtree) (option)

The selected subtree disappears recursively from the picture. In the ENFORCE mode data objects are to be deleted too.

Option	Actions taken
with root (default)	The whole selected subtree disappears. In the ENFORCE mode each atom included in the selected subtree is to be deleted with all their connections. Corresponding types and relations can be destroyed utterly. No other inductive effect is taken.
without root	If differs from the default option in that the root line of the subtree does not disappear. In the ENFORCE mode each atom displayed by the root lines of the disappearing part is to be disconnected from the atom displayed by the root line of the selected subtree. The corresponding relation can be destroyed utterly.
disconnect	The whole selected subtree disappears. In the ENFORCE mode the atom displayed by the root line of the selected subtree is to be disconnected from the atom in its visible parent line, if there is any. The corresponding relation can be destroyed utterly.

CLEAR (weak subtree)

All the domains in the selected subtree are to be made blank. Identical lines with no indented hierarchy are only to be displayed once. It never alters the database.

MOVE (strong subtree)

The selected subtree is to be moved. The subtree disappears from the source picture. This operation can be used in an inter-picture sense too. See PASTE for terminating a MOVE.

COPY (weak subtree)

The selected subtree is to be copied. The source picture remains unchanged. This operation can be used in an inter-picture sense too. See PASTE for terminating a COPY.

PASTE (line)(option)

This operation terminates a COPY or MOVE operation. The subtree to be copied or moved is inserted into the picture according to the option specified. In the ENFORCE mode database update requires confirmation.

Option	Actions taken
after (default)	The copied/moved subtree is to be inserted after the selected line (skipping of course all the lines marked by a longer indentation). The root line of this subtree is to be marked by the same indentation as the selected line.
before	The copied/moved subtree is to be inserted prior to the selected line, and its root line is to be marked by the same indentation as the selected line.
under	The copied/moved subtree is to be inserted immediately after the selected line. Its root line is to be marked by an indentation as compared to the selected line.

ADD LINE (line) (option)

A line is to be created and inserted into the picture according to the option specified. In the ENFORCE mode database update may occur.

Strings to be displayed in the inserted line should be entered through a form asking for them. Menus of tokens already entered into the database are available.

Option	Actions taken
after (default)	The created line marked by the same indentation as the selected one is to be inserted after the selected line (skipping of course all the lines marked by a longer indentation).
before	The created line marked by the same indentation as the selected one is to be inserted ahead of the selected line.
under	The created line marked by an indentation as compared to the selected one is to be inserted immediately after the selected line.

Suppose we have a picture including the portion of Figure 4. If we want to enter some other information about the Goat let us say the nature of those born in the Year of the Goat right after the line displaying it, then we have to select (mark) either the line displaying the name of this category and to ADD LINE under it, or the line displaying its serial number in the cycle of Twelve and to ADD LINE before it, see Figure 5.

```
...
category: the Goat
    in_the_cycle: 8th
    year: 1907
...
```

Figure 4. A portion of a picture

If we want to display another category let's say the Pig right after the information about the Goat, we can select the line displaying the name of this latter category and ADD LINE after it, see Figure 5.

```
...
category: the Goat
    nature: generous and shy, blessed with many
        virtues and just as many failings
    in_the_cycle: 8th
    year: 1807
category: the Pig
...
```

Figure 5. Result of the two ADD LINE operations on the picture in Figure 4

UNFOLD (line) (option 1, option 2)

Our view of information displayed in the selected line is to be opened out by revealing all the lines which could come up as indented ones according to the database's content. It never alters the database.

Option 1	Actions taken
natural only (default)	If the selected line displays an atom then all the atoms connected to it should be displayed. If however the selected line does not display an atom, all the relations interpreted on the type in the selected line should be displayed.
full	All the relations interpreted on the type in the selected line are to be involved neglecting whether they have any connections referring to the atom in the selected line.

Option 2	Actions taken
non-repeating (default)	The parent line of the selected one is not to be displayed between the indented lines.
repeat parent	Even the connection or relation between the selected line and its parent line is to be involved.

For the indented lines of the selected one which are already in the picture the following rules apply:

- Already existing lines will not be repeated;
- If the selected line displays an atom and the already existing indented one displays a blank domain then this blank domain will be filled with appropriate values instead of repeating this latter line;

- If the indented line contains a domain expression, it remains unchanged, and an other indented line will be inserted with the same relation and type names displaying values or not depending on the selected line.

```
...
category: the Goat
     nature: generous and shy, blessed with many
          virtues and just as many failings
     in_the_cycle: 8th
     year: 1907
category: the Pig
     in_the_cycle: 12th
     [should_avoid] category: the Snake
     [best_marriage_partners] category: the Rabbit
     year: 1911
           1923
           1935
...
```

Figure 6. After unfolding the line which displays the category "Pig" (with default options)

ADD VALUE (domain) (option)

A new atom of the type specified in the line of the selected domain is to be inserted. If the domain did not contain a value no option is offered. The required value will be inserted in that very domain, however, it must not contradict the selection criteria, if there is any specified.

If the selected domain already contains a value, the new value is to be inserted according to the option specified.

In the ENFORCE mode database update may occur.

The value to be displayed in the selected domain should be entered through a form asking for it. The menu of values already entered into the database are available.

Option	Actions taken
after (default)	The line containing the new atom will be placed right after the indented hierarchy of the line displaying the selected domain. The skeleton of the indented hierarchy of the selected line if there is such a hierarchy at all, will be inserted after the line containing the new atom. This skeleton contains all relation and type names but domains remain empty.
before	The line containing the new atom will be inserted above the line displaying the selected domain.

DELETE_VALUE (domain)

The selected domain must contain a value which is to be abandoned. In the ENFORCE mode the atom is to be deleted with all its connections. No type or relation can be destroyed, however.

EDIT EXPRESSION (domain)

A selection criteria is to be defined or modified. Editing the selected domain is refused if it contains a value (see EDIT TOKEN). No value may be specified (see ADD VALUE).

```
...
category: the Goat
    nature: generous and shy, blessed with many
        virtues and just as many failings
    in_the_cycle: 8th
    year: 1907
category: the Rooster
    nature:
    in_the_cycle:
    year:
category: the Pig
    in_the_cycle: 12th
    [should_avoid] category: the Snake
    [best_marriage_partners] category: the Rabbit
    year: 1911
          1923
          1935
...
```

Figure 7. Result of the ADD VALUE operation with an argument displaying the category "Goat" and the option after

EDIT TOKEN (token)

One of the strings displayed in the selected line is to be altered either in the picture containing the selected token (FREE mode), or in the database, and in all pictures displaying this very string (ENFORCE mode). This operation is unavailable in the CHECK mode.

Respectively either the relation or the type is to be renamed or the value is to be changed keeping all the connections. Editing the selected domain is refused if it is either empty (see ADD VALUE) or contains an expression (see EDIT EXPRESSION).

6. Galleries

Our pictures are stored in a special directory called Gallery which supports transactions dealing with pictures.

6.1. Picture Qualification

Pictures stored in a Gallery are qualified but their quality can be altered any time.

- A *Sketch* is a picture which need not be valid. Pictures to be depicted in FREE mode, or having become invalid are always requalified to this quality. This is the quality of created pictures too.

- A *Composition* is a valid picture which however can become invalid and requalified to Sketch as the database changes.
- A *Protected* picture may never turn invalid. Depicting operations in ENFORCE mode on any picture violating this restriction are refused.
- A *Master Piece* moreover may not be changed at all. Depicting operations in ENFORCE mode on any picture violating this restriction are refused.

Beside these there are two *standard,* read-only pictures in each Gallery. The picture *Types* contains all the existing types. The picture *Scheme* contains all the existing relationships. These pictures or any of their parts can be copied freely however.

6.2. Gallery Organization

A Gallery consists of two main parts

- the *Exhibition* in which all pictures are updated according to the EDIT TOKEN operations and checked up on being valid or not; and
- the *Archive* in which pictures are not maintained at all.

6.3. Transactions dealing with pictures

Each Gallery belongs to a single co DB. On opening the Gallery its Exhibition-menu is displayed. At request the Archive-menu is displayed too but in a separate window. From these menus the user can access the picture transactions namely:

- Create Picture
- Open Picture
- Delete Picture
- Copy/Move Picture
- Rename Picture
- Requalify Picture

All pictures have to be created except the standard ones. Opening a picture the operations on pictures are available. From an opened Gallery any number of pictures can be opened simultaneously. The other transactions work roughly in a way as can be expected.

7. Implementation issues

As we have already mentioned, co DB is experimentally developed for AT & T UNIX PCs. The implementation exploits

- the hierarchic file system of UNIX;
- the multiple window management capability supported by TAM routines; and
- the manipulation of abstract objects at the operation system's level provided by UA.

We manipulate two abstract objects at the operation system's level: the *co DB* database and the *Gallery.*

Commands assumed to be applicable to all ordinary abstract objects of this level such as create, open, close, delete, move, copy, rename are also defined on both of these objects.

Apart from the fact that each Gallery refers to a single co DB, any number of Galleries can be associated with a co DB. On opening a co DB the menu of Galleries associated with it is displayed.

8. Summary

We have expounded a new, non language oriented approach of interactive database interface in contrast to [15], [16], [17], [18] etc. This approach by matching modern requirements gives naive users demanding database supply altering dynamically an easy-to-use tool to interact directly with database.

We introduced the concept of picture which are user friendly abstract objects, having no fixed structure. Their content is transient, and they serve as a unified tool for accessing the database.

Our approach also provides a consolidated mechanism to draw computer aided comparison between independent databases containing diverse data, and to settle database communication protocols aiding interaction between them.

References

[1] Jones, E., Using Excel TM for the PC, (Osborne McGraw-Hill, Berkeley, California, 1988).
[2] Macintosh, HyperCard User's Giude (Apple Computer, Inc., 1988).
[3] Christie, B. (ed.), Human factors of the user-system interface, (North-Holland, Amsterdam, 1985).
[4] Shu, N. C., Visual Programming, (Van Nostrand Reinhold Company, New York, 1988).
[5] Hernádi, Á., Bodó, Z., Knuth, E., A different interactive interface for database management systems, Proceedings of the 11th Int'l Seminar on Database Management Systems, Seregelyes, Hungary (Oct. 3—7, 1988), pp. 85—94.
[6] Knuth, E., Vaina, A. M., Bodó, Z., Hernádi, Á., Beyond data crunching: A new approach to database interaction, Proceedings of the 3rd Austrian—Hungarian Informatics Conference on "Beyond number crunching", Retzhof, Austria (Sept. 14—16, 1988), pp. 91—104.
[7] Bodó, Z., Hernádi, Á., Knuth, E., Adatok mint "kepek", Proceedings of the 4th National Congress of the John von Neumann Society for Computing Sciences on "Application '89", Pecs, Hungary (March 28—Apr. 1, 1989), Vol. II, pp. 308—318 (in Hungarian).
[8] Hernádi, Á., Bodó, Z., Knuth, E., Context-reflecting pictures of a database, Proceedings of the EUUG Spring '89 Conference "UNIX: European Challenges", Brussels, Belgium (Apr. 3—7, 1989), pp. 273—282.
[9] Knuth, E., Hernádi, Á., Bodó, Z., Pictures at a Data Exhibition, in: Chang, S. K., (ed.), Visual Languages and Visual Programming (Plenum Publishing Company, in print).
[10] Knuth, E., Hernádi, Á., Bodó, Z., Gyenese, J., Data Gallery, Proceedings of the World Conference on Information Processing and Communication, Seoul, Korea (WOCON—INFOR 89, June 13—16, 1989).
[11] Knuth, E., Hernádi, Á., Heppes, A., Beech, D., Visual database management, Proceedings of the 4th IFIP Working Conference on User Interfaces, Napa Valley Lodge in Yountville, CA (Aug. 21—25, 1989, NORTH HOLLAND, in print.
[12] Bodo, Z. et al., Data Gallery — Common Base Definition, Reference manual Version 9. (Computer & Automation Institute, Hungarian Academy of Sciences, Budapest, December 1988.)
[13] Bracchi, G., Paolini, P. and Pelagatti, G., Binary Logical Associations in Data Modelling, in: Nijssen, G. M., (ed.), Modelling in Database Management Systems (North-Holland, Amsterdam, 1976).
[14] Vermeir, D. and Nijssen, G. M., A Procedure to Define the Object Type Structure of Conceptual Schema, Information Systems, Vol. 7 No. 4 (1982) pp. 329—336.
[15] Lans, R. F. van der, Introduction to SQL (Addison—Wesley, 1988).
[16] Hartman, P. A., R: BASE System V and 5000, (TAB Books Inc., USA, 1988).
[17] dBASE III User Manual (Copyright Ashton—Tate, 1984).
[18] INFORMIX (Registered trademark of Relational Database Systems Inc., USA).

YYY-A database design tool*

HARRI LAINE

University of Helsinki, Department of computer science
Teollisuuskatu 23, SF—00510 Helsinki, Finland

Abstract

YYY is an interactive, graphical tool set for designing databases. Presently it contains tools for the design of the enterprise (conceptual) schema and an expert system for generating the schema for the relational database. On the enterprise level the data model supported in YYY is a variant of the Entity — Relationship Model. A relational database schema is represented as SQL 'create table' — statements. This paper discusses the overall structure of the tool set, the data dictionary, the principles of the user interface and the rules that control the generation of the relational database schema.

1. Introduction

During its design the database is described in various levels of abstraction. Most researchers identify three levels. We call these levels the *enterprise level,* the *representation level* and the *internal level.* On the enterprise level (often called conceptual level [ElNa89]) the main concern is on *what* is described in the database, i.e. on how the object of the data is structured. On the representation level (called also logical level or implementation level [TeFr82, ElNa89]) we are concerned about the *logical data structures* that are used in representing the data. On the internal level the technical structures and access paths of the database are considered. As related to each level of abstraction, there is a schema that contains the description. Database design is a process of constructing a schema for each level of abstraction. A natural course of action is to proceed from the user oriented enterprise schema to the computer oriented internal schema. The intermediate representation schema is needed because most of the current database management systems rely on it.

Database design process is often divided into four steps: *requirements collection and analysis, conceptual design, data model mapping* (logical design) and *physical*

* Lecture presented at the 1st Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

design [TeFr82, ElNa89]. During the first step the needs of the users as related to the contents and the use of the database are collected and written down. In conceptual design these needs are analyzed and the *enterprise schema* is constructed to reflect the needs about the contents of the database. Thus, the enterprise schema should be considered as a part of the documentation of the user requirements. Especially, when the users take active part in the design, the two first steps of the design proceed simultaneously.

Conceptual design is an iterating process. Many preliminary versions of the enterprise schema are usually constructed, evaluated and modified before a satisfactory result is concluded. The schema should be made available in various forms for inspection and evaluation. This is a proper place for a computer assisted tool. Actually, some tools, such as ER—Modeler, IEW, Excelerator, to support the conceptual design have been developed, within the last few years [Xeph88]. These tools differ from each by their technical environment, their functionality, and by the data models that they support. Tools to assist the conceptual design are the kernel components of the YYY database design tool. These tools will be described ir section 3.

The next design step, data model mapping, produces the schema for the database management system. If the database management system would use the same data model that is used in representing the enterprise schema this step would not be needed. However, database management systems that are based on semantic data models are not yet common in practice [HuKi87]. Thus, a mapping is needed. The data model, on which the enterprise schema is mapped, is presently increasingly the relational model of data [Codd70]. YYY tool set contains a *mapping tool* that produces relational database schemata. The mapping tool is discussed in section 4.

Section 2 describes YYY database design tool as a whole, and section 5 outlines the further development of the tool set.

2. YYY tool set

YYY database design tool set is the present phase in a series of experimental database design tools constructed during the last ten years at the department of computer science in the University of Helsinki. We started with a semantic data model [LaMP79], and developed a data definition language (HULDA) based on that data model [Lain81]. A compiler to produce the data dictionary representation of HULDA schemata was constructed. Connection matrices and diagram representations could be obtained via a plotter. The tools were implemented in a maim frame (Burroughs) environment. They were experimented in a few database design projects. The conclusions of the experiments are reported in [Lain86]. The data definition language representation of the enterprise schema was found to be good as a detailed document for the adp-professionals. It was not considered good as a document for the end users, nor as an input medium. The further development of the main frame tools ceased when our main frame computer was changed.

The user interface and the end users were of main concern when the next generation of our database design tools were developed. Database design tools that are intended to the end users must be based on their native language. There are many examples of database design projects in Finland, where the end users have not accepted the documents that contain both English and Finnish [Kall89]. Due to this, we decided to use Finnish language as the only language in our tools.

We used fast prototyping as our development technique. Our starting point was to develop an interactive graphics editor that works in IBM compatible micro computer and could be used for constructing both the diagram and the data dictionary representation of the enterprise schema. The first version of the schema editor was programmed in Basic, because it was the only high level programming language which, at that time, provided the necessary graphic primitives for our Olivetti high density graphics. The prototype proved out to be quite handy in education and in constructing small schemata. For the next version, three alternatives were considered for the implementation tool: Windows graphics environment, Turbo Pascal and compiled Turbo Basic. Because of limited resources, the last alternative was selected. This version has been in educational use for over a year and works well also with large schemata.

