TECHNICAL REPORT 94.07, DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF ZURICH, MAY 1994

Software Development Based on Executable Specifications and Transformations

Markus P. J. Fromherz, Xerox PARC, Palo Alto, USA
Norbert E. Fuchs, Department of Computer Science, University of Zurich, Switzerland

Starting from requirements for a software development method, we derive two prototypes of such a method. Essential characteristics of these prototypes are executable specifications, semantically equivalent external and internal representations of specifications, the use of a repository of predefined components, and schema-based transformations. Crucial for our project is our decision for logic languages. This choice allows us to represent declarative specifications, efficient programs, and the development system itself in the same language. Furthermore – combined with the characteristics of our development method – this choice results in a smooth transition from one activity of software development to the other, from informal to formal representations, from requirements to programs.

1 Introduction

Many authors have pointed out deficiencies of the standard software life-cycle, and suggested as a remedy either enhancements of the life-cycle, or alternatives to it [Agresti 86; Scacchi 94]. Here, we take a different approach by deriving a software development method directly from requirements for it. These requirements emphasise the need to use and to gain experience during software development. We start by enumerating required attributes and features, then derive prototypes of our software development method satisfying more and more – but not yet all – of these requirements. At the same time we revise our initial requirements. This approach is constructive, and inherently reflective, since the derivation of the method resembles the method itself.

2 A First Set of Requirements for Software Development Methods

We define a software development method as the process of interactively establishing requirements for a software product and deriving a software product satisfying these requirements. In the following, we will refine this rather broad definition by stating requirements for the software development method itself.

Before we enumerate these requirements, their origins need to be discussed. Software development, like traditional engineering disciplines, relies more on practical experience and pragmatic decisions than on a comprehensive theory. Thus a software development method is based on experience, and its usefulness and adequacy have to be demonstrated in practice.

The following requirements reflect our experience in industrial and academic software projects, and our preferences. Most of the requirements are probably undisputed, but others, e.g. the requirement for formality, are not shared by all software developers, and still others, e.g. the requirement for executable components, are often contested.

- *The method should eventually support all development activities.*

  The method should particularly support critical development activities, e.g. requirement analysis and specification, and provide the possibility to automate routine activities, e.g. consistency checks and transformations.

- *The method should not prescribe a strict order of activities, and should be adaptable to different development processes and working styles.*

  Some problems of the standard life-cycle, and of other software development methods, arise because a specific order of activities is assumed, or demanded. This has proved to be unrealistic. Furthermore, people prefer to stick to their own established concepts and proven methods, both of which can be very diverse. Thus the method should not prescribe a rigid software development process with a strict order of predefined activities, instead it should allow users to define their own process and working style.

- *The method should support individuals as well as groups of people.*

  The method should assist individual developers as well as groups of users and developers sharing the development work. The method should also support the division of work between people and computers, e.g. by providing clerical assistance for computer-supported cooperative work (CSCW).

- *The method should neither give the impression of being 'intelligent', nor of taking the initiative from the people using it.*

  Software engineering cannot be completely automated. The experience and the abilities of domain specialists and software developers – necessary for the success of a development project – can often not be expressed by fixed rules. Thus the initiative and the decisions should essentially remain with the people using the method, while the method should mainly provide clerical support (e.g. consistency checks), and automate only standard processes (e.g. those for which heuristics were found). Furthermore, to provide a good user interface, the method should lead its users to a clear conceptual model of its working.

- *The software development method should be supported by formal methods.*

  The software development method should be supported by formal methods and tools as far as possible. Only formal methods provide the mathematical foundations necessary to reason with confidence about software, to verify and to validate it. Of course, this requirement does not include inherently informal activities like knowledge acquisition and project management.

- *The formality of the method should be hidden as far as possible.*

  Many users and software developers feel uncomfortable with formality. Thus we want to hide the formality of the method from them as far as possible. One way is to paraphrase formal descriptions in familiar concepts relying on natural language and pictures.

- *The method should produce executable components early on.*

  Executable components that are available early in the development process provide not only conceptual, but also behavioural models of the software system to be implemented. The behaviour of the system interacting with its environment can be demonstrated and observed before it exists in its final form. Thus executability complements inspection, reasoning and formal proof as a means to validate the system with respect to the requirements. Executable components can also help to complete and to clarify initially vague requirements.

- *The method should support both explorative and evolutionary prototyping.*

  Since requirements for software are often not completely or consistently defined, the method should support experimentation to clarify the requirements. Users may also want to experiment with different implementations. Evolutionary prototyping means that there is no essential difference between the first development of the software system and its maintenance. Following Floyd, we consider software development as a continuous and cyclic activity [Floyd et al. 89].

- *The method should build software from predefined components.*

  The reuse of software, especially of powerful and adaptable predefined building blocks, results not only in better and more reliable software, but also in a more efficient development process.

