SETL/E
A Prototyping System Based on Sets

Ernst-Erich Doberkat
Ulrich Gutenbeil
Wilhelm Hasselbring

University of Essen
Computer Science/Software Engineering
D-4300 Essen 1
Germany

October 5, 1990

Abstract

A CASE tool must, by definition, be a tool for engineering software for supporting the entire life cycle — from requirements analysis to maintenance. In this paper we will present approaches to prototyping and transformational programming that is capable of complementing this software life cycle for overcoming the well known deficiencies of this classical model of software production. After introducing the concepts and principles we will discuss **SETL/E** as a language for realizing these approaches. Finally we will summarize the actual and future work on this prototyping system.

1 Introduction and overview

One of the more recent approaches for complementing the classical model of software production using the life cycle approach is Rapid Prototyping. Having a look at the literature it seems that Rapid Prototyping is used as an umbrella notion for a multitude of activities, and it is not always too easy to find some sort of common denominator. We consider a prototype as a model exhibiting the essential properties of the final product. Thus a prototype is a model, and this model has to be executable as a program so that at least part of the functionality of the desired end product may be demonstrated.

Prototyping has been developed as an answer to deficiencies in the waterfall model, but it should not be considered as an alternative to this model. It is rather optimally useful when it complements the waterfall model. The definition given above makes it plausible that prototyping may be used during the early phases of the design. Dearnly and Mayhew [2] suggest to augment the analytic first phase for the construction of software with the components planning and definition of requirements by a prototyping phase. It is rather evident, however, that the effectiveness of prototyping exceeds the first phase in the waterfall model considerably.

Our approach to prototyping is an evolutionary development in versions. This is due to the fact that the developed product may manipulate the environment in which it is used. Thus the prototype has to evolve in accordance with the changing environment. The linear ordering of development steps in the classical waterfall model is mapped here into successive development cycles. This implies that users are involved in the system development process which supports the communication between users and developers.

This approach is still compatible with the phase-oriented software production model. It primarily affects the implementation phase, but is based on the overall design. However the maintenance phase as such is

replaced by further development cycles based on the existing system version and new requirements. The prototyping environment has to be integrated in the production environment to justify this approach. In the next section we will discuss this in more detail.

Prototypes should be built in very high-level languages to make them rapidly available in the early phases of the production process. Our approach rests on the hypothesis that set theory is a natural vehicle for expressing one's thought when developing programs, since most algorithmic problems may easily be formulated in terms of sets and maps. Such a prototype utilizing constructs from set theory directly is usually not a very efficient program since the runtime system has a lot of things to do for executing the highly expressive constructs. To obtain a more efficient production level version program transformations are needed.

As a first step it is possible to apply dialect transformations in a broad spectrum language for transforming the high level specifications to a level of e.g. Ada whereby deeply nested set theoretic constructs have vanished. The set-less representation is then transformed by crossing a language boundary into a language like C. This approach also supports the reuse and verification of software products. We will discuss these aspects in more detail in the third section.

In the final sections we will present the Set Theoretic Language/Essen (SETL/E) as a language for prototyping and transformational implementation and the actual and future work on this system.

2 Prototyping: Modeling Programs and Data

The classical model of software production using the *life cycle* approach has severe deficiencies indicating the desirability of complementing this model by other approaches. One of the more recent approaches for this is *rapid prototyping.* This term denotes a multitude of activities, judging from its use in the literature, and it is not always too easy to find some sort of common denominator, see [10, 1, 5]. We stick here to Christine Floyd's definition given in [7], according to which prototyping refers to the welldefined phase in the production process of software in which a model is constructed which has all the essential properties of the final product, and which is taken into account when properties have to be checked, and when further steps in the development have to be determined. We want to record for later use that a prototype is a model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer. Thus software prototypes are distinguished as models from other models in engineering: the clay model of a bridge is a model but certainly no model that allows demonstrating part of the functionality of the final product.