One of our goals was to develop a tool that is able to work with 640K of memory. All extensions to the schema editor reduce the maximum size of the schemata that can be processed. To facilitate large schemata we split the functions of the conceptual design tool into three separate tools: *the schema editor, the dynamics editor,* and the *output facility.* It was decided that the further development of the tools should be carried out mainly in Turbo Pascal. Pascal versions of the dynamics editor and the output facility are ready and in use. The Pascal version of the schema editor (version 3) is still under development.

In addition to the tools for the conceptual design, YYY tool set contains a data model mapping tool to produce the relational database schema. This tool is implemented in Turbo Pascal and in Prolog. All the tools can be started from a start up menu. The contents of the present YYY tool set is depicted in Fig. 2.1.

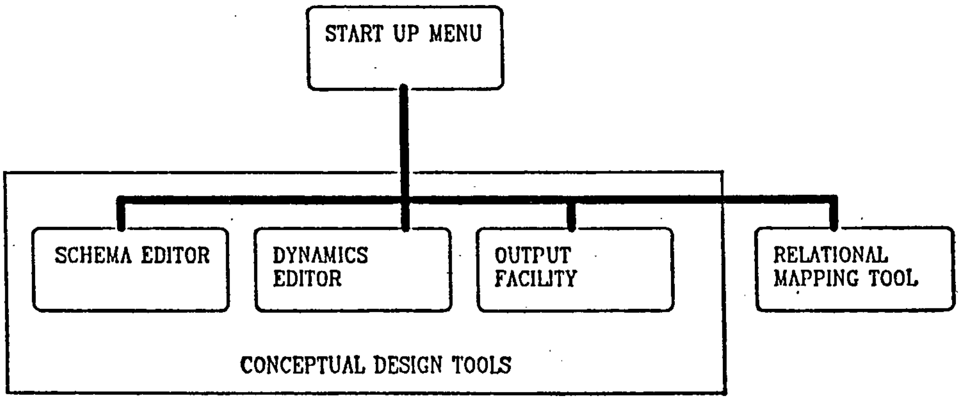


Figure 2.1. Components of the YYY tool set

2.1. Data dictionary

YYY design tools store an enterprise schema as four text files: a file for the static structures (⟨schema⟩.DDA), a file for the definitions related to the static structures (⟨schema⟩.DDB), a file for the description of the dynamics (⟨schema⟩.DDC) and a file for the definitions related to the dynamics (⟨schema⟩.DDD). The schema files are stored as Prolog like fact files. The structure of the facts is shown in Fig. 2.2. The definition files contain multiple definition text blocks. Each text block starts with a header line, that identifies the object to be defined. This is followed by at most 200 lines of definition text. The text block ends with a footer line. Definition files can be edited with any text editor.

For the processing the whole schema is loaded in memory. Dynamic data structures with multiple indexes are used. The definition files are transformed into indexed files (version 3 of the editor). A definition is loaded into memory only when needed. At the end of the session the indexed definition files are transformed back to text files.

Entity statement:

yksilo(
⟨entity type name⟩, ⟨x-coordinate⟩, ⟨y-coordinate⟩,
⟨position in hierarchy⟩, ⟨population size⟩,
⟨schema version identifier⟩).

Property statement:

ominaisuus(
⟨entity or event type name⟩:⟨property type name⟩/⟨value type⟩,
⟨functionality⟩, ⟨necessity⟩, ⟨identifier?⟩,
⟨changeability⟩, ⟨schema version identifier⟩).

Relationship statement:

yhteys(
⟨relationship type name⟩,
⟨role name⟩:⟨entity type name⟩
(⟨minimum restriction⟩, ⟨maximum restriction⟩),
⟨role name⟩:⟨entity type name⟩
(⟨minimum restriction⟩, ⟨maximum restriction⟩),
⟨duplicate counter⟩,
⟨x-coordinate⟩, ⟨y-coordinate⟩,
⟨original duplicate counter⟩,
⟨schema version identifier⟩).

Hierarchy statement:

hierarkia(⟨entity type name⟩, ⟨entity type name⟩).

Version statement:

versio(⟨yyy version identifier⟩,
⟨schema version identifier⟩,
⟨drawing technique⟩).

Event statement:

tapahtuma(⟨event name⟩, ⟨frequency⟩, ⟨schema version identifier⟩).

Effect statement:

vaikutus(⟨event name⟩, ⟨affected object name⟩,
⟨affected property name⟩, ⟨type of effect⟩,
⟨schema version identifier⟩).

Figure 2.2. Fact structures in schema files

3. Tools for the conceptual design

3.1. The data model

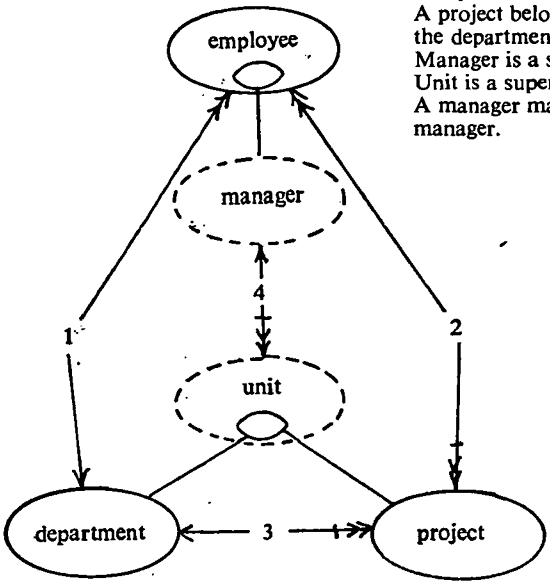
One of the main differences between the conceptual design tools is the data model supported by the tools. There are tools that support multiple data models, but only as far as the drawing technique is concerned. To fully support a data model, is to allow only such constructs that obey the constraints embedded in the model. These constraints vary between different models and also between the variations of the same base model [HuKi87]. The most commonly supported data model in the design tools is the Entity—Relationship Model, and its numerous variations [Xeph88, HuKi87]. The original Entity—Relationship model [Chen76] imposes very few restrictions on the constructs. Relationship types of any degree may be defined. Both entities and relationships may have attributes. Only entities may participate in relationships. On my experience it is hard to work with such a loose model. There are too many ways to model the same phenomena, and the model does not force to select any of them.

YYY supports an Entity—Relationship Model variant. The kernel of this variant is the simplified Entity-Relationship model variant recommended by the Finnish Standardization Association [SFS88, Sorv88]. YYY data model supplements this model with concepts needed in describing the dynamics of the object system and with constraints that restrict the generalization hierarchy. It divides the phenomena of the object system (mini world) into four categories: *entities, relationships, properties* and *events*. Relationships are restricted to the binary ones. Entities and events may have properties, but relationships can not have properties. Only entities may participate in relationships. Properties are represented with (attribute, value) -pairs. It is assumed that the database contains facts about the existence of entities, the existence of relationships and the existence of the properties of entities. Events are phenomena that affect entities, relationships and properties. They are not represented directly in the database.

The design task is to specify the types of entities, relationships, properties and events. The relations between the specified types must also be determined. In YYY a specification is more than just naming the type. All the types can be attached with textual definitions. Property types must be characterized as identifying or nonidentifying, fixed or varying, single-valued or multi-valued, and necessary or just nice to know. Application depended value types can be specified as related to the property types. A selection of about thirty pre-defined value types is provided. Participation restrictions (minimum and maximum) and the use in identification, are used in characterizing relationship types. A generalization hierarchy can be defined between entity types. Some of the entity types must be defined as *base types*. The others are either super types or subtypes.

The dynamics of the object system is modelled with event types. The kind of effect (creation, change, termination) characterizes the relations between event types and entity, relationship or property types.

Diagrams play an important role in conceptual design. They are the means of providing the overview of the design. YYY provides only entity level diagrams. These diagrams hide out the property types. The diagrams show only the static structures. There is a large variety of drawing techniques attached to Entity—Relationship models. It is fairly easy for a tool to provide multiple drawing techniques if their symbol sets have a lot of common symbols. YYY design tool provides four drawing techniques that are all rather well known in Finland. Chen's original drawing technique [Chen 76] is not included. Fig. 3.1. gives an overview of the default drawing technique in YYY.



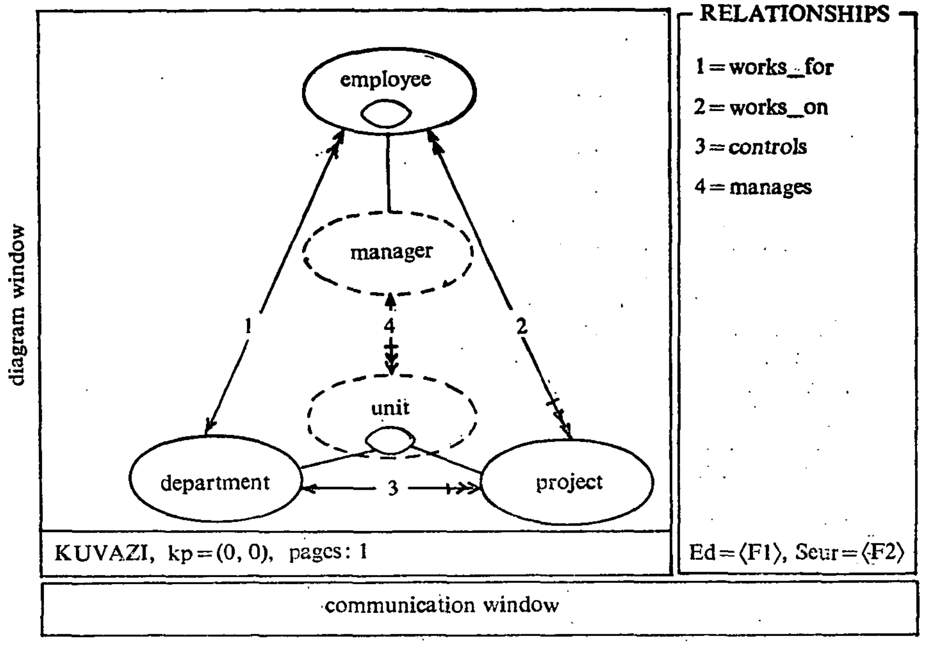
An employee works for exactly one department. A department has many employees, at least one.
A department controls many projects, not necessarily any.
A project belongs to one department and is identified via the department.
Manager is a subtype of employee.
Unit is a super type of department and project.
A manager manages multiple units. A unit has only one manager.

Figure 3.1. An example of an YYY diagram

3.2. The schema editor

The schema editor is used for defining the static structures of the object system, i.e. the entity, relationship and property types. It provides a graphical interactive user interface. The screen is split into three windows. The *diagram window* shows a part of the diagram. The *name window* shows the names that correspond to the short identifiers appearing in the diagram window. The *communication window* is used for user communication, entering specifications and selecting the functions from the menus. Fig. 3.2. shows an example of the schema editor screen.

The user communication is based on menus and on questions asked by the program. The main menu is explained in Table 3.1.



KuvatUlos — NäytönKuva
Tulostus kestää jonkin alkaa
Tulostus tiedostoon MK2. FIC Voit vaihtaa tiedostonimen alkuosan
— Anna uusi alkuosa tai ⟨Enter⟩ Jos hyväksyt oletusarvon:

Figure 3.2. An example of the schema editor screen

Table 3.1. The main menu of the schema editor

Entity	Operations on entity types: specify a new type, remove an existing type, change or view the specification, re-position the entity type symbol in the diagram, integrate two types.
Relationship	Operations on relationship types: specify a new type, remove an existing type, change or view the specification, change the participant, re-position the center point of the connection line.
Hierarchy	Operations on generalization hierarchies: specify a new subset relation, remove a subset relation.
Diagram	Operations on the diagram: re-position entity type symbols, zoom, re-position the diagram window, change the drawing technique, insert/remove page limit lines, expand the diagram window over the whole screen.
Diagram Output	Operations to obtain sketch quality output: Select the output device, print the screen image, print the expanded diagram window image, print entity names, print relationship names.
Exit	The construction of the data dictionary files. Exit the program, if wanted.

To make the use of the tools easier some of the menu alternatives activate a bunch of actions. For example, the selection 'specify new entity type' activates the following actions:

```
repeat {
    Name the entity type,
    Specify the population size, position in hierarchy and textual definition,
    Position the entity type symbol in the diagram,
    repeat {
        Identify an entity type to which the new one is connected,
        Specify the relationship
    } until empty name is provided,
    if the new type is 'super type' then
        repeat {identity the type that is 'lower' in hierarchy
        } until empty name is provided
    else if the new type is 'sub-type' then
        identify the type that is 'upper' in hierarchy,
    when 'wanted'
    repeat {Specify a property type
    } until 'no more property types'
} until empty name is provided.
```

YYY schema editor requires the user to position the entity symbols in the diagram. The connection lines that represent the relationships are drawn automatically, but may be edited later on. Manual positioning of symbols requires more user effort than automatic positioning. An advantage of manual positioning is that the user has the control over the diagram, a small change does not re-organize the hole diagram. YYY schema editor offers a default answer for most of its questions. This makes the construction of the enterprise schema easy. As related to each modification of the schema the editor makes some integrity tests. These tests are not, however, complete. The new version of the editor will perform a more complete integrity and quality test each time the schema is written in the data dictionary.

YYY schema editor has been used in database design education in the University of Helsinki for three years. It has had several hundreds of users. It has proven to be easy to learn and quite useful.

3.3. The output facility

High quality paper documents of the enterprise schema are very important in database design. These documents are used as working papers within the design project. In addition, they are the means to distribute information about the design to the users and to the environment. One requirement for the documents is that they should be obtained quickly. The printing time of about ten minutes per page, which is quite common in many design tools, is just too much.

YYY produces all its paper output via auxiliary files. The schema editor can produce output files of the diagram on screen image basis. The files are produced for IBM Graphics Printer/Epson graphics compatible matrix printers. It takes less than a minute to produce an output file and about a minute to print the file with a matrix printer.

YYY tool set contains a separate output facility for the production of high quality paper documents. Diagrams are coded in PostScript. The output facility provides many alternatives to determine how the diagram shall be split for output.

These include: the whole diagram into one A4-sheet (recommended if the diagram is not very large, example as Appendix 1), one or two diagram pages in one A4-sheet and user specified area per one A4-sheet. Diagram output ca be obtained in any of the four drawing techniques supported by YYY.

Text documents represent the data dictionary information as a formatted report. Users may control the amount of definition texts in the report (without definition, with entity and relationship level definitions only, and with exhaustive definition texts). The document can be restricted to cover only those objects that have changed since a certain version of the schema. Fig. 3.3 shows an example of a textual report.

```
SCHEMA: EXAMPLE                         04. 0.8. 1989    15:46:06          page: 2

Entity type: employee                   — base type,     population about: 100

Property              Value type                         Characteristics
-----------------------------------------------------------------------------------
ssn                   sos—sec—no                         identifier
fname                 name                               necessary
lname                 name                               necessary
sex                   sex                                necessary fixed
bdate                 date                               necessary fixed
address               address                            necessary
salary                money                              necessary
-----------------------------------------------------------------------------------------------------
Relationship type     Role            Holder                               Restr.
works-on              employee        employee                             0—n
                      project         project                              1—n
works-for             employee        employee                             1—1
                      department      department                           1—n

Hierarchy upper       Hierarchy lower
--------------------------------------------------------------
employee              manager
```

Figure 3.3. An example of a textual report *(translated into English)*

3.4. The dynamics editor

The dynamics editor can be used in describing the 'real world' events and their effects on entities, relationships and properties. No graphical symbol is used for an event type. In the diagram the affected objects are high-lighted with color. It is possible to define event types that affect multiple objects, both entity types and relationship types. When effects are defined the affected objects are selected from a pick list. The present version of the dynamics editor is the first prototype and it is not yet complete. Output facilities are still missing and the user interface should be improved. Our intention is to implement the textual reports and the effect matrices in the near future. We are also going to implement a cluster analysis program and apply it to the effect matrices to see whether it can assist the specification of the database update programs.

4. The mapping tool

YYY contains also an expert system for generating the relational database schema. The reulting schema is represented as a collection of SQL 'Create table'-statements. Four variations of SQL are supported ORACLE—SQL [Orac87], DB2—SQL [DaWh88], INGRES—SQL and a slightly extended ISO SQL-standard (proposed features of the forthcoming SQL2 standard are included) [ISO87]. The resulting schema specifies the structures of the relations, their primary keys and also their foreign keys (inclusion dependencies). When the SQL variation does not contain means to define the primary keys or the foreign keys, information about them is included as a comment.

The generation is carried out in three phases. The first phase pre-processes the enterprise schema and eliminates certain characters of the Scandinavian character set that cannot be used in SQL.

The second phase applies the transformation rules. It produces a file that specifies the relations, their attributes, their keys and their foreign keys. It also gives an explanation on what rules were used for each decision. The transformation can be done either in automatic or in interactive mode. In automatic mode default rules are used in certain problematic situations, such as, whether to represent a subtype (by default no) or a super type (by default yes) as a relation of its own, and whether to introduce artificial identifiers to replace large user keys that are widely used as foreign keys (by default no). In interactive mode the situation is explained to the user, a solution is proposed and acceptance or rejection is requested.