- *The method should use only one relevant document.*

  A set of documents describing a piece of software tends to be inconsistent. Often the code is the only up-to-date documentation of a program, while other documents are obsolete. There should be a single relevant document from which all other documents can be derived if needed This document describes the current version of the software, and possibly summarises its development history.

3 Basic Decisions

The prototypes of software development methods that we present in the sequel rest on a small number of basic decisions reflecting our preferences. These decisions broadly address and partially answer our requirements for a software development method. They can be considered as constraints of the methods we investigate, but by themselves they do not yet constitute a particular method.

3.1 Logic Programming

We decided to use logic languages for the representation of specifications and programs, and also for the implementation of our tools.

Traditionally, the language of logic has been used as a powerful, concise and declarative language for software specifications. The language of logic is based on sound mathematical foundations, has well-defined semantics, and permits descriptions at a very high level of abstraction.

The restriction to Horn clause logic and the mechanisation of proofs that lead to languages like Prolog make specifications in logic executable [Kowalski 85], though some of the qualities mentioned in the preceding paragraph may get compromised. A recent paper points out the advantages of Prolog as a specification and prototyping language [Slonneger 93].

Furthermore, as Kowalski states, in logic programming there is no syntactic difference between a program and its specification. Both specifications and programs can be executed by an interpreter based on automatic deduction – though programs are expected to be more efficient than specifications. Programs can be derived from their specifications by logical deduction, and can thus be considered as computationally useful consequences of their specifications.

Powerful implementations of logic languages, particularly of Prolog, allow us to implement efficient software development tools in logic languages. Thus logic programs can be both tools and objects of these tools. This has most valuable consequences – reflection and self-reference become possible.

3.2 Executable Specifications

An important basis of our method are executable specifications, i.e. formal specifications in high-level, problem-oriented computer languages, specifically in logic languages.

Why should we make specifications executable? There are basically two reasons: validation and prototyping.

As Hoare points out there can be no formal method to validate formal specifications with respect to informal user requirements [Hoare 87]. Thus we have to rely on other means, usually on inspection and reasoning, to convince ourselves of the adequacy of specifications. As a further and complementary means it has been suggested that specifications should be executable [Agresti 86]. Executability provides immediate reflection of the consequences of the specifications, hence realistic experience. Instead of reading the specifications, developers can pose their questions as queries or procedure calls. Executing a specification can be especially helpful to validate non-functional requirements, e.g. for user interfaces.

Requirements of real-life projects are often incomplete and unclear, thus making it impossible to derive a comprehensive and unambiguous specification. Executable specifications can be used as prototypes to gain experience that helps to complete and to clarify requirements.

Very often specifications can be constructed as compositions – e.g. formulas of predicate logic – of simpler properties or operations that are available in the specification language [Hayes & Jones 89; Fuchs 92a; Fuchs 92b]. These components are opaque; the implementation of the property or operation they provide remains hidden. Thus behind their seemingly simple interface, components can be arbitrarily complex, they can even be executable. If they are executable one gets the executability of the complete specification for free, though the specification does not necessarily terminate.

Executable specifications can be considered as prototypes, though they have two attributes which distinguish them from what one usually calls a prototype. First, while prototypes are often produced by ad hoc means, executable specifications must have all the attributes of good specifications, e.g. clarity and conciseness. Second, prototypes are often used to demonstrate the feasibility of only a subset of the required functionality, while executable specifications eventually have to provide all of it.

3.3 External and Internal Representations of Executable Specifications

We decided to provide semantically equivalent external and internal representations of specifications.

There is one common internal representation of the specification that forms the core of our method, and has a number of essential characteristics.

- *The internal representation is based on a logic language, i.e. it is formal.*
- *The internal representation expresses the current state of the specification. It forms the single relevant document of the development method. All other documents are derived from it.*

- *The internal representation is executable and serves as the basis for validation, and as an explorative and evolutionary prototype.*
- *All development tools, including the program derivation tools, operate – more or less independently – on the internal representation.*
- *Altogether, the internal representation opens the way to continuous, cyclical software development that does not distinguish between initial implementation and later maintenance.*

Several graphical and textual external representations – also called views – depict the internal representation. More important, the external representations serve as means for the input of specifications. Graphical and textual editors let us develop specifications from predefined components of a repository. Specifications are automatically mapped to the common internal representation. This mapping is reversible.

Some people regard representations like transition networks, data flow diagrams, or structured English as informal, others call them semiformal, because these representations may have a formal syntax, but often lack a formal semantics. There have been attempts to assign to these representations a formal semantics by mapping them to a formal language manually or automatically [Wing 90]. We follow the correspondence theory [Kowalski 93] by mapping elements of external representations automatically to the internal formal representation. This mapping is reversible. Thus, though our external representations may – and should – give the impression of informality, they are in fact formal and have the same semantics as the internal representation. Depending on the viewpoint, the visible external representations can be considered side-effects or icons.