Floyd's definition shows moreover that prototyping can be used to overcome the disadvantages of the classical model since a model can be easier manipulated than a production program. In particular it is possible to proceed in an explorative way by binding the properties to be tested to the model and then evaluate it. A model can grow so that an evolutionary way of program development is possible, and finally it is not required by the tentative nature of a prototype that all requirements are fixed already. These rather general statements will be substantiated by discussing some variants to prototyping.

We would like to point out here that we discuss only prototypes which are executable. This contrasts to the approach used for example in the CIP-project [9] in which specifications are used which may contain non-effective components and which are consequently not necessarily executable.

Following Floyd, we will develop prototyping along two orthogonal axes. One axis precises the functionality to be modeled, the second axis describes the intentions pursued by the prototype in greater detail. The functionality can be horizontal oder vertical, the intentions may be classified through the categories "experimental", "evolutionary" and "throw away".

A *vertical* prototype realizes selected functionalities in each and every detail (in a way which would be done in a production program). All other functionalities are only sketched, and this is done usually only in so far as it is necessary for the proper functioning of the prototype. Vertical prototyping is apparently useful when carefully selected functions are studied in order to make statements about their behaviour. A large part of the activities attributed by folklore to prototyping, viz., constructing user interfaces falls into this

category of vertical prototyping: here emphasis is put on constructing the interface in all details while the data processing functions proper are usually neglected or only sketched.

A *horizontal* prototype implements all those functionalities which the final product is to realize. But this does not happen in a way which is suggested for the final version but rather as a model (so that a horizontal prototype may be composed recursively from prototypes for the individual functionalities). This kind of prototyping is most useful as an approach whenever the complete program to be constructed is at one's disposal, hence when principle questions regarding the entire design have to be answered.

While the characterization as horizontal or vertical prototypes makes use of the way the diverse functionalities of the prototypes are realized the classification in an evolutionary, experimental or throw away cares more about the way the prototype is constructed and the way it is used later on.

The bitter fate of a *throw away* prototype is characterized by its name: the purpose here is a practical demonstration of possible system functions where feasibility is emphasized so that this approach has a strong explorative component. Prototypes which have been developed in this way are rather well suited for complementing the early phases in the life cycle model since they may be used for stabilizing requirements.

While a throw away prototype emphasizes exploration in order to be able to discuss desired properties and to try alternative approaches the *experimental* prototype focusses on the investigation of solutions with respect to its adequacy. We see here — as Floyd does — some variants in the functions satisfied by the prototypes: it may supplement the specification, refine parts of the specification (coming quite close to the vertical approach), or even mean an intermediate step from the specification to an implementation.

Evolutionary prototypes grow in different versions till they stabilize and have reached a fixed point. They may be classified according to whether they are incremental oder evolving. Both classes are subjected to a cyclic approach. For the construction of an incremental prototype one starts with the first incomplete solution and widens the solution stepwise to a complete one. Evolving solutions are usually constructed using a cycle consisting of design, implementation, evaluation which is observed until the solution satisfies the requirement. This kind of prototyping should be used in all situations in which effects coming from the environment have to be taken into account: a prototype is brought into the working environment for the end product, possibly changes this environment, and consequently forces changes in its specifications or requirements. This then requires changes in the prototype which is brought into the working environment, is evaluated and so on.

It is essential and common to all approaches that they can work only under essential cooperation of the user. Considered as a process it is important for prototyping to seek the consensus between the developer and the user. In this way the user is given an opportunity to essentially influence the development of the final product — this is a marked contrast to the distance which may be observed between a user and the development process in the waterfall model. In prototyping the developer obtains important insights into the user' s problem sphere which may soften the developer's estrangement to his product and allow for a custom-tailored solution. Thus prototyping generates learning effects mostly by feed-back.