The third phase constructs the SQL statements. Application dependent value types are transformed into SQL data types. A conversion table is included for the pre-defined value types. Users may add new items in this table or replace it with a table of their own. The conversion table defines the conversion between the application dependent value sets and the SQL standard. Rules that are embedded in the program take care of adjusting the type definitions for the supported database management systems. If there is no conversion rule for some value type the corresponding data type will become 'undefined'. Column names may be problematic in the generation of the SQL statements. The SQL variations accept names with the length from 8 to 32 characters. The enterprise schema accepts property names of up to 24 characters and the second step of the transformation may generate foreign key names of up to 128 characters. No automatic cut off is provided. If the user works in interactive mode, it is possible for her to rename all the tables and columns. The names that are longer than what is allowed are indicated. YYY keeps track on the items that have been renamed. When the relational schema is re-generated the renaming made during the past generation runs can be taken into account, even in the automatic mode of the transformation.

Several rule sets to map an Entity—Relationship schema to a relational schema have been proposed e.g. [MaSh89, Chen76, ElNa89]. Our rule set consists of twenty rules. We aim to a minimal set of relations. Thus we accept null values, express the properties of a subtype in the relation attached to the corresponding base type, and represent all 1-to-n relationships with foreign keys. It is not allowed to specify identifying properties for sub- and super types. If a super type is represented as a relation of its own (the default) this relation will contain an artificial key, and an add-on attribute 'type' to indicate the base type. We have also the rule "If the user key of the relation is big and it should be used in many references then consider an artificial key".

To produce normalized relations, our mapping rules as all the other rules, presuppose that the Entity—Relationship Schema is normalized, i.e. redundancy is eliminated, properties are attached to correct entity types, and the relationship types are defined between the correct entity types. The analysis of the quality of the enterprise schema cannot be done automatically based on the information in the schema. But, it is possible to look for the potential problems and show them to the user to reconsider.

5. Future development

In the present system the definition texts must be written with a separate editor. This problem will be corrected already in the next version of the schema editor that will contain an embedded text editor. The tool set does not contain any tool for view integration. A simple integration tool that renames the objects and reorganizes the diagrams so that the schema editor can be used for integration has been designed. An add-on function that makes it possible to directly utilize data item lists as pick lists in specifying properties is planed, as well as the improved integrity and quality test for the enterprise schema.

References

[Chen76] CHEN P. P., The Entity-Relationship Model — Towards a Unified View of Data, ACM Trans. on Database Systems, 1, 1 (March 1976), pp. 9—36.

[Codd70] CODD E., A Relational Model for Large Shared Data Banks, Communications of the ACM, 13: 6, 1970.

[DaWh88] DATE C. and WHITE C., A guide to DB2, Second Edition, Addison-Wesley, 1988.

[ElNa89] ELMASRI R. and NAVATHE S. B., Fundamentals of Database Systems, Benjamin Cummings, 1989.

[HuKi87] HULL R. and KING R., Semantic database modeling: Survey, applications and research issues, Computing Surveys, 19, 3 (Sept. 1987), pp. 201—260.

[ISO87] ISO: ISO 9075, Information processing systems — Database language SQL + addendum 1, 1987.

[Kall89] KALLIALA E., IEW — Case software and SRM, Univ. of Helsinki, Dept. of Computer Science, Series C—1989—35, (in Finnish), 1989.

[Lain81] LAINE H., A Data Definition Language and its Use in Data Base design, Report of the Fourth Scandinavian Research Seminar on Systemeering, Univ. of Oulu, 1981, pp. 188—197.

[Lain86] LAINE H., User Interface in a Database design tool, Report of the Eighth Scandinavian Research Seminar on Systemeering, Aarhus University, Denmark, 1986, pp. 262—277.

[LaMP78] LAINE H., MAANAVILJA O., and PELTOLA E., Grammatical Data Base Model, Information Systems, 4:4, 1978.

[MmSh89] MARKOWITZ V. M. and SHOSHANI A., On the Correctness of representing Extended Entity-Relationship Structures in the Relational Model, Proc. ACM SIGMOND—89 Conf., 1989, pp. 430—439.

(Orac87] ORACLE CORP., SQL*Plus Reference Guide, 1987.

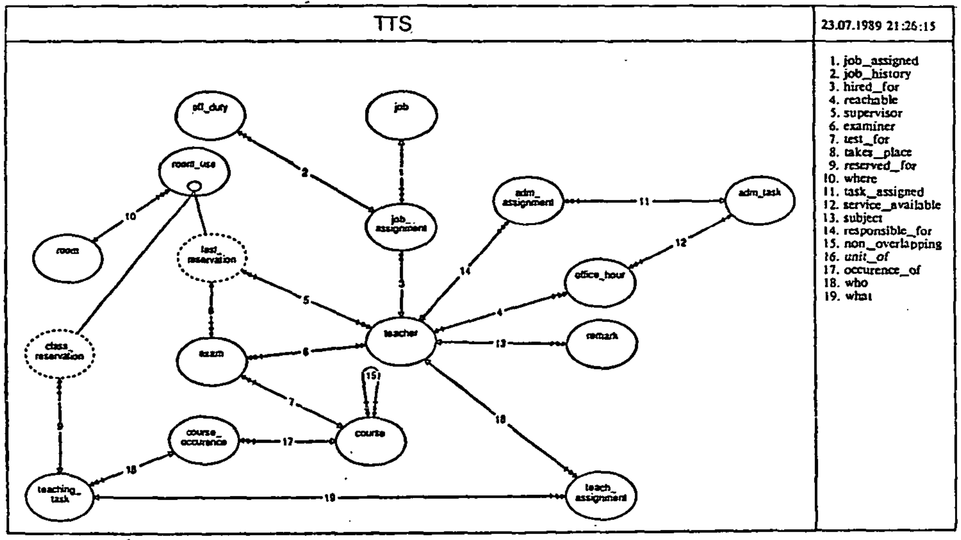
[SFS88] Finnish Standardization Association: Methods of Systems Development; ERTI—Conceptual Analysis, Handbook 106, 1988 (in Finnish).

[Sorv88] SORVARI J., ERTI — A method for Conceptual Analysis and Modeling of Information, Joint Finnish Soviet Software Symposium, Helsinki 15—18. 11. 1988, Proceedings to appear in 1989.

[TeFr82] TEOREY T. and FRY J., Design of Database structures, Prentice Hall, 1982.

[Xeph88] Xephon Inc.: Case Systems: Xephon Buyers Guide, 1988.

Appendix 1: High quality diagram output-all in one page



Frames for protocol description*

SALLY WAGNER-DIBUZ

Central Research Institute for Physics
H—1525 Budapest, 114, P.O.B. 49., Hungary

1. Introduction

The increasing number of computer networks and their spreading application leads to the production of new computer network protocols. The specification, implementation, and test sequence generation for these protocols is a tiring work, which has to be supported by software tools. Expert system technology can be applied well in the field of communication protocols.

This paper deals with the application of frames for protocol description, which gives a formal description of the protocol. The transformation from other formal description methods into the protocol representation with frames can be given. The frame representation is a new method for protocol description, which provides a flexible and well-structured description of the protocol.

Frames are useful for representing the protocol development process too. Using a frame-based knowledge representation method data and procedural knowledge can be represented in the same way, with frames. However frames describing the protocol can be easily separated from the frames and demons representing the procedures of the protocol engineering process. This means that the protocol description with frames can be the knowledge base of an expert system for protocol engineering [WAG87], [TAR88].

In the paper examples are shown for the frame representation of protocols, and for the usage of the protocol description for the simulation of the communication between two transport entities. For demonstration the frame-based language FAIR developed in SZKI is used.

Section 2 contains a brief overview of communication protocols. In Section 3 we write about knowledge representation with frames. In Section 4 a protocol model is given, which can be represented with frames as it is described in Section 5. Section 6 summarizes the conclusions.

* Lecture presented at the 1st Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

2. Communication protocols

PROTOCOLS

The reference model defines seven layers in a communication network. Each entity in a layer communicates with the entities in the lower layer, with the entities in the upper layer and with the peer entities which are placed in the same layer but in other nodes. The peer entities communicate with each other according to the protocol (fig. 1.). The protocol defines the messages which can be sent during the communication, it determines the semantics of the communication and its time attributes. For the communication with the upper and the lower layers the entity uses service primitives (*SP*), and for the communication with the peer entity protocol data units (*PDU*) are used. Protocols have to be standardized so that computers produced by different producers are able to communicate with each other. There are several protocols in one layer especially in the application layer, where each application needs a new protocol.

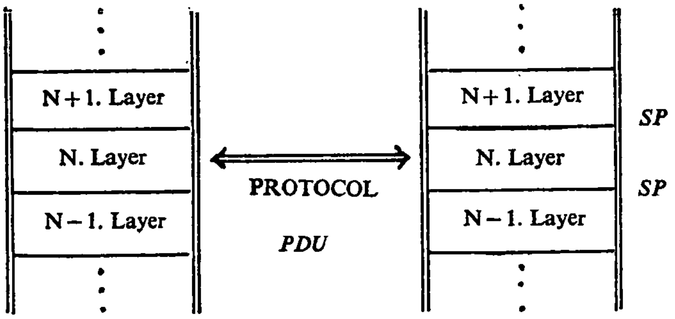


Fig. 1. The communication between the entities

PROTOCOL ENGINEERING

The protocol standard is a long and complicated description of the protocol in English. It defines everything that have to be stated about the protocol: the syntax of the messages (the bit-map), the possible responses to a given message, the parameters and their values, and the protocol mechanisms. The process in which the protocol standard is realized with an entity that is able to communicate with other entities using the protocol is called protocol engineering. (Fig. 2.) It consists of the following tasks:

— protocol specification,
— protocol verification,
— protocol implementation,
— conformance testing.

All of these activities need tiring and mostly manual work. There is a great need for software tools which support protocol engineering. Formal methods for protocol description are needed for these software tools. These formal descriptions give a more exact and clearly arranged description of the protocol, which is called the

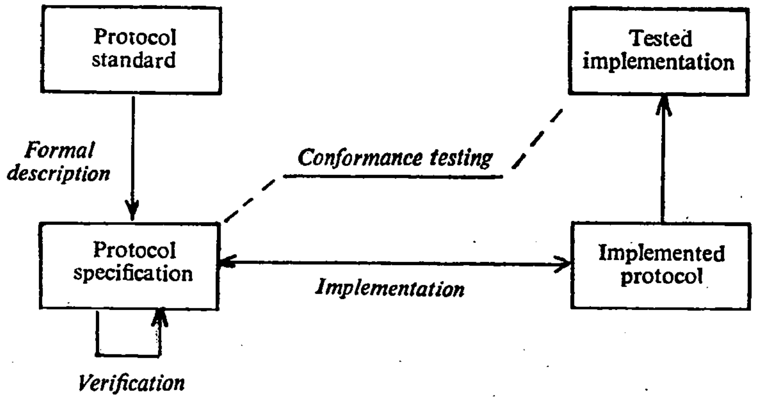


Fig. 2. Protocol engineering

protocol specification. It is easier to understand the protocol operation from the protocol specification and easier to implement the protocol in a programming language using the protocol specification. The testing of the implemented protocol is also based on the protocol specification.

3. Knowledge representation with frames

Frame—based systems usually treat information on three different levels:

— frames,
— frame—structures, and
— worlds.

Generally a frame can be considered as a structured symbolic model of a concept. Actually it is a (usually ordered) set of arguments, called slots, each of which is used to represent a property of the concept to be modelled. Every slot may have a value, which can be another frame, or an expression in some formal language.

In order to cope with the complexity of real world concepts meta—information can be associated to any part of a frame:

— to the frames,
— to the slots, and
— to the values.

The meta—information is also formalised in frames, these are called meta—frames, and the pieces of meta—information are stored in the slots of the meta—frames. This information can be used to define the range of the slot values, demons for describing the protocol activities, new frames for storing further and deeper information about the slot values, questions for filling in the slots with a given protocol etc. In this paper we use meta—frames only for describing range of values, and demons, which specify the "active" part of the program.

Frame—structures are used to represent the relations among the concepts modelled by the frames. This means that if certain slots are not found in a given frame, then search is made by the system along the relations to see if the slot and its value are contained in another frame accessible from the present one along such relations. If so then the slot and its value will be inherited by the original frame. In our example we shall use only the "is_a" relation. If two frames are in "is_a" relation it means that the first will inherit all lacking information from the other one.

Worlds are groups of frames, which can represent several things, like hypothetical alternatives, processes changing with time etc.

THE FAIR

FAIR (Frames in Artificial Intelligence Representation of knowledge) is a frame—based system integrated with MPROLOG. The form of a frame in the FAIR language is the following:

```
frame: name;
     slot_1: value_11, value_12, ..., value_1n1;
     slot_2: value_21, value_22, ..., value_2n2;
     ...........................................................
     slot_m: value_m1, value_m2, ..., value_mnm;
end.
```

The language contains several predicates for manipulating the frames. Here we enumerate only those predicates which are used in the example in Section 5:

create_frame	crf
access_value	acv
find	

The interested reader can find further details about frame—based knowledge representation in [BAR86] and in [ECS88], and about the FAIR language in [ECS 89].

4. A model for protocols

In this paper we use the abstract state machine (ASM) model of the protocol. The model is defined by seven sets:

$$(S, I, O, V, A, P, T)$$

where
S: is the finite set of states.
I: is the finite set of the incoming messages of the protocol.
O: is the finite set of the outgoing messages of the protocol.
V: is the finite set of variables, which can be divided into two distinct sets:
— local variables
— global variables.
A: is the finite set of actions, there are local actions concerning local variables, and global actions concerning all the variables.
P: is the finite set of predicates, which map the variables into the values true or false. There are three subsets of predicates:

— controlled,
— settable,
— non-controlled.

T: is the finite set of transitions, a transition is defined by the following:

$$(s, i, p, a, o, s')$$

which means that the automaton in the state "*s*" receives an input "*i*", and if predicate "*p*" is true it executes action "*a*", sends an output "*o*" and goes into state "*s'*".

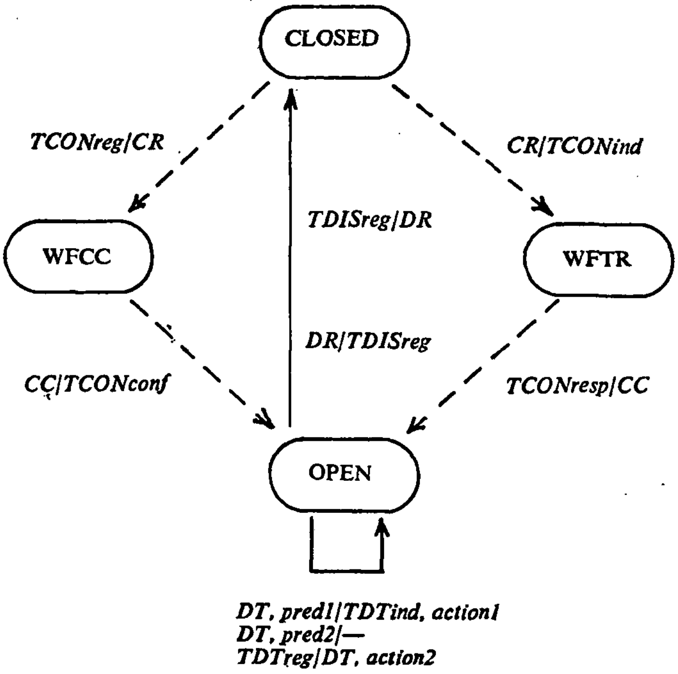


Fig. 3. Simplified state transition graph of transport protocol class 0

Fig. 3 shows the simplified state transition graph of the transport protocol class 0. Its ASM model is the following:

S = { CLOSED, WFCC, WFTR, OPEN }
I = { *TCONreq, TCONresp, TDISreq, TDTreq, CR, CC, DT, DR* }
O = { *TCONind, TCONconf, TDISind, TDTind, CR, CC, DT, DR* }
Vlocal = { SZEGM, ... }
Vglobal = { ∅ }
P = {*pred*1, *pred*2, ...},
A = {*action*1, *action*2, ...},
T = { (CLOSED, *TCONreq*, *CR*, WFCC), ... }

(The elements of the transitions which do not have values can be omitted.)

This is a very simple protocol, but the ASM model is capable of describing more complex protocols. The ASM model can be the basic model of the protocol in an expert system for protocol engineering. In the next section we demonstrate how this model can be stored in a frame structure.

5. A frame based approach to protocol description

Frames describing the ASM model can be divided into three groups:

— atomic,
— complex,
— pattern.