By providing semantically equivalent external and internal representations, we can solve the conflict between readability and preciseness of specifications, between the different conceptual worlds of the domain specialist and the software developer. We also offer an effective and viable solution to the formal-informal controversy [Meyer 85; Deville 90; Wing 90; Fraser et al. 91; Flener & Popelinsky 94] which combines the advantages of both informal and formal specifications.

The executability of the internal representation and the semantics-preserving mapping between the internal and external representations enables us to simulate the execution of the specification on the level of the external representations. Thus validation and prototyping in concepts close to the problem domain become possible.

Altogether, we get a smooth transition from one activity of software development to the other, from informal to formal representations, from requirements to programs.

3.4 Repository

Another important constituent of our development method is a repository of predefined components. These components or elements can be building blocks, e.g. for specifications or programs, but they can also be data for tools like transformations. One typical application of the repository is the composition of specifications from predefined components with the help of graphical and textual editors. Another application are program transformations that rely on transformation schemata of the repository. In general, we can say that the repository contains knowledge about the problem domain and about program development.

Using a repository has a number of definite advantages.

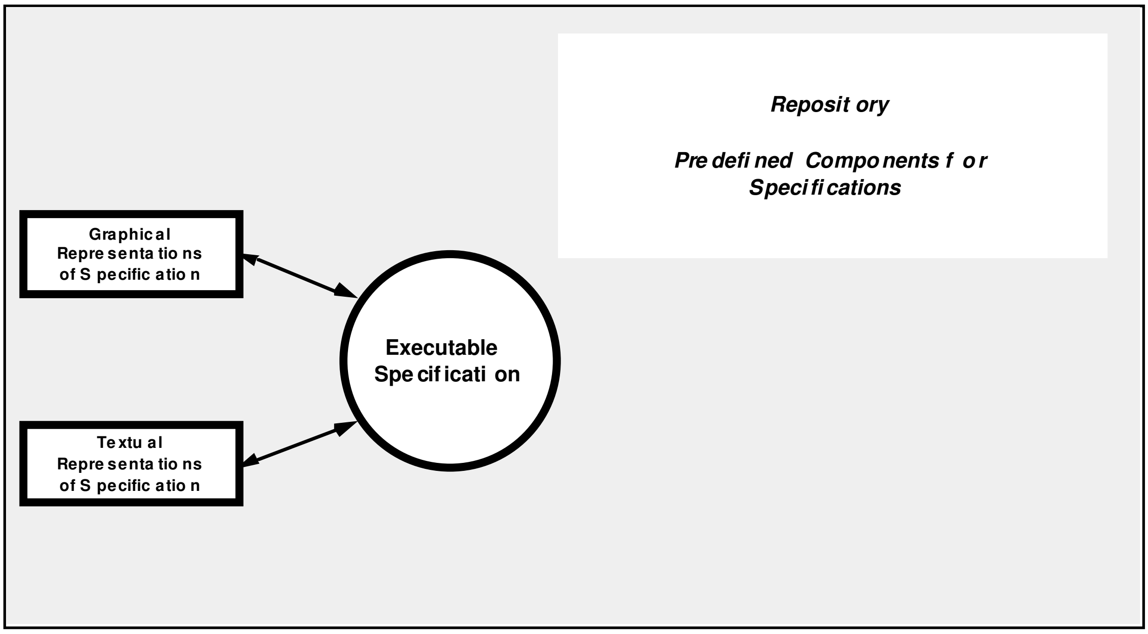
- *The repository supports reuse of components, especially if these components are adaptable. Reuse of standardised, reliable components results in better software and in a more efficient development process.*
- *Specifically, reuse of components for tools facilitates the automatic application of these tools.*
- *The components of the repository essentially support abstraction since they can be used as black boxes. This has two important aspects. Abstract external representations of specifications are mapped to the internal representation and only now have to spell out details that were implicit before. On the other side, components can hide implementation details, thus facilitating portability.*
- *Components of the repository can provide default behaviour which allows us, for example, the execution of skeletal or partially incomplete specifications.*
- *Finally, sets of interconnected components of the repository can built a framework for a problem domain, they can be considered as a theory for this domain.*

The common internal representation of specifications and the repository have been presented as two separate items. This need not be the case. In fact, we will introduce specification frameworks that play both roles.

4 A First Prototype: The Explore Method

4.1 Overview

A first prototypical attempt to design a software development method fulfilling our requirements is restricted to executable specifications. The method, *Explore*, provides three main elements: an executable specification language with problem-oriented views, a repository of executable components, and an appropriate development method [Fromherz 93].



The – largely internal – specification language is based on logic programming, which gives us a general-purpose executable language. In many problem domains, however, large parts of a specification can be expressed better in some problem-oriented notation. Multiple external representations – views – give us this flexibility. Views not only paraphrase the internal formal representation, but may also serve as means for the input of specifications.