The process of modeling applies to programs as well as to data: in the process of developing an application the algorithms have to be explored, but the data and data structures on which the algorithms are to work may emerge from this explorative activity as well. Semantic data models working with objects, attributes, and *ISA*-relationships investigate ways of modeling data according to their semantic contents (cp. [11]). They are used for designing record-oriented schemata where the approach is somewhat similar to the one used in software prototyping, but rather than modeling programs high-level representations of data are modeled. This model is mapped into a lower-level structure (see [11], 1.4). Khoshafian and Briggs point out that data modeling should accomodate the user by making the representation and manipulation as close as possible to the user's perception of the problem (cf. [12], p. 606). Hence it is desirable to

- model data according to the user's needs,
- iteratively refine data representations (which requires access to previously formulated data models),
- reuse patterns or templates of previously formulated data models,

- share data either between different users and different prototyping sessions.

We see that there are in fact striking similarities between prototyping programs and modeling data. Both construct a model to be experimented with and eventually to be transformed into a production version. Thus it is valuable to have a programming language which is able to serve both sides,

- the software engineer who wants to model programs
- the data engineer who wants to construct a semantic model of her data.

Consequently, a prototyping language needs the ability to incorporate semantic data models. Thus it will be possible to formulate data on a very high level for modeling purposes, and it is simply a matter of economy to make data persistent: once data are modeled it is not necessary to compute them each time they are used. A related concern for reusing data comes from the observation that more than one program may want to access them. Thus one program may generate data and another one may want to access these data.

Once the protoype becomes stable, it may be transformed into a production program, see 3. The data which have been modeled using the prototype, however, are usually not affected by this transformation. Thus we may experience the situation that we have high-level data structures formulated in a prototyping language, following its data structuring principles and accessible in binary form in it, but not acccessible in the production language. Consequently, reusability of code may be intertwined with reusability of data. Reusing code by program transformations ought to be complemented by a method of making data reusable by transformations. A first attempt to solving this problem in the context of transforming **SETL** programs to **Ada** may be found in [14].

We will discuss our approach in the context of **SETL/E**, (see [6]) a weakly typed prototyping language supporting sets, maps, tuples, and procedures as the generic data structuring facilities in the tradition of **SETL** ([15, 5]).

3 Transformational Programming

SETL/E is a wide spectrum language which allows programming close to the conceptual ideas of a solution as well as programming quite close to the machine on a level comparable to Ada. Prototypes are usually formulated on a very high expressive level, since they may make use of all the expressive and notational power of finite set theory. This makes for convenient problem solving, but it is also to be blamed for slowly executing programs. The gap between prototypes and production efficient programs written in a production language is to be bridged by transformations. Here a two-step procedure is observed: first transformations are performed within the language, thus lowering the expressive level towards the production end, then a transformation (or better: a translation) produces the production language program. Since the latter translation is essentially well understood (it resembles in some ways the production of intermediate code in a compiler once types have been inferred), we will briefly discuss transformations that do not cross the language boundary.

The most important class of transformations is based on Paige's method of finite differencing. This is a variant of the technique of strength reduction used in optimizing compilers for replacing expensive operations by cheaper ones. Strength reduction works by establishing and maintaining invariant relations when objects local to a loop, and in a similar way invariants are maintained by finite differencing.

We give a simplified definition of differentiability; for a full account see [13], and for a tutorial discussion [5]. Let $E = f(x_1, \ldots, x_n)$ be an applicative expression, and dx_i a modification to x_i. Then two code blocks $[B_1, B_2]$ are called the *derivative* of E w.r.t. dx_i iff the equality

$$E = f(x_1, \ldots, x_n)$$

is an invariant for the code block

$$B_1; dx_1; B_2$$

(thus the equality $E = f(x_1, \ldots, x_n)$ holds before and after performing the compound code block; note that the equality after executing the block works with the new value of x_i). The block B_1 (the prederivative) is used to manipulate the value of E for the old value of x_i, and B_2 (the postderivative) is used correspondingly for the new one. The differential of $\partial E < B >$ of E w.r.t. a code block B is computed as follows

1. replace each definition dx_i in B by the block $B_1; dx_i; B_2$
2. replace each occurrence of $f(x_1, \ldots x_n)$ in the code block generated by step 1 by the variable E.