An atomic frame contains a state transition. The form of the generic frame for a transition is the following:

```
frame: transition;
     start_state: ;
     input: ;
     predicate: ;
     action: ;
     output: ;
          meta_slot: output_meta;
               if_accessed_demon: output_demon;
          meta_end
     end_state: ;
          meta_slot: end_state_meta;
               if_accessed_demon: end_state_demon;
          meta_end
end.
```

The slots of the meta—frames contain demons which are activated when the value of the slot is accessed during the operation of the program. These demons can be for instance procedures writing out the output and the end—state on the display. The frame of a transport protocol state transition is the following:

```
frame: edge5 ;
     is_a: transition;
     start_state: open;
     input: dt;
     predicate: ;
          meta_slot: ;
               if_accessed_demon: pred1;
          meta_end
     action: action1;
     output: tdtind;
     end_state: open;
end.
```

Complex frames store the information about the elements of the sets defining the ASM model. The following is the generic frame for the inputs, and the next frame represents the inputs of our transport protocol example:

```
frame: inputs;
    asp_from_upper_layer: ;
    asp_from_lower_layer: ;
    pdu: ;
end.
frame: trans_inputs;
    asp_from_upper_layer: tconreq, tconresp, tdisreq, tdtreq;
    pdu: cr, cc, dr, dt;
end.
```

The bit format of the messages, their parameters with their values, can also be represented by complex frames.

Pattern frames store test suits, or subgraphs of the state transition graph. This information is available also in the atomic frames in the description of the transitions. Such redundant information is useful in the procedures of protocol engineering.

```
frame: pattern;
    states: ;
    inputs: ;
    outputs: ;
    transitions: ;
    represented_actions: ;
    fixed_variables: ;
end.
```

This is the example of a pattern frame representing the connection establishment phase of the transport protocol:

```
frame: connection_establishment;
    is_a: pattern;
    states: closed, wfcc, open;
    inputs: tconreq, cc;
    outputs: cr, tconconf;
    transitions: edge1, edge2;
end.
```

COMMUNICATION SIMULATION

The communication of two transport protocol entities can also be described with frames. For this all the information stored about the protocol in the frames have to be put in work. This can be done by demons, which are procedures written in Prolog in our case, as the FAIR language uses PROLOG. Two frames represent the protocol entities, and one more frame represents the channel connecting the entities. They are the following:

```
frame: transport_entity_1;
     actual_state: ;
     input: ;
          meta_slot: ;
               range: tconreq, tconresp, tdisreq, tdtreq, cr, cc, dr, dt;
               if_added_demon: state_transition;
          meta_end
end.
frame: channel;
     message: ;
          meta_slot: ;
               if_added_demon: transmit;
          meta_end
end.
```

The state_transition demon is activated when the input slot of the transport_entity frame gets a new value. This demon searches through the state transition graph of the protocol (*frames edge_1,..., edge_n*) to find the transition, which the protocol presents in this part of its operation. The frame: *transport_entity_i* tells the actual state of the protocol and the message it receives from the other entity. The appropriate transition can be found using this information. If such a frame is found, then its predicate has to be examined. If it succeeds, the demon in the action slot of the frame representing the transition is run, and the output message and the new state of the protocol can be found in the frame. If the examined predicate doesn't succeed, the search through the transitions is continued, as there must be a transition which has the same start_state, input, and which predicate succeeds. The following PROLOG statement is the activity part of the *state_transition* demon:

```
state_transition_activity: —
     acv(actual_entity,name;N),
     acv(N,input,I),
     acv(N,actual_state,S),
     find(Frame,[start_state, input, predicate], [S, I, succeed]),
     run(Frame, action),
     acv(Frame, output, O),
     acv(Frame, end_state, E),
     crf(N, actual_state, E).
```

6. Conclusions

In this paper we described a method for representing a protocol in a knowledge base. The frame—based language can be used in a natural manner for specification of protocol data units and the communication itself. This kind of protocol specification is:

— Comprehensible even by non-expert user.
— Executable directly by machines.
— It can be used as a protocol description for protocol engineering methods.

The examples given in the paper show some further advantages:

— Every aspect of the protocol can be described as deeply in the hierarchy of frames as it is needed by the user.

— The data types and the control mechanisms of the protocol can be defined by frames in an identical way.

— It gives a runnable description of the protocol, the operation of the protocol can be easily simulated and demonstrated.

References

[BAR86] Barr, A., Feigenbaum, E. A., The handbook of artificial intelligence I—III., Heuris-TECH Press, Stanford, 1986.

[ECS88] Ecsedi-Tóth, P., A frame-based approach to protocol engineering, Technical report of SZKI, 1988.

[ECS89] Ecsedi-Tóth, P., Wagner, P. A., "FAIR, a frame-based system integrated with MProlog", Proc. of Expert Systems Conf. Visegrad, 1989.

[TAR88] Tarnay, K., Wagner-Dibuz, S., "A protocol consultant", Proc. of DECUS Europe Symposium, Cannes, Sept. 1988.

[WAG87] Wagner-Dibuz, S., Protocol consultant, an expert system for protocol engineering, KFKI report 28—M, Budapest, 1987.

A Programming Environment for a Transputer-Based Multiprocessor System*

M. Aspnäs and R. J. R. Back

*Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14, SF—20520 Turku, Finland*

Abstract

This paper presents a transputer-based multiprocessor system, Hathi—2, and the programming environment being developed for this system. Hathi—2 is mainly programmed in the language Occam, and thus the programming environment is based on the Occam model of parallelism and communication. The programming environment gives the user an abstraction of the physical structure of the multiprocessor system. The user sees the multiprocessor system as a pool of resources (processors and communication links), which are allocated to the users program and connected to the topology described by the program structure. The environment is implemented on a Sun graphical workstation.

1. Introduction

This paper describes the design of a graphical programming environment for a transputerbased multiprocesor system. The programming environment consists of a number of program development tools integrated under a common graphical user interface.

The Hathi-2 multiprocessor system was designed and built in a joint project between the Department of Computer Science at Åbo Akademi and the Technical Research Center of Finland (VTT/TKO) in Oulu. As a part of the project, a number of application programs have been implemented on Hathi-2. The experiences gained from the applications show that more sophisticated program development tools are needed for multiprocessor systems of this kind. At present, programming multiprocessor systems is considered more difficult than programming sequential computer system. This is mainly due to the lack of programming tools available for use in the design and debugging of parallel programs.

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

A parallel program for a MIMD-type multiprocessor system is normally designed as a number of independent sequential processes, which communicate with each other by sending and receiving messages through point-to-point commuication channels. When writing a parallel program, the logical process network is first designed. The logical process network describes the structure of the processes in the program and their interconnections through logical communication channels. The processes are written and tested separately, until the programmer is confident in their behaviour. After this, the programmer decides how these processes are placed on physical processors and executed in parallel. Two steps are required to do this: first, one must describe how the processes are placed on physical processors and what communication links connecting these are needed, and second, the multiprocessor system has to be connected (reconfigured) into this topology.

Both these steps involve a substantial amount of work for the programmer and introduce an additional source of errors. When the programmer has written a parallel program, he wants to experiment with different processor interconnection topologies and process placement schemes and make the program as well balanced and effective as possible. This is done by monitoring the execution of the parallel program and identifying the bottlenecks of the program. Information about the utilization of the physical resources used by the program during execution is gathered and presented to the user. The bottlenecks in a parallel program are usually caused by either overloaded physical communication channels or by processors which are allocated too much computation. In the ideal case, all physical resources have an evenly distributed utilization, and no bottlenecks exist in the program. To remove an identified bottleneck, the programmer has to change the logical process network, the placement of the logical processes on the physical processes or the interconnection structure of the physical processors. Often all these are changed simultaneously, and the programmer has to place the logical processes onto the physical structure again, and the design cycle is repeated.

To identify and remove logical errors in a parallel program, the programmer wants to observe the logical behaviour of the program during execution. In a parallel program, this can be done by using algorithm animation techniques, in which the program execution is presented to the user in a graphical way as an animation of the execution. Traditional methods for program debugging (traces, breakpoints etc.) can not generally be used, as there is no global control of the system.

Thus, the programming cycle for parallel programs consists of designing the logical process network and the processes, reconfiguring the pysical process network into a suitable topology, mapping the logical process structure onto the physical processor network, debuging and correcting logical programming errors and monitoring the execution of the program to identify bottlenecks, which often leads to changes in the logical program structure, and so the cycle is repeated.

At present, the programmer has to do all these steps manually. Clearly, some of these steps could be done automatically by a set of programming tools. The programming environment presented in this paper gives the user this type of utilities, by providing an integrated set of tools for mapping a process structure onto a physical processor network, monitoring the resource utilization of an executed program and animating the logical behaviour of a program. The presented programming environment provides the user with an abstract view of the multiprocessor system by hiding the physical interconnection structure of the system from the user.

The paper is organized as follows: the architecture of the Hathi—2 system is presented in Section 2. In Section 3 we give a short description of the programming language Occam. In Section 4 we describe the programming environment and finally, in Section 5 we describe the future developments of the presented programming environment.

2. The Hathi-2 Multiprocessor System

Hathi-2 is a reconfigurable general purpose multiprocessor system consisting of 100 32-bit IMS T800 transputers [Inml], 25 16-bit IMS T212 transputers and 25 IMS C004 crossbar switches. The system can be characterized as a loosely coupled MIMD multiprocessor, with a reconfigurable distributed interconnection network and a modular design. A more detailed description of the Hathi-2 architecture and its use can be found in [AsBaMa], [AsMa] and [Peh]. The distributed switching network is described in [Äij].

Hathi-2 consists of 25 identical boards, each containing four T800 transputers, one T212 transputer and one 32 link crossbar switch. The T800 transputers are connected pairwise to each other via one of the four communication links. The three remaining links are connected to the crossbar switch (see Fig. 1). Three links from each switch are used as I/O links, i.e., to connect users host computers and peripheral units to the system. The remaining 16 links from the crossbar switch are used to form a statical torus connection between the boards in the Hathi-2 system, thus forming the distributed switching network.

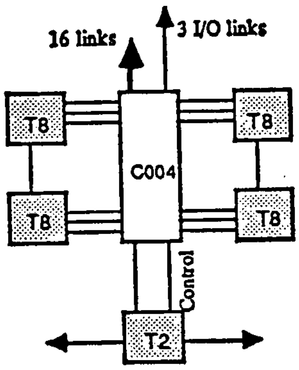


Figure 1. Hathi-2 board architecture

The C004 crossbar switch is controlled by the T212 transputer via a control link. One link on the T212 is connected to the crossbar switch and can be connected via the switch to any other transputer link. The two remaining links on the T212 (links 0 and 1) are used to connect the T212 transputers into a ring, thus forming the distributed control system.

The crossbar switches on the Hathi-2 boards are connected to each other in a static torus connection by connecting each pair of neighbouring boards to each

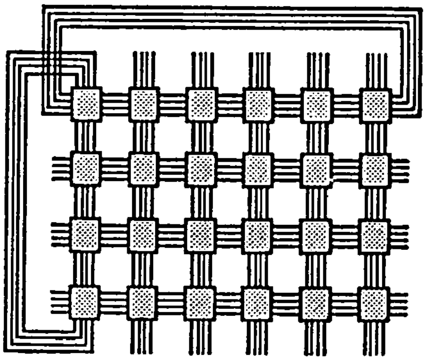


Figure 2. Hathi-2 board connections

other with four links (see Fig. 2). The crossbar switches form a distributed switching network connecting the communication links of the T800 transputers, which enables the system to be reconfigured by software.

Hathi-2 is used as a back-end computing resource. The user edits, compiles and links his programs on a host computer, i.e., a Sun workstation. The program can then be loaded on to the multiprocessor system and executed.

The Hathi-2 system can be shared between a number of simultaneous users by paritioning it into several smaller independent multiprocessor systems (see Fig. 3). All users are allocated a separate partition which is independent of all other partitions. A user has full control over his own partition, but can not interfere with other users.

The T212 transputers are connected to each other in a ring, thus forming a separate control system which controls the switching network (see Fig. 4). The control system is totally independent from the rest of the system. The only connection between the user and the control system is via a link connecting one T212 transputer to the

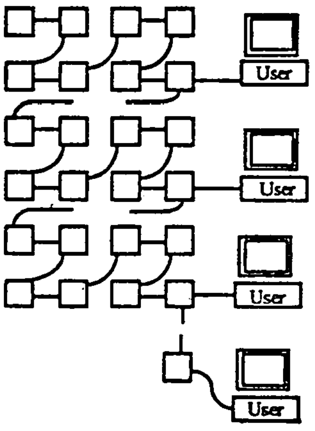


Figure 3. Partitioning the system

users host computer. The user can request system services by sending commands to the control system via this link.

The control system has two main tasks: to control the distributed switching network and to monitor the activities in the system. The Hathi-2 architecture contains hardware dedicated to monitoring the resource utilization in the system. The monitoring hardware consist of a CPU load meter which measures the CPU utilization by observing the bus activity and a FIFO buffer connecting all T800 transputers on a board to the controlling T212 transputer. The FIFO buffer can be used for sending reports about resource utilization from the T800 to the T212 without affecting the communication links.

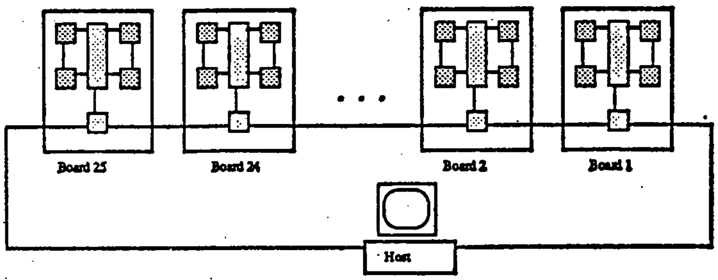


Figure 4. The control system

The control system also contains an interrupt subsystem implemented using the transputers EVENT interrupt. A processor in the control system can send an interrupt signal to all processors in the same partition. This interrupt is used in the monitoring system to generate a synchronizing signal which divides the time into short time intervals. The CPU and link utilization are measured for each interval and reported to the user.

3. The Occam programming language

Occam [Inm2], [JoGo] is a high-level programming language based on the CSP language [Hoa]. An Occam program consists of a number of sequential processes, which communicate with each other via unidirectional channels using synchronous message passing.

A channel connects two processes, of which one acts as a sender and the other as a receiver. A process sends a message *M* via a channel *c* with an output statement *c*!*M*, and the receiving process inputs a message from the channel to a local variable with an input statement *c*?*M*. A process can wait for input from a number of channels at the same time, using an **ALT** construct. The sending process can not choose between different communication alternatives, but commits itself to a communication when it executes an output statement. Communication is synchronous, i.e., the process which first executes a communication statement remains waiting until its communication partner executes a corresponding communication statement.

Parallelism is expressed in occam by the **PAR** construct, which specifies that

```
PAR
  SEQ
    X:=5
    c!X
  SEQ
    c?Y
    Y:=Y*2
```

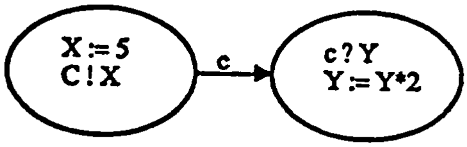


Figure 5. Communicating processes in Occam

two or more processes are executed in parallel. Sequential execution is specified with the SEQ construct. Scope is expressed in Occam by indentation. In the example in Figure 5, two processes communicate with each other via a channel *c*.

More than one process can be executed simultaneously on one transputer. The transputer divides its time between processes using a simple round-robin scheduler, which is built into the transputer hardware. Communication between processes executed on the same transputer is implemented through memory locations.

To execute a program with real parallelism on more than one transputers the programmer has to describe on which transputers the processes are to be executed and which communication links are used for communication between the processes. This is done by an Occam-like configuration language. The example in Figure 6 describes a ring of three processors, each executing a process Calculate. The processes communicate with each other by inputting from link 3 and sending on link 2. The user thus has to explicitly describe on which processor each process is executed and which communication links are used for communication between the processes. This means that the user has to have detailed knowledge about the hardware structure of the multiprocessor system.

```
CHAN OF INT C0, C1, C2:

... SC PROC Calculate (CHAN OF INT From.previous, To.next)

PLACED PAR

 PROCESSOR 0 T8
 PLACE C0 AT 2 :-- link2out
 PLACE C2 AT 7 :-- link3in
 Calculate (C2, C0)

 PROCESSOR 1 T8
 PLACE C0 AT 7 :-- link3in
 PLACE C1 AT 2 :-- link2out
 Calculate (C0, C1)

 PROCESSOR 2 T8
 PLACE C1 AT 7 :-- link3in
 PLACE C2 AT 2 :-- link2out
 Calculate (C1, C2)
```

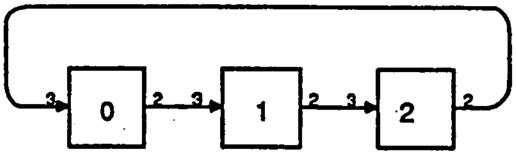


Figure 6. Placing processes on processors

4. The programming environment

The programming environment developed for the Hathi-2 multiprocessor system is designed by integrating a number of tools and utilities under a graphical user interface. The approach taken has been to use as much as possible of already existing software, i.e., editors, compilers, configurers, network loaders and debuggers. This is possible, because the Hathi-2 architecture is fully software compatible with Inmos transputer products.