The *repository*, an object-oriented framework, supports the integration of internal and external representations and also hides many implementation details. Graphical and textual editors allow us to develop specifications from predefined components of the framework. The executability of the internal representation and defaults given by the framework permit the execution of the specification on the level of the external representations. The method thus gives the possibility to validate specifications and to prototype in concepts close to the problem domain.

Finally, we have created a *development method* for executable specifications that makes use of these properties of specification language, views, and predefined executable components. Given an object-based conceptualisation of the problem domain, this development method is mainly a two-step cycle which consists of the integration of requirements into the framework and of their subsequent specialisation. This cycle is inherently intertwined with the verification and validation of the specification.

4.2 The Specification Language *Explore/L*

Every specification has to be built on an analysis of the requirements and a particular conceptual model of the problem domain [Fromherz 93]. The discussion of these tasks is beyond the scope of this paper, but we would like to point out that the model for describing the requirements does not preclude any specification language. As the identification of objects and their relationships has proved to be natural and useful in domain analysis [Gibson 90; Booch 91], we assume an object-based conceptualisation.

We investigated various logic languages as internal representations for executable specification. In our view, the ability to structure specifications in the large and to support reuse is a most important requirement, so we looked at various frame-based and object-oriented logic languages. We also wanted to be able to execute incomplete specifications, to separate normal from exceptional behaviour, and to represent assertions and test-related procedures in the framework.

In order to satisfy our requirements, we developed *Explore/L*, our own object-oriented extension of Prolog [Fromherz 91]. It features classes and instances with typed attributes, relations, data encapsulation, multiple inheritance, assertions/demons, optional functional notation, and handling of dynamic and incomplete methods. See [Fromherz 93] for an extensive motivation of these features. Explore/L programs may access Prolog programs and vice versa.

As Explore/L programs can be compiled to Prolog, standard logic programming techniques such as declarative debugging and partial evaluation can be adapted to Explore/L. The Explore/L development system is also the implementation basis of all other tools for Explore. Explore/L is implemented in LPA MacProlog.

4.3 The *Specification Framework*

An object-oriented framework is a set of interconnected objects that provide the basic functionality of applications in a specific domain. With such a framework available, an application is built by reusing the structure of the framework while specialising its objects to the individual requirements. In this respect, a framework is more than just an object library, as it provides not only classes and instances, but also the design of an application. For Explore, we have introduced the notion of a *specification framework*, an object-oriented framework for executable specifications.

The specification framework is an attempt at a general model in which user requirements can be formulated. The framework supports the formalisation of informal ideas and aids in categorising the information gained through domain analysis. Inherently, the framework also supports the idea of reuse.

Through default functionality, the framework can support the separation of problem-oriented from implementation-oriented parts, advocate standards for typical operation, typical interfaces etc., and support the evolution of specifications. In particular, Explore's framework *Explore/F* supports models for behaviour specifications and interface specifications. These models are defined by sets of abstract classes with predefined relations. These relations have to be filled in, or overridden, by concrete objects of the specification. Default relations allow us to handle concrete objects which leave part of their definition open. Thus, specifications can temporarily remain incomplete.

The framework classes also provide default operations which describe implementation-oriented aspects such as window handling. Default operations typically provide so-called hook methods. Through this mechanism, specifications can inherit the overall behaviour of the framework and achieve customised functionality at specified points.

Finally, where specifications follow the structure of the framework, the framework supports the analysis, execution and presentation of specifications. Again, we feel that an object-oriented logic language was instrumental in satisfying this range of requirements.

4.4 Multiple Views

The specification framework defines several subsystems for various aspects. For complex behaviour with elaborated interfaces, finding and specialising suitable classes for the particular requirements is not trivial. In general, for each aspect defined by a specification – behaviour, data, user interfaces etc. – a different specification notation may be desirable, e.g. finite-state machines, object-oriented abstractions, dialog construction. Likewise, different people prefer different notations for the same aspect, e.g. Petri nets instead of transition networks. The ability to adapt to individual preferences becomes even more significant if prospective users of the specified software are to participate in its specification.

In Explore, views are different external representations of the common internal representation, Explore/L. View editors, collectively called *Explore/V*, allow us to graphically and textually compose specifications from predefined classes of the framework. Each editor notation is a language for combining objects in an abstract, problem-oriented manner. View editors can also be used to represent existing parts of the specifications in their notation and to generate reports. Finally, the editors support the navigation in the specification from their point of view.