Consider for example the expression

$$E := \{x \in S \mid F(x) = t\}$$

for a set S and a map F from S to the integers in which S does not occur free. Adding a new element x to S results in the following block B_1:

$$\textit{if } F(x) = t \textit{ then } E := E \cup \{x\}; \textit{fi};$$

and Block B_2 is empty. Thus maintaining the equality $E = \{x \in S \mid F(x) = t\}$ by recomputing the latter set is replaced by the test $(F(x) \stackrel{?}{=} t)$ and by an insertion, if necessary. Thus a substantial amount of computing time may be saved, as the following example shows. Specialize $F(x) = t$ to $x \textit{ mod } 2 = 0$ by computing the differential of $E := \{x \in S \mid x \textit{ mod } 2 = 0\}$ w.r.t. the following code block B

```
S := { };                                  $ here
read(i);
loop while i ≠ om do                       $ om is the undefined element
        S := S ∪ { i };                    $ there
        read (i);
end;
print({ x ∈ S | x mod 2 = 0 });
```

It can be shown that E is differentiable w.r.t. B. The set S is modified at the points marked here and there. The prederivate at here is the assignment $E := \{\}$, and the postderivative is empty. For the point marked there, the postderivative is also empty, and the prederivative is given above. Thus computing $\partial E < B >$ yields

```
E := { };                                  $ prederivative for here
S := { };                                  $ here
                                           $  empty postderivative
read(i);
loop while i ≠ om do
        if i mod 2 = 0 then
            E := E ∪ { i };
        fi;                                $ prederivative for there
        S := S ∪ { i };                    $ there
                                           $ empty postderivative
        read(i);
end;
print(E);
```

Differentiability of an expression can be verified automatically for a large class of expressions, and the derivative $[B_1, B_2]$ can be computed given an appropriate knowledge base of primitive derivatives. Paige's transformation system RAPTS computes additionally informations on the profitability of the transformation, and usually the complexity may be reduced substantially.

Consequently, these transformations provide a way of making a program more efficient without sacrificing correctness. It is not to be expected, however, that the manipulations are performed automatically. The user rather has a collection of a few powerful transformations at her disposal, the system assists in finding invariants, and performs the transformations selected by the user.

4 SETL/E as a Language for Prototyping and Transformational Programming

As described in section 2, all approaches on prototyping have in common to seek the consensus between user and developer and therefore are a method for the user to influence the design of the final software package. To satisfy these requirements, a prototyping language must provide the possibility to formulate on a high expressive niveau and thus permits the programmer to concentrate on the essential of the solution to the problem, the functionality, without becoming lost in detail like storage layout, input/output of complex user-defined data structures or preventing overflow of length restricted static structures.

SETL/E is an imperative language based on finite set theory and first order logic. Thus **SETL/E** programs are easy to read and they allow a concise formulation of the problem and make it possible for the software engineer to work directly on the executable formal specification. **SETL/E** is weakly typed, so the programmer is freed from declaring the data objects used, and it supplies a very powerful mechanism to introduce short hand notations for code fragments in form of parameterized and with local variables equipped macros. Programming in the large in **SETL/E** is supported by a module concept, in which data structures are subdivided in specification and implementation module and which makes possible programming techniques like information hiding, data abstraction and data encapsulation.

The type concept of **SETL/E** offers simple types like atoms, Booleans, integers, reals and strings, and structured types with arbitrary length and heterogeneous component types, additional named and anonymous procedures as first class objects, for developers used to program in functional languages. The structured types are linear ordered tuples, unordered sets and mathematical maps which make it possible to design all kinds of complex data structures up to data bases. The facet of a prototyping language for data engineers is also supported in **SETL/E** by a persistent store, which allows storing data as well as first class operations manipulating these data and therefore to save these data types for later use in other programs.

To manipulate the structured types on a high abstract niveau, powerful operations like tuple and set former expressions, existential and universal quantifiers, iterators for structured objects and routines for unformated input/output are available. On the other side the user who wants to program on a niveau of languages like **Ada** to formulate in the fashion of a vertical prototype in each and every detail, is provided with for- and while loops, an **Ada**-like exception handling mechanism, and procedures for formated input/output.