The utilities that have been developed for Hathi-2 in the project are based on the specific hardware characteristics of the system and are not directly portable to other architectures. These tools include a utility wich allows the user to reconfigure the topology of the system, a monitoring utility which is used for monitoring the utilization of the resources of the system, and an animation tool which is used to visualize the execution of a parallel program.

The goal of this work is to make the multiprocessor system easier to use for the programmers by building a user-friendly graphical interface to the tools, and to hide the physical structure of the multiprocessor system from the programmer. The user should be able to construct a parallel program for the Hathi-2 system entirely within the programming environment. The whole cycle of editing a program, compiling, loading the program onto a number of processors and executing it, debugging the program and monitoring the performance of the program can be carried out within the programming environment.

4.1. The user interface

The user interface of the programming environment is based on a hierarchical graph editor. The user describes the *process structure* of a distributed program by drawing a graphical representation of the processes and their interconnections. The graph representing a parallel program consists of a number of nodes and arcs, the nodes representing processes and the arcs representing communication channels between the processes. A node in the process graph is associated either with a sub-graph or directly with the code of the process. The source code describing a process can be edited by selecting the node representing the process by clicking on it with the mouse. This will bring up the Occam folding editor, and the code of the process can be edited in the normal way.

The processes in the process graph are grouped together to form *tasks.* A task is a separately compiled unit of code (in Occam called a *SC*), which is executed on one processor and usually consists of a number of parallel communicating processes. The processes constituting a task are executed on one transputer using the transputers timeslicing scheduler. Thus, the process graph is condensed into a *task graph,* which determines the physical structure of the processor network on which the program is to be executed. In Figure 7 is an example of a process graph, which is condensed into a task graph using four processors connected into a pipeline. The physical communication links connecting processors are drawn with fat lines, and are always bidirectional (consisting of two unidirectional links).

The utilities in the programming environment use the information about the distributed program contained in the process graph, the source code of the processes and the grouping of the processes into tasks. The editor used is a stand-alone

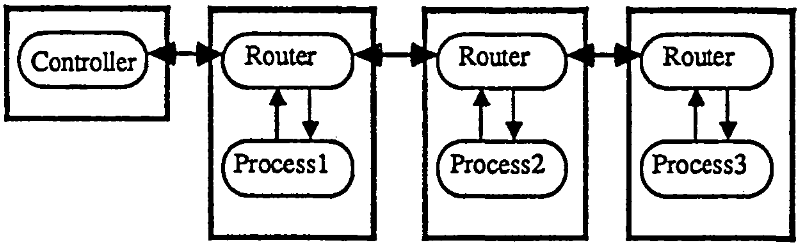


Figure 7. A process graph partitioned into a task graph

Occam folding editor. Similarly, the compiler, the configurer, the network loader and the debugger are the stand-alone Occam program development tools from Inmos. When the user invokes one of these tools by selecting an apropriate entry from a menu in the user interface or by clicking on a node in the process graph, this is translated to a corresponding Unix call which activates one of these utilities.

4.2. The mapping utility

The mapping utility developed for Hathi-2 automatically maps a task graph onto the transputers in Hathi-2 and establishes the needed link connections between the transputers. The input from the user to the mapping utility consists of the task graph of the distributed program. As output, it generates the configuration statements needed by the Occam configurer to place this program structure onto a physical topology. The mapping utility also generates the commands needed by the reconfiguration software to connect the transputers into the topology described by the task graph.

The mapping utility makes it possible to hide the physical structure of Hathi-2 from the user. The user does not have to explicitly specify which of the four links on a transputer should be used for communication with other processors. This is a very useful feature when writing parallel programs, since the design of the configuration statements is considered to be difficult and very error-prone. The mapping of the processor graph onto the physical structure of Hathi-2 is done by a simulated annealing algorithm [Bok].

It is possible to find processor structures that cannot be mapped to the hardware structure of the multiprocessor system. First, not all graphs can be established on a transputer network, because a transputer has only four links. One example of this is a 5-dimensional hypercube, which requires a node degree of five. Second, the architecture of the distributed switching network in Hathi-2 imposes some limitations on which graphs can be established. The main limiting factor here is that there are only four links available between every pair of neighboring boards in the static torus interboard connection. Finally, the algorithm used in the mapping utility does not guarantee that a mapping of a graph to the structure of Hathi-2 is found. However, the mapping algorithm has proved to work well in practice for a large class of problems.

4.3. The monitoring utility

The monitoring utility is used for monitoring the utilization of the resources in the multiprocessor system during program execution. It is used for finding bottlenecks in parallel programs executing on the system, and to provide information about the load balance of the programs. Monitoring is done by observing the CPU and link activity in the transputer network. The monitoring software is based on the monitoring hardware built into the Hathi-2 architecture, which makes it possible to monitor the system without introducing any substantial overhead on the main computation.

Monitoring data, i.e., data about CPU and link utilization on the transputers executing the monitored program, is gathered by the transputers in the control system. This data is sent through the control system to the users host computer, where it is stored in a file and presented to the user.

The time during which monitoring is done is divided into short time intervals (typically 100 to 500 milliseconds), and for each interval the monitoring system records the percentual utilization of the CPU, the number of bytes transmitted over a link and the time a process has spent waiting for a communication to take place.

The monitoring data is presented to the user as the average percentual utilization of the CPUs and links during an interval. The presentation is based on the task graph of the distributed program. For each transputer, its percentual CPU utilization is presented as a number written inside the node representing the transputer in the graph. Similarly, the percentual utilization of each link is written above the arc representing the link in the graph. In Figure 8 there is an example of how the results from a monitored program is presented to the user.

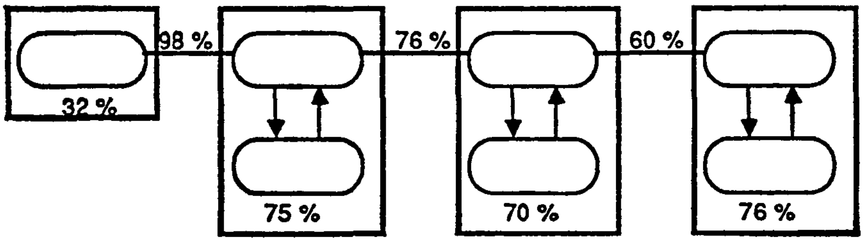


Figure 8. Presentation of monitoring data

The user can control the length of the time interval. Monitoring data is sampled with a fixed interval during program execution, but the data can be presented with any interval longer than this. If the user wants to see the result of the monitoring in an interval longer than the sampling interval, the mean values from a sufficient number of sampling intervals are calculated. The user can browse through the monitoring data both backwards and forwards in time. Normally, the first time a program is monitored, the user wants to view the result using a rather large timestep, to get an overall picture of the behaviour of the program. The user can later examine the execution of the program more closely, using a smaller time interval.

4.4. The animation utility

A program animation is a graphical visualization of the execution of a program [Sol]. Program animation is used as a high-level debugging tool, which gives the programmer an understanding of how his program behaves during execution. This is especially important for parallel programs, as it is very difficult to get a picture of the overall behaviour of a program executing on a large number of processors.

Animation of parallel programs on the Hathi-2 system is implemented using the same hardware features as the monitoring system, i.e. the FIFO buffers. In the animation system, the data sent from the transputers to the control system contain information that controls the graphical animation of the executed program. This data is interpreted as graphical commands, which are executed by an animation process and which result in a graphical illustration of the program execution.

The animation is done by inserting commands into the animated processes, which send messages about their present state of execution to the animation process. The animation process receives these messages and translates them into graphical commands that update the screen. The user has to specify on which points in the execution the state of the program should be reported. The user also has to describe how each state should be represented in the animated picture. This is done by a graphical tool, which allows the user to draw the pictures of which the animation consists.

The execution of the animated program must also be slowed down, so that the user has time to register the updates on the screen. The program is slowed down uniformly, without affecting the logical behaviour of the program. This is implemented using the synchronization mechanism in Hathi-2. All processes are forced to wait for a synchronization signal, which is sent from the control system.

5. Conclusions and future work

This report describes work in progress in the Millipede project at Åbo Akademi. The tools that have been described have already been implemented and are now beeing separately tested and evaluated. The design of the graphical user interface which integrates the tools into a programming environment has recently started and is scheduled to be ready in the last quarter this year. After that, the environment will be evaluated and any possible further improvements and features will be considered.

Several components of the environment will be developed further. The routing algorithm used by the reconfiguration software for establishing link connections between processors through the distributed switching network will be developed [Shen1]. Also the mapping algorithm which is used for mapping a task graph to a physical configuration of Hath-2 will be improved by investigating different types of heuristic algorithms [Shen2], [Shen3]. Finally, the graphical user interface will be developed, based on the experiences of the users. The goal is to make the environment as simple as possible to use for the programmers.

Acknowledgements

The Hathi-2 multiprocessor system was designed and built in the Hathi project, which was financed by TEKES, The Academy of Finland, Åbo Akademi and VTT. Part of the work has been done in the FINSOFT III research program, financed by TEKES. The authors also wish to thank Kaisa Sere for comments on the paper.

References

[AsBaMa] M. Aspnäs, R. J. R. Back, T-E. Malén, The Hathi-2 Multiprocessor System, Reports on Computer Science, Ser. A, No. 80, Åbo Akademi, 1989.

[AsMa] M. Aspnäs, T-E. Malén, Hathi-2 Users Guide, version 1.0, Reports on Computer Science, Ser. B, No. 6, Åbo Akademi, 1989.

[Ber] F. Berman, Experience with an Automatic Solution to the Mapping Problem, in The Characteristics of Parallel Algorithms, Jamisson, Gannon and Douglas (ed.), MIT Press, 1987.

[Bok] S. Bokhari, On the Mapping Problem, IEEE Transactions on Computers, C-30, no. 3 (March, 1981), pp. 207—214.

[CrMa] P. Crol and G. Manson, Configuration Tools for a Transputer Workstation, in Applying Transputer Based Parallel Machines, A. Bakkers (ed.), Proceedings of the 10th Occam User Group Tecnical Meeting, Enschede, Netherlands, IOS, 1989.

[EkMa] P. Eklund, T-E. Malén, Block Placement in Switching Networks, Proc. CONPAR-88, Manchester, Great Britain, Cambridge University Press, 1988, pp. 289—295.

[EMMM] J. Eudes, F. Menneteau, L. Mugwaneza and T. Muntean, PDS: Advanced Program Development System for Transputer Based Machines, in Applying Transputer Based Parallel Machines, A. Bakkers (ed.), Proceedings of the 10th Occam User Group Tecnical Meeting, Enschede, Netherlands, IOS, 1989.

[Hoa] C. A. R. Hoare, Communicating Sequential Processes, Communications of the ACM, 21, 8 (Aug. 1978), pp. 666—677.

[Inm1] Inmos Limited, *Transputer Reference Manual*, Prentice-Hall, 1988.

[Inm2] Inmos Limited, *Occam 2 Refernce Manual*, Prentice-Hall, 1988.

[JoGo] G. Jones and M. Goldsmith, *Programming in Occam 2*, Prentice-Hall, 1988.

[Peh] K. Pehkonen, A Dynamically Reconfigurable Parallel Computer Hathi-2, Licentiate thesis, University of Oulu, Department of Electrical Engineering, 1989.

[Shen1] H. Shen, Fast Path-disjoint Routing in Transputer Networks, to appear in Proc. First Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, 1989.

[Shen2] H. Shen, Mapping Parallel Programs onto Transputer Networks, to appear in Proc. Australian Transputer and OCCAM User Group Conference, Melbourne, Australia, 1989.

[Shen3] H. Shen, Self-adjusting Mapping: A Heuristic Mapping Algorithm for Mapping Parallel Programs onto Transputer Networks, to appear in Proc. 11th Occam User Group Technical Meeting, Edinburgh, Great Britain, 1989.

[Sol] U. Solin, Animation Techniques for Parallel Agorithms, Proc. International Conf. on Parallel Processing and Applications, 23—25. 9. 1987, L'Aquila, Italy.

[Äij] T. Äijänen, Distributed Interconnection of a Reconfigurable Multicomputer System, Microprocessing and Microprogramming, 3—1988, pp. 243—246.

PICA — A graphical program development tool*

AIMO A. TÖRN

Åbo Akademi, Dep. Comput. Sci., DataCity
SF—20520 ÅBO, Finland

Abstract

A technique and a tool PICA for rigorous program development with flowcharts is presented. This technique uses stuctured program flowcharts extended with assertion nodes containing program variable names and assertions about their values. An assertion node is connected to or from a statement node depending on if it represents a pre-condition or a post-condition. A tool for convenient use of the technique has been implemented as an Add-On to the Design software of Meta Software on a Macintosh II. The feasibility of using PICA is demonstrated by developing an algorithm for a small non-trivial programming task. The incentive for presenting the PICA technique is to create broader interest for rigorous programming methods by presenting one technique applicable to program development using flowcharts.

Index Terms—Automatic programming, computer-aided design, graphics, flowcharting, program correctness, rigorous programming, software design.

1. Introduction

There exists a rather extensive litterature on rigorous program development only to mention the text books [Jones 1980, 1986; Gries 1981; Reynolds 1981; Bjørner and Jones 1982; Backhouse 1986]. However, at large rigorous methods are still rather seldom used by programmers in practice. Reasons for this may be that rigorous methods require additional knowledge from their users, that the methods are deemed as labourious and thus unpractical, and that they are not easily integrated with generally used informal program development methods.

In order for a technique to be accepted by a larger group the technique should not be more formal than necessary and should be naturally integrateable with some well known informal program development method. Our choice of informal method upon which to build the rigorous tool is the graphical flowchart based method used

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

in HOS [Hamilton and Zeldin 1976] and there adjusted to structured programming practice. For rigorous program development using plain text several techniques exist, however, since Floyd presented his technique [Floyd 1967] techniques for flowcharting tools seem to have rendered very little interest. Despite the fact that flowcharts are more expressive and allow easier screening because of their ability to efficiently use the possibilities of the two-dimensional medium (paper, screen) used by man, their use have decreased over the years. One explanation of this might be the shift from off-line to on-line program design using text editors, which normally do not support graphical representation. The programmer has thus been forced to choose between *on-line working — plain text representation* or *off-line working — graphical representation.* The effect has been further increased by the same change towards plain text representation which can be noticed in the programming literature.

The rapidly increasing number of installed workstations with graphics makes the technique *on-line working — graphical representation* available to an increasing number of programmers. There seems to be a rising interest today in using graphical representation to increase the quality (e.g., readability and correctness) of intermediate and final program designs by the proponents of rigorous program development [Buhr *et al* 1989]. We will here demonstrate the use of a graphical tool PICA (Program—Information Charts with Assertions) supporting rigorous program development. The tool has been implemented as an Add-On to the Design software of Meta Software. In PICA pre-and post-conditions are added to the flowcharts as explicite graph elements showing the program variable names together with assertions on their values [Törn 1980, 1981].

The PICA technique will be explained in Sec. 2 using a trivial programming task. In Sec. 3 the PICA tool is used to derive a program for *The Longest Upsequence* problem [Gries 1981].

2. The PICA Technique and Tool

We first discuss formal program development and then illustrate the PICA technique and tool using a simple programming task.

2.1. Formal Program Development. Programming aims at establishing the result condition R. Correctness of a program S thus means both finding S and verifying that R is true when the program stops (partial correctness) and verifying that the program will always stop (correctness).

Normally developing S and verifying R can be made only if some precondition Q is valid when the program starts. For instance, when a program for computing the square root of a real number is developed it is naturally assumed that the real number is greater than or equal to zero. However, the programmer cannot be sure that the program will always be used as intended and good programming practice therefore means that the program should address also the complementary case.

A well designed program should therefore contain an initial part that decides whether Q is valid or not and produces some "natural" result (e.g. an error message) for $\neg Q$ represented by the truthness of an exception condition R_e. Correctness further requires that Q and $\neg Q$ are evaluable and therefore a pre-initial part S_0 of the program (possibly containing inputs, must be written that secures this. This is in accordance with Floyd's PROMP-READ-CHECK-ECHO implementing the idea that each component of a program should be protected from input for which that component was not designed [Floyd 1979]).

A program specified in this way can be said to be *properly specified* because for each possible pre-condition a specification of the corresponding wanted result exists. A properly specified program thus has the following general appearance

$$S_0; S';$$

where

$$\begin{aligned} S'\colon\ &\mathbf{if} \neg Q \rightarrow S_e \\ &\blacksquare\ Q \rightarrow S \\ &\mathbf{fi} \end{aligned}$$

2.2. The Flowchart Technique. The flowchart elements used to describe *sequence, choice, iteration, refinement* and *assertions* are shown in Fig. 1. The elements are those used in HOS, with exception for the element of choice here represented structured in the same way as the element for iteration. Assertions are represented by dotted boxes divided into an upper and a lower part. The upper part contains the name of a variable, the lower part a predicate on that variable. The implied assertion is that the predicate is true.