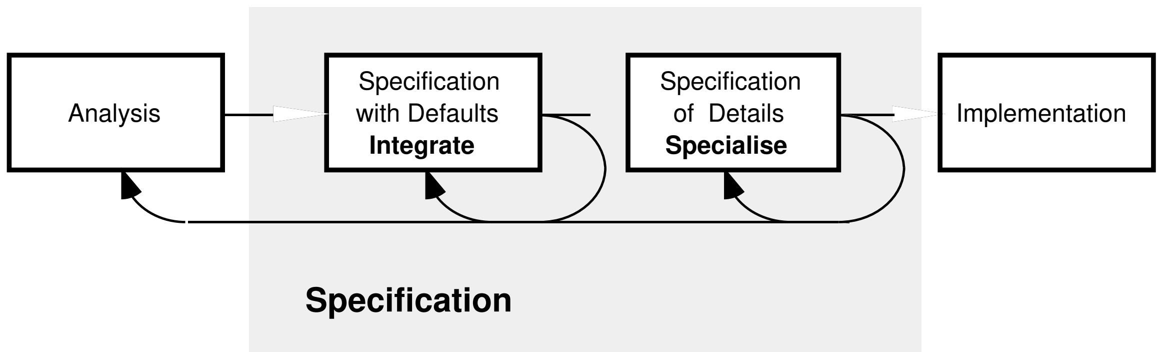
The specification framework not only benefits from the multiple view approach, but also supports it. As the framework provides a common structure for different views, it facilitates translation between views, redundancy control, and complementarity checking.

Explore/V currently consists of the *Process Editor*, the *Dialog Editor*, and the *Hierarchy Editor*, to specify behaviour, user interfaces, and object-oriented abstraction, respectively.

4.5 The Development Method

We have shown that the elements of Explore, its common language, framework and view editors, are a tightly integrated set of techniques and tools. There is a natural way in which these techniques are used to develop specifications.

Overall, specification development consists of two steps. The initial conceptual analysis yields a set of objects with their interrelations. In a first step, the specifier concentrates on the global problem definition and *integrates* objects and relations into Explore/F, which provides default behaviour and default representations for a high level of abstraction. Integration is naturally guided by the framework and further assisted by the view editors. Consequently, early versions of the specification mainly reflect the framework; they are executable, but do heavily rely on defaults.



In a second step, defaults are overridden and further operations are added in order to *specialise* the specification to the required functionality. This is an incremental, evolutionary process where the specification can be validated after each step.

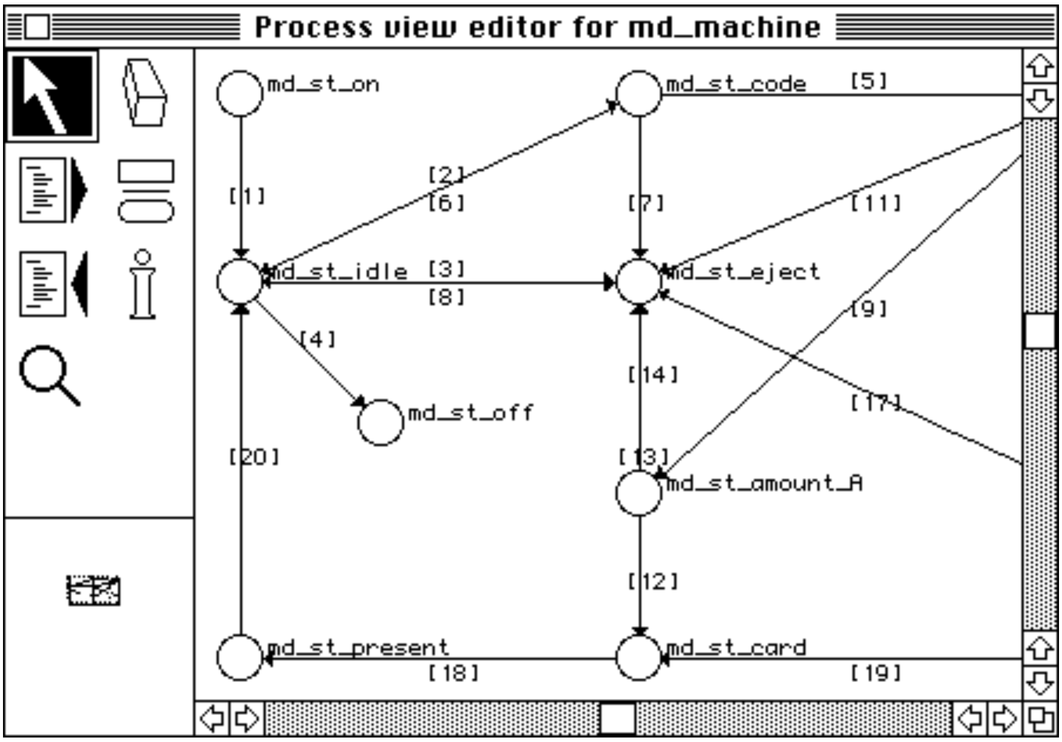
4.6 An Example Specification with *Explore*

We used as specification examples a small library database [Kemmerer et al. 87], and an automated teller machine called *SimpleMat* [Fuchs 89]. To demonstrate working with Explore, we will sketch the specification of the automated teller machine. As its name indicates, SimpleMat provides the minimal functionality of an ATM – withdrawing money with or without receipt. The complete requirements of SimpleMat can be found in the appendix.