Thus **SETL/E** is a wide spectrum language which permits transforming expressions on a high level into expressions on a lower one without leaving the language. This makes optimizations like reductions of the strength for arithmetic expressions and differentiation calculus for set expressions possible as discussed above.

The following **SETL/E** program for topologically sorting a directed graph provides an example.

```
program TopSort;
visible nodes, edges;
get("%s", edges);
nodes := domain(edges);
if is_dag() then
        SortTup := [ ];
        while nodes <> { } do
           x := select y in nodes | (notexists z in nodes | [z, y] in edges);
           x into SortTup;
           nodes less x; edges lessf x;
        end while;
end if;
put("%x\n", SortTup[#SortTup .. 1]);
 -- define the procedure is_dag
procedure is_dag;
 -- returns true iff the graph does not contain a cycle
S := nodes;
```

```
shrink_S: loop
        z := select y in S | edges{y} * S = { };
        if z = om then quit shrink_S; end if;
        S less y;
end shrink_S;
return (S <> { });
end is_dag;
end TopSort;
```

The graph is read in by reading the set *edges* which contains pairs, i.e. tuples of length 2. An edge between the nodes x and y is indicated by listing the pair $[x, y]$ in the set *edges*. The variables *nodes* and *edges* are declared as `visible`, hence as accessible in all scopes surordinate to the one containing the declaration (variables and constants are by default local to the scope in which they occur). The set *edges* may be interpreted as a (possibly set valued) map, assigning each node x the set $edges\{x\}$ of its neighbors. The set *nodes* is the domain of *edges*, i.e. the set of all first components of tuples in *edges* (we could alternatively have defined *nodes* as $\{e(1) : e \text{ in } edges\}$). The procedure *is_dag* tests whether or not the graph contains any cycles by repeatedly selecting and removing nodes from the candidate set S. If there is no longer any z to be removed from S, the variable z gets the value `om`, indicating that it is no longer defined. The procedure *is_dag* returns `true` iff the graph does not contain a cycle. The main program repeatedly selects a node x without a predecessor, puts x into the tuple *SortTup* of nodes already sorted, removes x from the set of all nodes as well as all edges emanating from x. This is done until there are no longer any nodes to be processed. The program terminates after writing the nodes in reverse order in which they have been found.

5 Actual and future work

The **SETL/E** system is at present under development. The definition of the language kernel is given in [6]. A system view on persistence was presented in [3] and considerations on concurrency are under way.

The way to execute a **SETL/E** program is to translate it into an ANSI-C program and then compile, load and execute the latter program.

The compiler construction system Eli [8] is the central tool for implementing **SETL/E**. It derives an executable compiler from specifications and produces the ANSI-C source of the derived compiler. Thus the implementation will be highly portable and target machine independent; our current target machines are HP 9000/800 and SUN 4 workstations.

The system kernel is complemented by some tools which support program development under the prototyping paradigma presented here. It goes without saying that minimal tools like tracer, debugger and browser are available, and that a graphical user interface based on X-Windows will be created. An attempt will be made for incorporating a tool for program development based on Knuth's literate programming.

It has become apparent that serious program development is difficult without substantial data base support. Incorporating persistence into the language proper opens this avenue. It allows describing semantic data models in the language itself which in turn will be utilized to supporting the reuse of modules and a data base for configuration control. By the same token, **SETL/E** may be used as a module interconnection language.

Work is under way to incorporate program transformations along the lines of section 3, and type finding as well as value flow analysis serve as a basis for translating the high-level set theoretic constructs into efficient data structures in the target program. For static, persistent data we have developed an algebraic approach that allows us to generate persistent data accessible through the target language, see [4].

Figure 1 gives an overview of the system structure, figure 2 sketches the flow of concepts in a graph of conceptual components.