Several assertion boxes may be connected by the logical operators *and, or* and $\Rightarrow$. A dotted arrow pointing from a statement box to an assertion box means that the assertion is valid after the execution of the statements in the statement box. A dotted arrow from an assertion box to a statement box shows that the assertion in the assertion box is valid when the computation reaches the statement box.

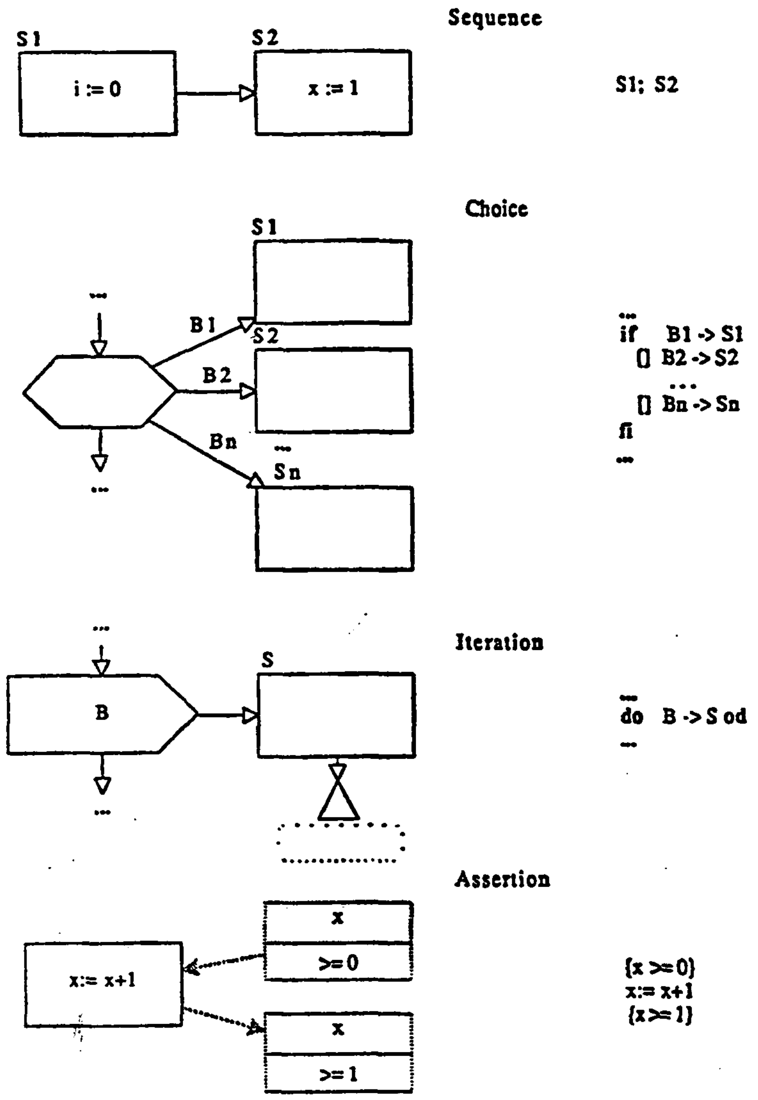
The flowchart in PICA notation corresponding to a properly specified program is shown in Fig. 2. The usual flowchart notation for the conditions valid at the branches of the *if*-statement (case statement) is used.

For proving that the program is correct we have to find S_e, S and to assert the result conditions in the PICA graph. For iterations it must also be proved that they are finite. The proof procedure consists of recursive applications of refinements of S, R and correctness proofs until such a program detail is reached that every step is sufficiently convincingly proved.

2.2. A Simple Programming Task. The result of using PICA for designing a program for adding the elements of a vector $x_1, \ldots, x_n$ is shown in the Appendix A. Some details of the PICA tool is also explained in the "text pages".

First a crude design is made. If a statement box must be refined a new flowchart page may be opened. On this child page an empty box with the surrounding of the refined box from the parent page will be exposed. The details of the design may then be introduced into the empty box. For each flowchart page there is a corresponding text page which can be used to complement the design so that a complete documentation of the design is obtained.

The tool is used in a three stage procedure. First the flowcharts together with the comments on the text pages are produced. For flowcharting a palette is available from which the flowchart elements needed are chosen and copied to the flowchart page. The tool is implemented on a Macintosh II with big screen which admits to have the plaette, the flowchart page and the text page open simultaneously. When the flowchart is ready the proof stage is entered. Unproved statement boxes have



```
S1; S2
```

```
...
if   B1 -> S1
  [] B2 -> S2
       ...
  [] Bn -> Sn
fi
...
```

```
...
do  B -> S od
...
```

```
{x >=0}
x:= x+1
{x>=1}
```

Fig. 1. PICA flowchart notations

thicker border lines than proved boxes. The PICA tool will keep track of the proof procedure so that the most refined parts have to be proved prior to cruder ones. There is no theorem proving facility available in the tool, i.e., the proofs are made informally by the user. In this activity the corresponding text page may be used to document the proofs. The tool will however check that pre- and post-conditions, and variants are existing where there must be such. It also reminds the user what

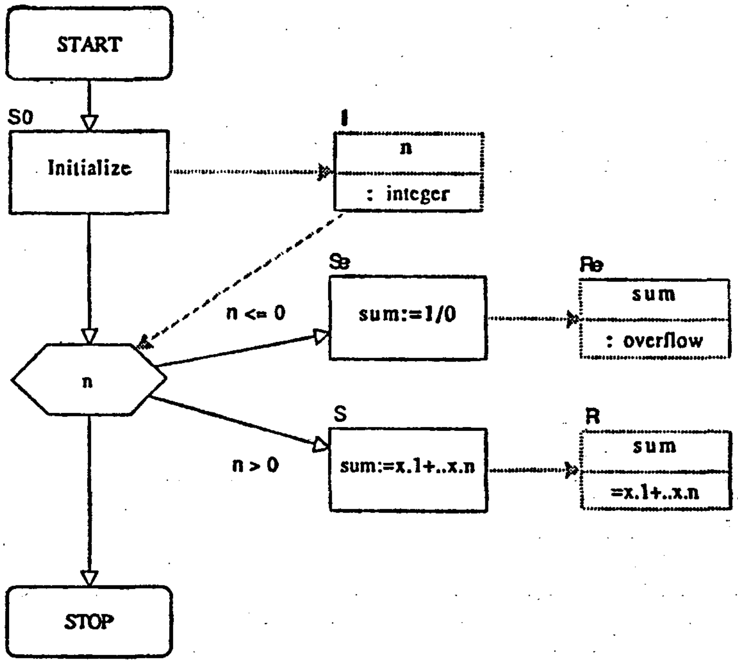


Fig. 2. General form of a program in PICA

have to be checked in order for a proof step to be complete. When the whole design has been proved correct the third stage which produces a skeleton program text may be entered. A printout from this stage is presented on the second text page of the design in Appendix A.

3. The Longest Upsequence Problem

The PICA tool will here be applied to the problem of designing an algorithm for finding the length of the longest upsequence *lup* (longest *up*) of a given vector $x_1, \ldots, x_n$, $n \geqq 0$. Based on the experiences of this some points are then discussed.

3.1. The Lenght of the Longest Upsequence. Let a *up* over x be defined as

$$\begin{aligned} up\colon\ & (x_{(i)},\ldots,x_{(i+k)}),\ k \geqq 0,\\ & (\ldots,x_{(i)},\ldots,x_{(i+1)},\ldots,x_{(i+k)},\ldots) = (x_1,\ldots,x_n),\\ & x_{(i)} \leqq x_{(i+1)} \leqq \ldots \leqq x_{(i+k)}. \end{aligned}$$

The resulting PICA design is shown in Appendix B.

3.2. Discussion. The development of an algorithm for the length of longest *up* starts with the division of the task to be performed into two cases, one of which is executed on each application of the algorithm. In order to be able to make refirement of the algorithm S knowledge about the problem to be solved is needed. This knowledge is presented as theorems about problem entities. The development then proceeds by successive refinement, using the knowledge contained in the theorems, and verification until such a detail is achieved that the algorithm can easily be coded using the target programming language.

4. Conclusions

The feasibility of using a specific rigorous program development technique PICA with flowcharts has been demonstrated by developing an algorithm for a non-trivial but small programming task. The PICA design is more easily screened and checked because of the greater freedom of flowchart representation. The technique is equivalent to several suggested techniques for plain text algorithm representation. The incentive for presenting the PICA technique is to create broader interest for rigorous programming methods by presenting one technique for those programmers who are proponents of flowcharting techniques for program development. In order to aid in using PICA a graphical tool supporting formal program development based on PICA is available. In addition to supporting the graphical representation and administering the proof procedure the tool is also able to automatically generate the equivalent plain text representation of the design including the assertions. This skeleton program can then be transformed into a compile ready version by further editing.

References

[Backhouse 1986] R. C. Backhouse, *Program construction and verification*, Prentice-Hall, Englewood Cliffs, N. J.

[Bjørner and Jones 1982] D. Bjørner and C. B. Jones, *Formal specification and software development*, Prentice-Hall, Englewood Cliffs, N. J.

[Buhr *et al* 1989] R. J. Buhr, G. M. Karem, C. J. Hayes and C. M. Woodside, *Software CAD: A revolutionary approach*, IEEE Trans. on Software Eng. 15, 235—249.

[Dershowits 1980] N. Dershowits, *The evolution of programs*, Technical Report UIUCDCS—R—80—1017, Department of Computer Science, Uni. of Illinois at Urbana-Campaign, 212 pp.

[Floyd 1967] R. W. Floyd, *Assigning meaning to programs*, In: Mathematical aspects of computer science 19, Amer. Math. Society, 19—32.

[Floyd 1979] R. W. Floyd, *The paradigms of programming*, Comm. of ACM 22, 455—460.

[Gries 1981] D. Gries, *The science of programming*, Springer-Verlag, New York.

[Hamilton and Zeldin 1976] M. Hamilton and S. Zeldin, *Higher order software — A methodology for defining software*, IEEE Trans. Software Eng. SE—2, 9—32.

[Hoare 1969] C. A. R. Hoare, *An axiomatic bases for computer programming*, Comm. ACM 12, 576—580, 583.

[Hehner 1984] E. C. R. Hehner, *The logic of programming*, Prentice-Hall, Englewood Cliffs, N. J.

[Jones 1980] C. B. Jones, *Software development, a rigorous approach*, Prentice-Hall, Englewood Cliffs, N. J.

[Jones 1986] C. B. Jones, *Systematic software development using VDM*, Prentice-Hall, Englewood Cliffs, N. J.

[Reynolds 1981] J. C. Reynolds, *The craft of programming*, Prentice-Hall, Englewood Cliffs, N. J.

[Törn 1980] A. Törn, *Structured programming using program flowcharts containing explicite representation of data including assertions*, Technical Report 10, Department of Computer Science, Åbo Akademi, Finland, 16 pp.

[Törn 1981] A. Törn, *PICA — A flowchart tool for structured programming supporting proving*, Technical Report 16, Department of Computer Science, Åbo Akademi, Finland, 17 pp.

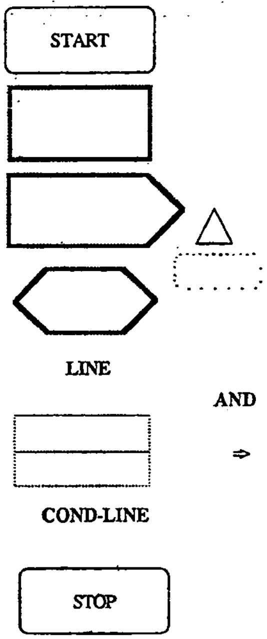
Vector Addition

A program for computing the sum of the elements of a vector x1+...+xn is to be designed. It is decided that n less than or equal to 0 is an exception and that the result produced in this case is an overflow condition.

The first part of the program is an initial part that guarantees that an integer is assigned to n. This is shown by the dotted arrow pointing from the box S0 to the box I.

From the condition box we have the two cases n<=0 and n>0. The box Se produces the exception and the box S the result for n>0.

The boxes have been copied from a palette similar to the one below.



LINE is chosen when connecting statement boxes and COND LINE when connecting a statement box and a assertion box.

There is a menu named FLOWCHART with the following options:

Open Palette

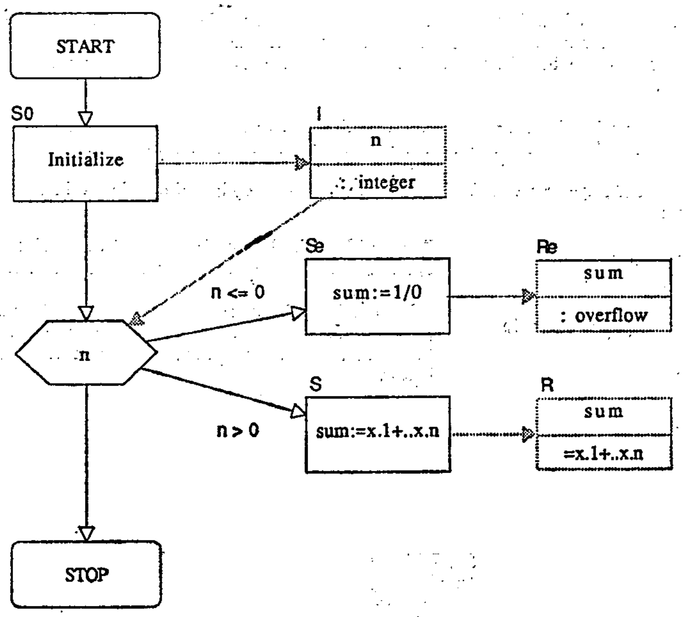
Open Text Page

Name Node
Fill in Guard
Refine Box

Set Obj. Horisontal
Set Obj. Vertical

Quit

We choose the box S and use the option Refine Box from the menu. This will produce a new flowchart page like the one on the next page but with initially empty inner part. Below S is initially only the box pointed to from the border.


START
S0
Initialize
I
n
: integer
n
n <= 0
Se
sum:=1/0
Re
sum
: overflow
n > 0
S
sum:=x.1+..x.n
R
sum
=x.1+..x.n
STOP

The refinement of the box S on the parent page is shown here. In the box S1 the initialization for the iteration is made and this makes the invariant P valid. The guard is given in box S2 and the variant n+1-i is shown at the end-of-iteration symbol.

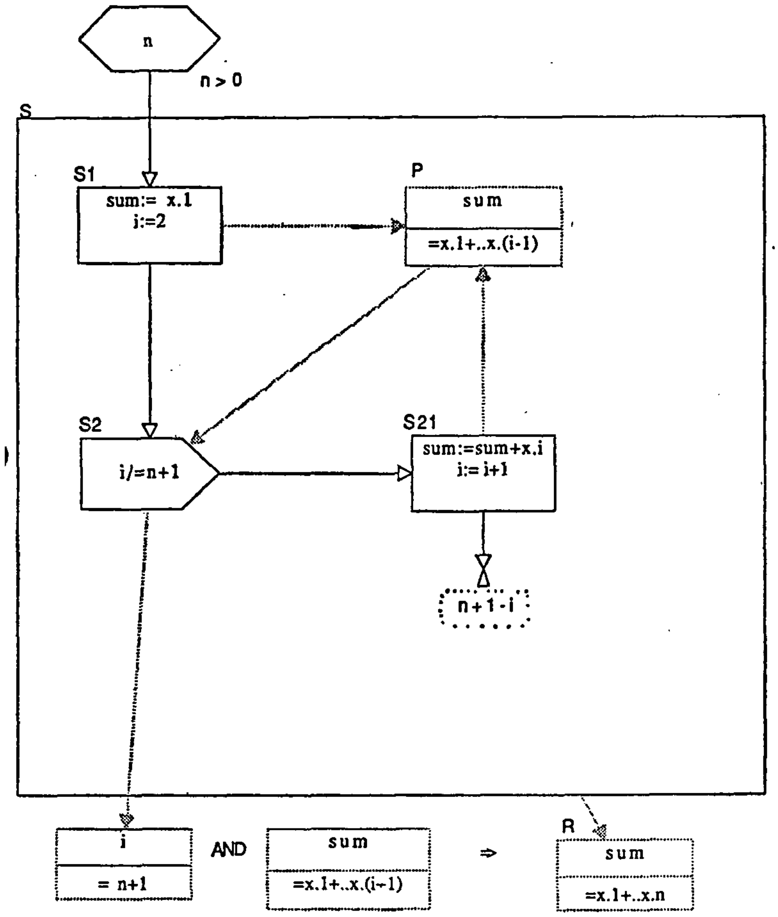
The iteration box is connected to the assertion box below S which shows the falsification of the guard and P. These together give the postcondition R.