Specifying SimpleMat consists of two integration and one specialisation steps. In the first integration step we define SimpleMat as a finite-state machine with the help of the tools of Explore's Process Editor. The next picture shows the Process Editor and a partial view of SimpleMat's state transition network.

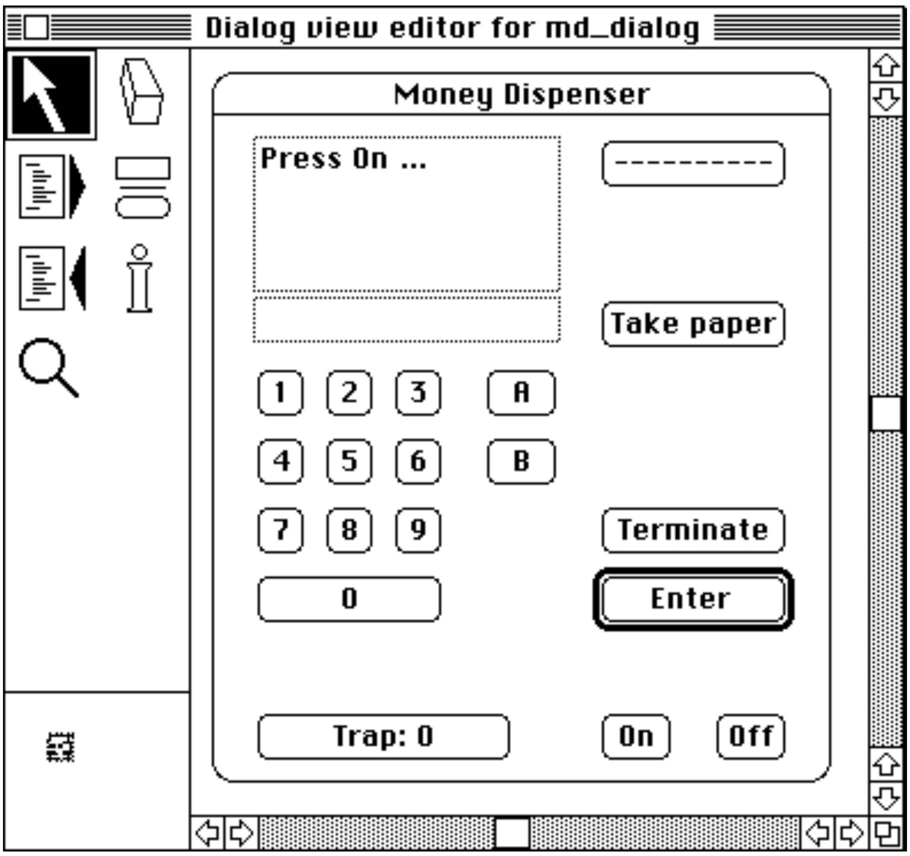
Using the default functionality provided by the specification framework this incomplete specification is already executable. We can put the finite-state machine through its paces and check if it has the required states and transitions.

For the second integration step, we assume that transitions of the finite-state machine are triggered by user interactions. We use Explore's Dialog Editor to define a user interface for the SimpleMat. The names of events triggering the transitions are used to



(Note: the numbers in brackets refer to the names of the transitions)

connect the transitions to elements of the user interface. The following picture shows the Dialog Editor with a part of SimpleMat's user interface.



Again, due to the default functionality of the specification framework this new version of the SimpleMat specification is executable. We can trigger transitions of SimpleMat's finite-state machine by simulating operations of the user.

To incorporate the required functionality of the SimpleMat we access the textual representation of the specification, i.e. the Explore/L code, and override or extend default methods and attributes.

Following is a section of the completed Explore/L specification for SimpleMat. These definitions represent two alternative transitions after the desired amount of money has been entered by the user. All objects (dialog, printer etc.) correspond to objects of the real world that perform certain functions, or have certain information available. The first transition is chosen if the amount is valid for the current card. In this case, the money trap is instructed to prepare the money, a receipt is printed (if button A was pressed), and the dialog interface is told to display an appropriate message. The second transition is chosen if the amount is not valid; again, a suitable message will be displayed.

```
tr_valid_amount(Event, Amount) :-
     current_card <- valid_amount(Amount) &
     money_trap <- set_amount(Amount) &
     (
       machine::current_state = st_amount_A
       ->
       TextIdent = text_prep
     ;
       TextIdent = text_paper &
       printer <- print_receipt
     ) &
     dialog <- tr_next(Event, next_text(TextIdent)).

tr_invalid_amount(Event, Amount) :-
     ~ current_card <- valid_amount(Amount) &
     dialog <- tr_next(Event, next_text(text_iamount)).
```

The SimpleMat being completely specified, we can execute the specification to validate its functionality with respect to given requirements, to complete and clarify vague requirements, or to experiment with the user interface.

4.7 Lessons Learned from the First Prototype

Though it was restricted to executable specifications, our first prototype of a software development method taught us some valuable lessons.