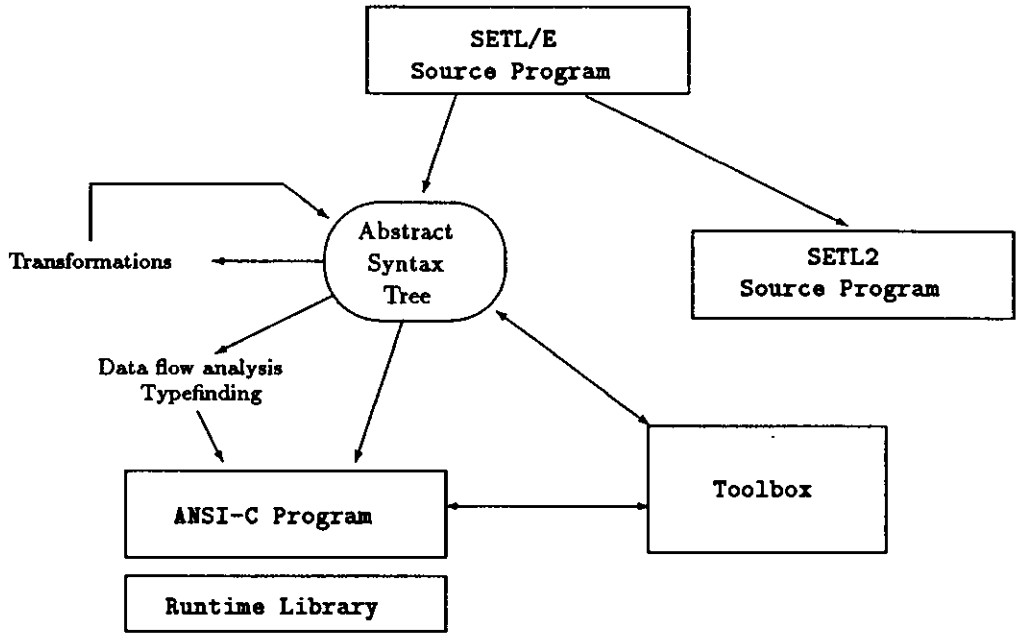

Figure 1: The system structure. Transformation, data flow analysis and typefinding are optional.

6 Conclusion

We work under the hypothesis that finite set theory is an adequate formalism for prototyping substantial software systems. This assumption is put to work in our system by the set oriented language **SETL/E**, a variant of and successor to **SETL**

- a programming environment supporting the work with sets with tools for
  - browsing, tracing and debugging
  - transforming programs based on a calculus of finite differences
  - translating prototypes to production programs
  - translating data based on an algebraic specification of data types
- persistence in **SETL/E**, thus making semantic descriptions of data possible in the language itself, supporting seamless transitions between prototyping of programs and data.

The system under construction may be considered as a CASE tool since it allows generating production code from stable operational requirements. We intend to study the problem of interfacing it with other CASE tools (e.g. the HP Workbench) in order to integrate our system into a production environment.

References

[1] R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllinghoven, editors. *Approaches to Prototyping.* Springer Verlag, Berlin, 1984.