In the proof stage a menu PROOF is used with the options shown in the box below:

Prove node **Unprove node**	The proof starts by proving S1 and S21 and then S2. It will be checked that the variant box has text.
Print flowchart on file	After proving S2 the remaining boxes on the parent page may be proved. After this we may use the Print-flowchart-on-file option. The result is shown below.
Quit	

```
S0: Initialize
{n : integer}
if n <= 0 ->
       Se: sum:=1/0
       {sum : overflow}
  [] n > 0 ->
        S: sum:=x.1+..x.n
        {sum =x.1+..x.n}
fi
```

```
S:
S1: sum:= x.1
i:=2
{Invariant: {sum =x.1+..x.(i-1)}
 Variant: n+1-i}
do i/=n+1 ->
      S21: sum:=sum+x.i
i:= i+1
od
{i = n+1 and sum =x.1+..x.(i-1)}
=>{sum =x.1+..x.n}
```


n
n > 0
S
S1
sum:= x.1
i:=2
P
sum
=x.1+..x.(i-1)
S2
i/=n+1
S21
sum:=sum+x.i
i:= i+1
n+1-i
i
= n+1
AND
sum
=x.1+..x.(i-1)
⇒
R
sum
=x.1+..x.n

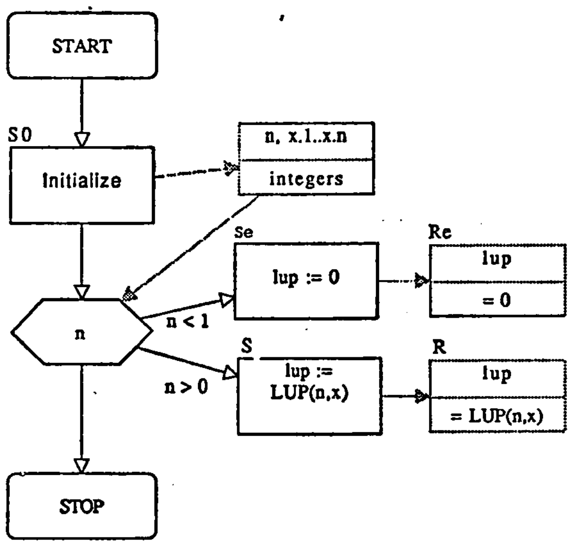
The longest upsequence problem

The length of the longest upsequence of elements given a vector x.1..x.n is to be determined. We use the notation LUP(n,x) for this number. The result of our algorithm is to be stored into the variable lup. Assume further that it is decided that the result for n<1 shall be 0.

A crude design of our algorithm is given in the page to the right. The following variables are used:.

```
integer:
  n = number of elements in the vector
  x.1..x.n = vextor containing the n integers
  lup = the variable to contain the result of the algorithm.
```

The algorithm consists of a conditional statement covering all values of n. For n<1 Se gives lup the value 0 as required, and for n>0 the statement S assigns the correct value to lup. The design is obviously correct providing that LUP(n,x) is computed correctly. The refinement of S is shown on the next two pages.


START
S0
Initialize
n, x.1..x.n
integers
Se
lup := 0
Re
lup
= 0
n
n < 1
n > 0
S
lup :=
LUP(n,x)
R
lup
= LUP(n,x)
STOP

In order to show how to compute LUP(n,x) we need to state some results. Let LSE(i,x) be the longest upsequence ending in x.i. Then

THEOREM 1: $LUP(n,x) = \max_{i \text{ in } I} LSE(i,x)$, where I=[1,n]. Obvious.

Our problem has now been reduced to computing LSE(i,x). For LSE(i,x) the following is valid:

THEOREM 2: $LSE(1,x) = 1$ and $LSE(i,x) \leq i$, $i = 2,\dots,n$. Obvious.

THEOREM 3: Let $n \geq 2$. Then

$$LSE(i,x) = 1 + \max_{j \text{ in } A} LSE(j,x),\ i = 2,\dots, n \text{ if } A \neq \emptyset$$

and

$$LSE(i,x) = 1 \text{ if } A = \emptyset,$$

where

$$A = \{j \mid 1 \leq j \leq i-1 \text{ and } x.j \leq x.i\}.$$

PROOF: For all upsequences ending in x.j, $1 \leq j \leq i-1$ for which $x.j \leq x.i$ the element x.i can be added giving an upsequence one element longer. If $A = \emptyset$ then x.i is the smallest element among x.1,...,x.i and therefore an upsequence ending in x.i consists of just the single element x.i, a sequence whose length is 1.

We use the vector e.1..e.n to store the results of computing LSE(i,x), i = 1..n. The computation of e.i can then be performed as follows:

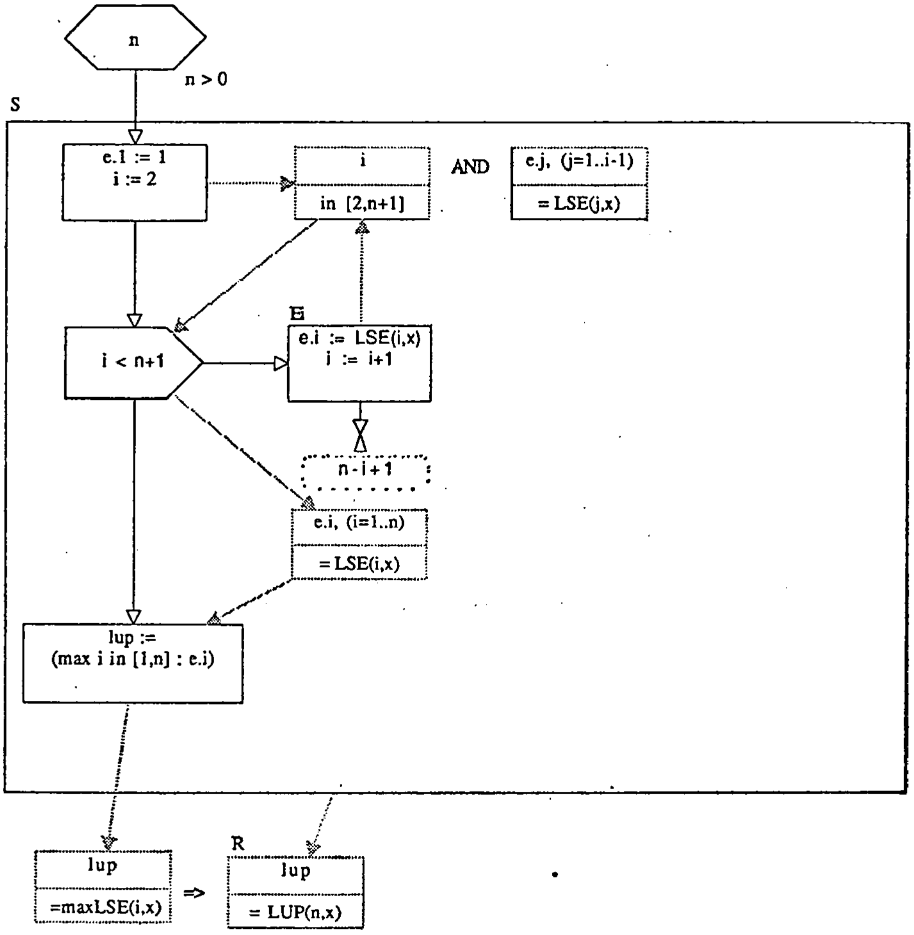
```
   i=1:
e.i  :=  1,

   i = 2..n:
e.i  :=  1  +  $\max_{j \text{ in } A}$  e.j,
```

where $A = \{ j \mid 1 \leq j \leq i-1 \text{ and } x.j \leq x.i\}$. The design of computing the vector e.1.. e.n is shown to the right. The following variables are used:

```
integer:
  e.1..e.n  =  vector  to  store  LSE(i,x)
   i=   index
```

The only nontrivial task is now to compute e.i. The node Ei is therefore refined and its design is shown on the next page.



n
n > 0
S
e.1 := 1
i := 2
i
in [2,n+1]
AND
e.j, (j=1..i-1)
= LSE(j,x)
E
i < n+1
e.i := LSE(i,x)
i := i+1
n - i + 1
e.i, (i=1..n)
= LSE(i,x)
lup :=
(max i in [1,n] : e.i)
lup
=maxLSE(i,x)
=>
R
lup
= LUP(n,x)

The computation of e.i given e.1..e.(i-1) is shown to the right. The set B in the loop invariant is

B={j | 1≤ j ≤ k-1 and x.j ≤ x.j}.

For k=i, B is equivalent to A.

The following variables are introduced:

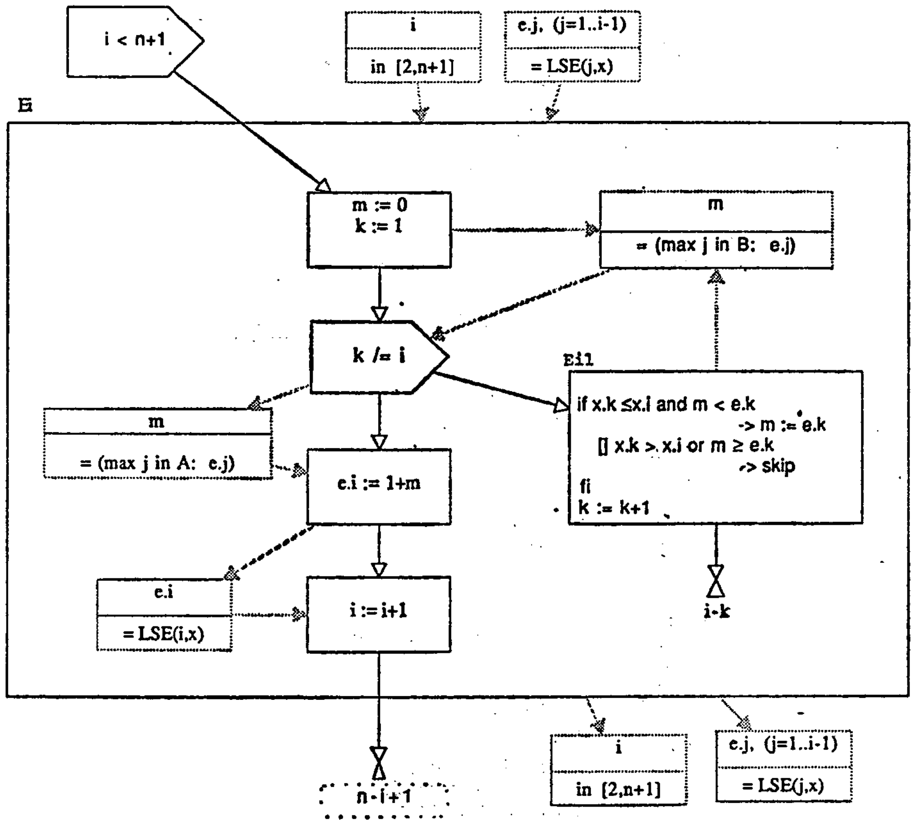
```
integer:
m: used to store (max j in B: e.j) for k = 1..i-1
 k   =   index.
```

Note that in the box Ei1 code is written instead for showing the design in the form of a flowchart. By utilizing this feature the trivial parts can be written more condensed and only parts where formal reasoning is of help are shown as charts. This possibility makes it possible to use the tool in a very flexible way and may therefore suit different tastes of programming.

When the design is ready an explicite proof stage is entered by quitting the PICA Flowchart and choosing the PICA Proof Add On from the apple menu. An unproved box has a thicker border line. Proving the correctness of a node is done by klicking on the node. The prover then checks that subordinate nodes are proved and that the node has necessary pre- and postconditions.

It is supposed that by separating the design part and the proof part the user will have a better chance of finding an error than if both tasks are interviened.

When the design is proved correct one may choose the Print Flowchart on File option from the Poof menu. This will give a skeleton program consistent with the design given in the flowcharts. This program may then be completed by further editing. The skeleton program resulting from the design presented in this example is shown on the next page. An edited running version in Simula is also presented.



i < n+1
i
in [2,n+1]
e.j, (j=1..i-1)
= LSE(j,x)
E
m := 0
k := 1
m
= (max j in B: e.j)
k /= i
Ei1
if x.k ≤x.i and m < e.k
-> m := e.k
[] x.k > x.i or m ≥ e.k
-> skip
fi
k := k+1
m
= (max j in A: e.j)
e.i := 1+m
e.i
= LSE(i,x)
i := i+1
i-k
n-i+1
i
in [2,n+1]
e.j, (j=1..i-1)
= LSE(j,x)

```
S0: Initialize
{n, x.1..x.n  integers}
if n < 1 ->
      Se: lup := 0
          {lup  = 0}
 [] n > 0 ->
      S: lup := LUP(n,x)
         {lup  = LUP(n,x)}
fi
```

```
S:
e.1 := 1
i := 2
{Invariant: {i  in [2,n+1] and e.j, (j=1..i-1)  = LSE(j,x)}
 Variant: n-i+1}
do  i < n+1  ->
      Ei:  e.i := LSE(i,x)
           i := i+1
od
{e.i, (i=1..n)  = LSE(i,x)}
lup :=
(max i in [1,n] : e.i)
{lup  =maxLSE(i,x)}
=>{lup  = LUP(n,x)}
```

```
Ei:
{i  in [2,n+1] and e.j, (j=1..i-1)  = LSE(j,x)}
m := 0
k := 1
{Invariant: {m  = (max j in B:  e.j) }
 Variant: i-k}
do  k ≠ i  ->
        Ei1:  if x.k ≤x.i and m < e.k
                        -> m := e.k
           [] x.k > x.i or m ≥ e.k
                        -> skip
           fi
           k := k+1
od
{m  = (max j in A:  e.j) }
 e.i := 1+m

{e.i  = LSE(i,x)}
 i := i+1
{i  in [2,n+1] and e.j, (j=1..i-1)  = LSE(j,x)}
```

```
integer procedure lup (n, x); integer n; integer array x;                comment
                                   ** lup = length longest upsequence of x.1...x.n;
⟨n, x.1...x.n integers⟩                                                          ;
begin
  integer procedure plup (n, x); ... ;
  if n lt 1  then                                                          comment
      Se; lup:=0;               comment ⟨lup = 0⟩;
  if n gt 0  then                                                          comment;
      S ; lup:=plup (n,x);      comment ⟨lup = LUP(n,x)⟩;
end;
  integer procedure plup(n, x); integer n; integer array x;
  begin
    integer procedure plse (i,x,e); ... ;
    integer array e(1:n);
    integer i, max;
    e(1):=1;
    i   :=2;                                                               comment
    ⟨Invariant: ⟨i in (2,n+1) and e.j, (j=1...i-1) = LSE(j,x)⟩
    Variant : n-1+1⟩                                                             ;
    while i lt n+1 do                                                      comment
        Ei; begin e(i):=plse (i,x,e);
                  i:=i+1
            end;                                                           comment
    ⟨e.i, (i=1,, n) = LSE(i, x)⟩                                                 ;
    max:=e(1);
    for i := 2 step 1 until n do if max lt e(i) then max:=e(i);
    plup:= max;                                                            comment
    ⟨plup = max LSE(i,x)⟩                                                        ;
end;
    integer procedure plse (i,x,e); integer i; integer array x, e;
    begin                                                                  comment
      ⟨i in (2, n+1) and e.j, (j=1...i-1) = LSE(j,x)⟩                            ;
      integer m, k;
      m:=0;                                                                comment
      k :=1;
      ⟨Invariant: ⟨m = (max j in B: e.j)⟩
      Variant : i-k⟩                                                             ;
      while k ne i do                                                      comment
          Eil; begin if x(k) le x(i) and m lt e(k)
                                        then m:=e(k);
                  k:=k+1
              end;                                                         comment
      ⟨m = (max j in A: e.j)⟩
      plse:=1+m;                                                           comment
      ⟨plse (i,x,e) = LSE(i,x)⟩                                                  ;
  end;
```

MICROTEST — A Testing Tool on PC*

ISTVÁN FORGÁCS, ATTILA HORVÁTH and ENDRE SOMOS

Computer and Automation Institute
Hungarian Academy of Sciences
H—1518 Budapest, Pob. 63 Kende u. 13—17, Hungary

Abstract

The MICROTEST system is described, which is a tool for decentralized "distributed" testing of application programs targeted to run on large mainframe computers. It detaches the testing process from the host environment, thus the programs can be tested on PC. The system includes a high-level test language for validating program results against the specification. Unlike the other testing tools, it does not use instrumentation but it compiles the source program into an object code which is interpreted. (This is the key to the transportation of testing to PC.) MICROTEST has five components. The Static Analyzer compiles the source program into an intermediate language, and produces data for a series of quality metrics. The Static Report Generator makes the Static Usage, the Branch and the Module Quality reports. The Assertion Compiler compiles the test program into the same intermediate language as the one used by the Static Analyzer. The Dynamic analyzer links the compiled programs and interprets it. The Dynamic Report Generator makes the Test Log, the Dynamic Test Path, the Data Coverage, and the Program Coverage reports.

Introduction

In the last few years the theory and practice of program testing came into prominence. The reason of the research is in the recognition of the fact that maintenance cost is the strongest component of software development expencies. Since error correcting is about two third of the entire cost of maintainability, the great effort seems intelligible [1, 2, 3].

There is a lot of testing techniques in the theory, but in the practice only three of them are used:

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

— structural testing,
— functional testing,
— code reading.

In case of structural testing, the program is investigated as a white box. The test cases are created based on the flow-graph of the program to reach a high program coverage ratio. Many criteria for program coverage were suggested from statement testing to path testing. The partial ordering of these strategies are in [4, 5]. In [4] a worst case estimation is given for the number of test cases.