On the positive side we learned that

- *specifications developed with Explore show a realistic behaviour,*
- *validation by execution quickly identifies missing requirements,*
- *execution of incomplete specifications is possible and provides valuable early information,*
- *view editors permit problem-oriented external representations of specifications,*
- *the repository supports adaptability and reuse, and isolates implementation details,*
- *Prolog/Explore proves to be reasonably efficient and highly expressive.*

Conversely, on the negative side we learned that

- *since part of the actual functionality of a specification must be defined in the formal internal representation Explore/L we are violating our requirement that the software development method should hide its formality,*
- *efficient implementation of object-orientation needs non-declarative elements.*

The following points remain undecided

- *scaling-up of the Explore method to larger examples,*
- *cooperation of several people.*

Formal specifications in Explore/L tend to be much longer than the requirements from which the specifications are derived. In another context, [Meyer 85] made a similar observation. At first sight, this might be considered as a negative result, but it is in fact a positive one since it demonstrates that implicit knowledge can be effectively stored in the repository. As a consequence, the external representations of specifications – in Explore at least the graphical ones – can remain abstract, concise and problem-oriented. During the mapping from the external to the internal representation of a specification the necessary details are filled in by the components of the repository. Cf. section 6.1 for further implications.

5 A Second Prototype: Adding Transformational Implementation

The first prototype of our method brought valuable experience, but is only partially successful since it does not adequately satisfy some requirements, and others not at all. We use the experience gained with the first prototype to derive the second one, and – at the same time – to revise our requirements for software development methods.

5.1 Transformational Implementation Paradigm

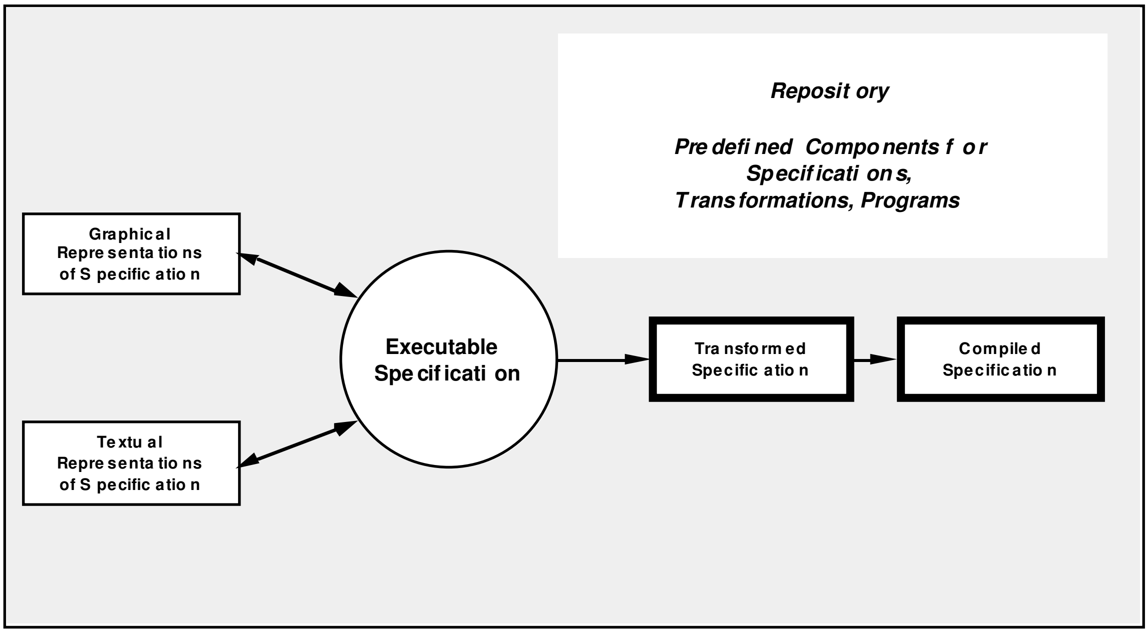
Our positive experience with the efficiency and expressiveness of Prolog/Explore suggests that we could use logic languages not only for the specification of programs but also for their implementation. Candidate languages are suitable extensions of Prolog, like extended Horn clauses [Lloyd & Topor 84] and Explore/L. Declarative subsets of these languages can be used as specification languages, while efficient programs are represented with the help of the full languages.

Since the internal representation of the specifications is a logic language, e.g. Explore/L, it is very tempting to compile the internal representation directly. However, this approach will lead to rather inefficient implementations because compilers perform only local optimisations, but not the global changes necessary to transform a potentially inefficient specification into an efficient program. Thus we decided to follow the transformational implementation paradigm [Agresti 86; Rich & Waters 86]. Specifications are first transformed by powerful source-to-source transformations and then compiled.

Hence we add a further requirement for software development methods.