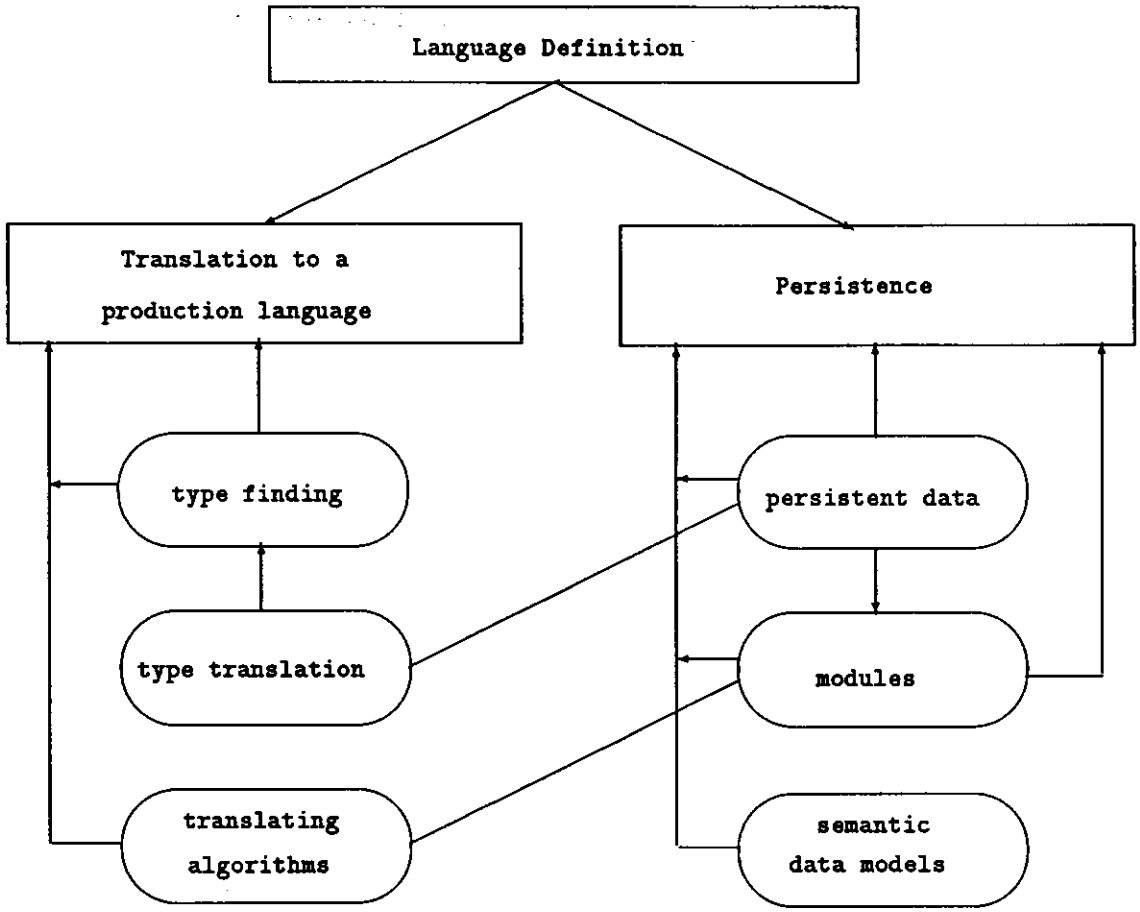

Figure 2: Dependencies among conceptual components

[2] P.A. Dearnley and P.J. Mayhew. In favour of system prototypes and their integration into the system development cycle. *The Computer Journal*, 26(1):36–42, 1983.

[3] E.-E. Doberkat. A proposal for integrating persistence into the prototyping language SETL/E. Informatik-Bericht 02-90, University of Essen, April 1990.

[4] E.-E. Doberkat. Transforming persistent data using an algebraic specification of setl/e's type system. Informatik-bericht, University of Essen, 1990 (in preparation).

[5] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL.* Leitfäden und Monographien der Informatik. Teubner-Verlag, Stuttgart, 1989.

[6] E.-E. Doberkat, U. Gutenbeil, and W. Hasselbring. Setl/e sprachbeschreibung. Informatik-Bericht 01-90, University of Essen, March 1990.

[7] Ch. Floyd. A systematic look at prototyping. In Budde et al. [1], pages 1–18.

[8] R.W. Gray, V.P. Heuring, S.P. Krane, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. Software Engineering Group Report 89-1-1, University of Colorado, Boulder, June 1990.

[9] The CIP Language Group. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L.* Lecture Notes in Computer Science, 183. Springer – Verlag, Berlin, 1985.

[10] S. Hekmatpour and D.C. Ince. *Software Prototyping, Formal Methods and VDM.* Addison-Wesley, Reading, Mass., 1988.

[11] R. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201 – 260, 1987.

[12] S. Khoshafian and T. Briggs. Schema design and mapping strategies for persistent object models. *Information and Software Technology*, 30:606 – 616, 1988.

[13] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Prog. Lang. Syst.*, 4(3):402 – 454, 1982.

[14] M. Schunk. Austausch persistenter Datenstrukturen zwischen Ada und SETL. Master's thesis, Dept. of Computer Science, University of Hildesheim, March 1990.

[15] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL.* Graduate Texts in Computer Science. Springer-Verlag, 1986.