In case of functional testing the program is investigated as a black box. The test cases are gathered from the specification and from other user information named 'oracle' [6, 7].

In case of code reading the function of the program is determined manually and compared against the specification. Although this method seems the less effective, experience shows that code reading sometimes can do better then the two others [8].

The development of testing tools

Most of the tools are based upon instrumentation. During instrumentation, traps or counters are inserted into the source code to measure the program coverage. These tools work on the basis of the structural testing method.

In case of earlier tools the test data was typed manually from the keyboard, while the result was read from the screen. The inserted counters measured the test coverage.

The next step was the automatic validation of the program. The validation procedures (assertions) were built into the program conditionally, thus the program could be executed with or without validation. (E.g. in [9], assertions were written as special comments, and an optional preprocessor compiled them into the source code.) This way the output can be validated but the input is not a basic component of the validation system (especially in the case of keyboard input). These assertion instructions were the predecessors of the test languages.

Since one of the essential requirements of testing is the reproducibility of tests, it is necessary to store the test data for all the test cases. This is done in some testing tools (e.g. in [10] the inpu tis received from the keyboard, it is recorded in a file, and when the test is repeated, this file substitutes the input device).

The last generation of testing tools detaches the description of test data (both input and output) from the program. It provides an opportunity to generate the test data from the specification itself (either manually or automatically) even before the implementation of the program. This method summarizes the advantages of structural and functional testing.

An example is the assertion language of the SOFTEST system [11] which uses first order predicate logic to define the program specification. The program behavior is specified in terms of PRE and POST assertions for the data states and INPUT/OUTPUT assertions for the data base file accesses. The test data is generated from the PRE and INPUT assertions and the test results are checked against the POST and OUTPUT assertions. All the assertions can be given in the forms of individual values, sets, ranges, functions, and relations. The SOFTEST system instruments the object code with assembly routines to execute the assertions. Directly the program variables are used, therefore the object code of the program should be handled by the testing system, which decreases the portability.

Now we can summarize the requirements of a good test system:

— the input and output test data should be stored separately from the program under test,

— the test data should be described with the help of a high level test language which supports all the conditional and cyclic data assignments,

— both the program under test and the test program should be compiled and linked into one executable or interpretable code,

— the system should be independent from the hardware environment,

— both the executed program and all kinds of data should be under complete control of the test system.

The MICROTEST system

The MICROTEST is a module test system which can test and debug COBOL programs on PC-s. The system contains two compilers to translate both the COBOL and the assert program into the same binary code named AL (assembly like) language. In this AL language the commands are coded binary and the Dynamic Analyzer interprets the commands. For example the COMPUTE $p3=p1+p2$ statement is translated to ADD $p1, p2, p3$ which are four integers in the command table. The first integer stores the code of the addition, the others store the pointers to the $p1$, $p2$ and $p3$.

The system consists of five elements:

— Static Analyzer,
— Static Report Generator,
— Assertion Compiler,
— Dynamic Analyzer,
— Dynamic Report Generator.

The Static Analyzer has two functions. First, it compiles the source code into the AL; second, evaluates the source program to produce data for a series of analytical reports and a number of quality metrics. The Static Analyzer differs from the others not only in compiling the program but in the instrumentation, too. It is needless to instrument the source code. During the static analysis, each I/O and external function calls are translated into an assertion call which calls the appropriate assertion program module. The Static Analyzer was made with the PROFLP compiler generator which was very useful in making the compiler part of the static analyzer for different COBOL versions.

The Static Report Generator produces the following reports:

— Module Quality Report (see TABLE I),
— Static Data Usage Report (see TABLE II),
— Branch Report (see TABLE III).

TABLE I

MODULE QUALITY REPORT		
Module name: CMERGE	Date: 12. 5. 1988	Page: 1;

Static Metrics:	Data Complexity	= 0.31
	Control Flow Complexity	= 0.92
	Data Flow Complexity	= 0.33
	Interface Complexity	= 0.27
	Portability	= 0.78
	Maintainability	= 0.46
	Testability	= 0.31

TABLE II

STATIC DATA USAGE REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Data no.	Lv no	Data Name	Strg Type	Data Type	Data Lng.	Dim	Picture	Data usage					
			file										
1	1	REC—1		disp	10				a		t		
2	2	KEY—1		disp	3		X(3)	p		r			
3	2	SAL—1		disp	7		9(5)V99		a				
4	1	REC—2		disp	10				a		t		
5	2	KEY—2		disp	3		X(3)	p		r			
6	2	SAL—2		disp	7		9(5)V99		a				
7	1	OUTREC		disp	10					r	t		
8	2	KEY—3		disp	3		X(3)						*
9	2	SAL		disp	7		9(5)V99						*
			work										
10	1	W		disp	17								*
11	3	SUM—SAL		disp	10		9(8)V99		a	r			
12	3	MAX—SAL		disp	7		9(5)V99	p	a	r			
13	77	N		comp3	3		9(5)		a	r		i	
14	77	PAR—SAL		disp	7		9(5)V99	p	a	r			
15	77	DISP—RESULT		disp	8		ZZZZ9.99		a	r			
			link										
16	1	LINK—DATA		disp	8								*
17	2	PAR		disp	1		A	p	a				
18	2	RESULT		disp	7		9(5)V99		a	r			

Number of predicates:	5	Number of arguments:	11
Number of results:	9	Number of transients:	3
Number of inits:	1		

Total number of data items used: 14
Total number of data items not used: 4

The Module Quality Report computes the following static metrics:
— data complexity,
— control flow complexity,
— data flow complexity,
— interface complexity,
— maintainability,
— testability.

The Static Data Usage Report contains information on the data fields which can be gained from the source code: data names, level number, data length, data type, dimension, picture, and data usage. A data can be predicate, argument, result, transient (parameter of an I/O or CALL statement), or it can get an initial value. A '*' character indicates when the data is not used at all.

The Branch Report describes the control flow of the program under test. The branches are identified by a branch number. The report contains the line number, the branch number and the source program lines.

The Assertion Compiler compiles the assertions into the AL language. The original SOFTEST language was developed to a high level language. In our system the assertion variable declaration, FOR, REPEAT—UNTIL, WHILE—DO, and CASE statements permit the sophisticated input and output test data assignment/validation. An example:

```
FOR $i = 1 TO 10 DO
    ASSERT IN abc[$i] E SET(1, 2, $i)
END;
```

At the first call the ten array elements of *abc* get the value 1, at the second call get 2, while at the third call *abc*[1] = 1, ..., *abc*[10] = 10.

The compiler has a built-in editor too.

The Dynamic Analyzer first links the compiled COBOL and test programs then interprets the linked code. This way the Dynamic Analyzer contains a driver, which interprets the object code and produces all the necessary statistics. During the run it reads the test data either from the assert file or from the keyboard, and validates the results against the data read from the assertion file. It produces statistics in order to obtain some dynamic metrics which qualify the program and the testing process.

The MICROTEST Dynamic Analyzer operates in an interactive dialogue mode; the user has the opportunity

— to stop the run at any moment,
— to take breakpoints into the program,
— to use step-by-step execution,
— to display/change any data item.

Though these are debugger functions, they can help the testing, too.

Whenever a validation error occurs (the actual data differs from the prescribed one), an assertion violation interrupt is executed. The error message contains both the COBOL and the assertion line number, the data name as well as the prescribed value or interval, and the actual value.

TABLE III

STATIC BRANCH REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 2

Line no.	Brch. no.	PROCEDURE DIVISION statement
93A	26	✻✻✻ EMPTY BRANCH for statement in line 92 ✻✻✻
94		✻
95	27	FILE—READ—2.
96	27	MOVE SAL—2 TO PAR—SAL.
97	27	PERFORM SAL—COMP.
98	27	WRITE OUTREC FROM REC—2.
99	27	READ INFILE—2
100	28	AT END MOVE ALL HIGH—VALUES TO KEY—2.
100A	29	✻✻✻ EMPTY BRANCH for statement in line 99 ✻✻✻
101		✻
102	30	SAL—COMP.
103	30	IF PAR — „A”
104	31	ADD PAR—SAL TO SUM—SAL
105	31	ADD 1 TO N
106		ELSE
107	32	IF PAR—SAL>MAX—SAL
108	33	MOVE PAR—SAL TO MAX—SAL.
108A	34	✻✻✻ EMPTY BRANCH for statement in line 107 ✻✻✻
109	35	FINISH SECTION.
110	35	STOP RUN.

Total number of statements: 51
Total number of branches: 35

TABLE IV

DYNAMIC TEST PATH REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Testcase	Branch Start Statements	Date	Time
1	51 52, 53A, 55, 59, 62, 64, 65, 67, 83, 85, 88, 102, 104, 65, 67, 83, 87, 95, 102, 104, 100, 100A, 65, 67, 83, 85, 88, 102, 104, 65, 67, 83, 85, 88, 102, 104, 93, 93A, 65, 68A, 69, 75, 78, 100	12. 5. 1988	12:39:21
2	51, 52, 54, 55, 59, 62, 64, 65, 67, 83, 87, 95, 102, 107, 108, 65, 67, 83, 87, 95, 102, 107. 108 100, 100A, 65, 67, 83, 85, 88, 102, 107, 108A, 93, 93A, 65, 68A, 69, 77, 78, 100, ✻ Error message: Assert violation	12. 5. 1988	12:41:19

TABLE V

PROGRAM COVERAGE REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Branch no.	Start stmt.	Total execution	Last execution	Not executed
1	51	3	1	
2	52	3	1	
3	53A	2	1	
4	54	1	0	
5	55	3	1	
6	58	1	1	
7	58A	1	1	
8	59	3	1	
9	61	1	1	
10	61A	1	1	
11	62	3	1	
12	64	3	1	
13	65	10	1	
14	67	7	0	
15	68A	3	1	
16	69	3	1	
17	75	1	0	
18	77	1	0	
19	78	2	0	
20	82	0	0	*
21	83	7	0	
22	85	4	0	
23	87	3	0	
24	88	4	0	
25	93	2	0	
26	93A	2	0	
27	95	3	0	
28	100	2	0	
29	100A	2	0	
30	102	7	0	
31	104	4	0	
32	107	3	0	
33	108	2	0	
34	108A	1	0	
35	110	2	0	

Total number of branches:	35
Number of branches executed:	34
Program coverage ratio:	97.14%

Dynamic Metrics:

Reliability	= 0.33
Integrity	= 0.11
Test Coverage	= 0.79

TABLE VI

DATA COVERAGE REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Data nr.	Lv. nr.	Data Name	Static Usage					Dynamic Usage				
			Prd	Arg	Res	Trn	Ini	Pre	Inp	Post	Out	N. A.
		FILE SECTION										
1	1	REC—1		a		t						*
2	2	KEY—1	p		r				in			
3	2	SAL—1		a					in			
4	1	REC—2		a		t						*
5	2	KEY—2	p		r				in			
6	2	SAL—2		a					in			
7	1	OUTREC			r	t						*
8	2	KEY—3									out	
9	2	SAL									out	
		WORKING—STORAGE SECTION										
10	1	W										*
11	3	SUM—SAL		a	r							*
12	3	MAX—SAL	p	a	r							*
13	77	N		a	r		i					*
14	77	PAR—SAL	p	a	r							*
15	77	DISP—RESULT		a	r							*
		LINKAGE SECTION										
16	1	LINK—DATA										*
17	2	PAR	p	a				pre				
18	2	RESULT		a	r					post		

Total number of data items (without Filler-s): 18

Number of predicates:	5	Number of PRE	asserted data:	1
Number of arguments	11	Number of INPUT	asserted data:	4
Number of results:	9	Number of POST	asserted data:	1
Number of transients:	3	Number of OUTPUT	asserted data:	2
Number of units:	1	Number of NOT	asserted data:	10

Data coverage ratio: 61.54%

The dynamic report generator produces the following reports:

— Test Log,
— Dynamic Test Path Report (see TABLE IV),
— Program Coverage Report (see TABLE V),
— Data Coverage Report (see TABLE IV).

Test Log contains everything which was displayed during the test run including all the assertion violations.

The Dynamic Test Path Report describes the test paths of each test case run executed by the Dynamic Analyzer. The report contains the test case number and the executed program path by printing the branch numbers in the same order as they were executed.

The Program Coverage Report is a table of program branches with their number of traversions since testing began. Branches which were not traversed are marked, this way new test data can be created to cover them, or unexecutable paths of the program can be revealed.

The Data Coverage Report describes the behavior of data items during the test run. It repeats some entries of the Static Data Usage Report to help the user to compare the static and dynamic results. The report describes the relation between the data items and the assertions. It indicates which data was input and/or output asserted or which data was not used at all.

The testing process using MICROTEST

The best known structural testing strategies are segment, branch and path testing. Segment testing requires each statement to be executed at least once during the test. Branch testing requires each branch (including the empty branches too) to be executed at least once. Path testing requires that all the paths in the program to be tested by at least one test case.

We chose the branch testing strategy because the segment testing is not effective enough [8] while the others require $O(n^2)$ [4] or more test paths in worst case, where n is the number of branches. (Branch testing requires $O(n)$ test paths.)

The process is the following: the programmer develops the COBOL program, and the tester develops the test program, independently. The tester selects test data from the specification. Then the Static Analyzer compiles the COBOL module, while the Assertion Compiler compiles the test program. The Dynamic Analyzer links the object codes and interprets it. If there are assertion violations, then either the COBOL or the test program should be corrected and compiled again.

In lack of assertion violations, the reports are investigated. If all the branches are covered then the testing is over, else new test data should be selected from the structure of the program. The new test program should be compiled again, and the process goes on until there are no assertion violations, and the branch coverage is acceptable.

Conclusions

We reviewed the development of test systems, and described the requirements of a high effective testing tool. The MICROTEST system is a result of these specifications. In the test data selection, both the functional and the structural methods are present.

The test data are separated from the program, so the test run can be repeated. At program execution, however, both the program and the test data are in a single interpretable binary program code. Since the Static Analyzer compiles the source code into the AL code, it is needless to instrument the source. Moreover, each module can be executed independently from the others since the assertion program simulates the calling and the called module. Thus bottom-up, top-down, and mixed testing strategies can be used as well.

The thorough evaluation of reports (especially the static and the coverage information) can significantly improve the program quality. The built-in debugger functions help program development.

The authors wish to express their gratitude to Mr. Harry Sneed, who has initiated the MICROTEST project and was the source of many ideas included in the system.

References

[1] M. V. Zelkowitz, Perspectives on Software Engineering, Computing Surveys, Vol. 10, No. 2, pp. 197—260, 1978.
[2] D. S. Alberts, "The Economics of Software Quality Assurance," Proc of National Computer Conference 1976, pp. 433—442.
[3] P. J. Smith, "The Requirement for Quality in the Design of Programming System," Proc. of Pragmatic Program and Sensible Software Conf., pp. 491—508, 1978.
[4] S. C. Ntafos, A "Comparison of Some Structural Testing Strategies," IEEE Trans. Software Eng., vol. 14. no. 6, pp. 868—875, June 1988.
[5] M. D. Weiser, J. D. Gannon, and P. R. McMullin, "Comparison of Structured Test Coverage Metrics," IEEE Trans. Software Eng., vol. 2. no. 2, pp. 80—85, March 1985.
[6] W. E. Howden, Functional Program Testing, IEEE Trans. Software Eng., vol. 6. no. 3, pp. 162—169, May 1980.
[7] W. E. Howden, "The Theory and Practice of Functional Testing," IEEE Software vol, 2. no. 5, pp. 6—17, Sept. 1985.
[8] W. R. Basili, R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies, "IEEE Trans. Software Eng., vol. 13. no. 12, pp. 1278—1296, Dec. 1987.
[9] J. C. Huang, "Program Instrumentation and Software Testing," IEEE Computer vol. 11, pp. 25—32, April 1978.
[10] T. J. McCabe, G. G. Schulmeyer, "System Testing Aided by Structured Analysis: A Practical Experience," IEEE Trans. Software Eng., vol. 11. no. 9, pp. 917—921, Sept. 1985.
[11] M. Majoros, H. M. Sneed "The Softests Program Test System," Systems and Software vol. 2. pp. 289—296, 1982.

Subscription information:

For Albania, Bulgaria, China, Cuba, Czechoslovakia, German Democratic Republic, Korean People's Republic, Mongolia, Poland, Romania, USSR, Vietnam
orders should be addressed to:

Kultura
Hungarian Foreign Trading Co.
H—1389 Budapest 62
P. O. Box 149
Hungary

For all other countries orders should be adressed to:

J. C. Baltzer AG
Scientific Publishing Company
Wettsteinplatz 10
CH—4058 Basel
Switzerland

Mailing address for editorial correspondence:

Acta Cybernetica
Árpád tér 2.
Szeged
H—6720 Hungary

INDEX — TARTALOM

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Gécseg Ferenc
A kézirat a nyomdába érkezett: 1990. február
Terjedelem: 13,75 (A/5) ív
Készült monószedéssel, íves magasnyomással
az MSZ 6601 és az MSZ 5602—55 szabvány szerint
90-703 — Szegedi Nyomda — Felelős vezető: Surányi Tibor igazgató