- *Transformational implementation.*

  Specifications and programs should be expressed in (subsets of) the same logic language. Programs should be derived from the specifications by deductive synthesis, eventually followed by compilation.



Deductive synthesis has a long tradition in logic programming, specifically unfold/fold transformations have played an important role [Gardner & Shepherdson 89]. The field is still very active, as can be seen from the large number of publications and the existence of conferences specifically dedicated to logic program synthesis and transformation [Clement & Lau 92; Lau & Clement 93; Deville 94]. Comprehensive overviews and pointers to the literature can be found in three recent publications [Bundy 92; Flener 93; Flener & Popelinsky 94].

Transformational implementation supports also the 'single document' requirement. Especially, if the transformations can be performed automatically, the internal representation of the specification forms the only relevant document of the software. Instead of programs only simpler specifications need to be maintained and evolved.

5.2 Transformation Schemata

The application of deductive transformation rules, e.g. the unfold/fold rules, requires user intervention and thus makes it difficult to automate transformations. Strategies have been proposed to overcome practical problems of individual unfold/fold rules and to automate transformations [Pereira & Shieber 87; Lakhotia & Sterling 88; Nielson & Nielson 90; Proietti & Pettorossi 90]. However, these strategies do not lead easily to strategies for transformations consisting of several applications of unfold/fold rules.

We suggest a new approach that is based on transformation schemata and promises a degree of automation not possible thus far. This approach leads to a natural extension of the repository – transformation schemata are simply added to it – and to a powerful synergy between transformation schemata and building blocks for specifications.

Many logic programs are structured similarly, and can be understood as instances of program schemata. By far the most comprehensive set of schemata covering all list processing programs was devised by Gegg-Harrison [Gegg-Harrison 89].

Let us assume that a program to be transformed can be described abstractly by a program schema, and the transformed program by another program schema. Thus we may say that we transform an instance of one program schema into an instance of another one. This leads us to the idea of transforming program schemata instead of programs, and to define *transformation schemata* [Fuchs & Fromherz 91].

Transformation schemata are standardised transformations which transform input program schemata into output program schemata. Each transformation schema incorporates a fixed sequence of transformation steps, e.g. applications of unfold/fold rules. Transformation schemata incorporate our transformation decisions, i.e. the order and type of transformation steps, definitions of new predicates, selections of literals to be unfolded or folded, or any other information relevant to the transformation in question. In brief, each transformation schema represents one specific transformation strategy. A set of transformation schemata constitutes an extensible transformation system and the transformation of individual programs is reduced to the search for an appropriate transformation schema in the system.

Transformation of a logic program means that a conjunction of literals

$$Q_1, A_1, \ldots, A_n, Q_2$$

is replaced by another conjunction of literals

$$Q_1, A, Q_2$$

Each A_i calls a predicate P_i while A calls the predicate P which is semantically equivalent to the predicates $P_1, \ldots, P_n$. Usually, we expect calculations using P to be more efficient than those using $P_1, \ldots, P_n$.

For conciseness we say that we transform the set of terms $\{P_1/A_1, \ldots, P_n/A_n\}$ into the term P/A. We derive P/A from $\{P_1/A_1, \ldots, P_n/A_n\}$ by the three steps *abstraction*, *selection*, and *specialisation*.

Abstraction

For each program P_i (i=1, ... , n) we identify a program schema S_i which describes P_i abstractly. This abstraction generates a set of substitutions θ_i for schema variables. The same abstraction leads from the literals A_i to the abstract literals G_i. In short, the abstraction step replaces each term P_i/A_i by its abstraction S_i/G_i, and we have $P_i/A_i = S_i\theta_i/G_i\theta_i$.

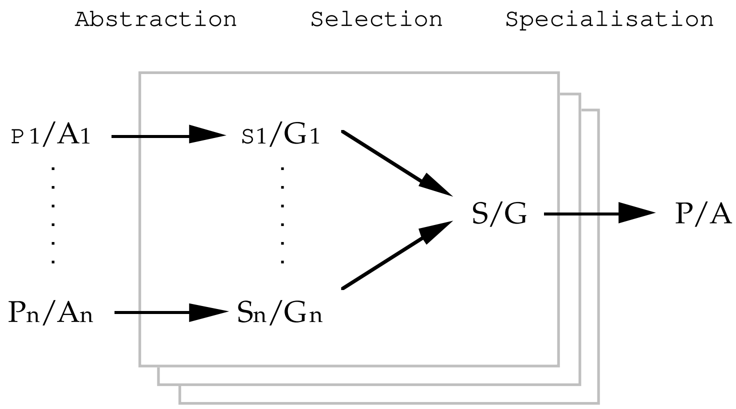
Selection

Transformation schemata transform the set of abstract terms $\{S_1/G_1, \ldots, S_n/G_n\}$ into an abstract term S/G, i.e. a transformation schema is defined as $\{S_1/G_1, \ldots, S_n/G_n, S/G\}$. In general there are several transformation schemata which have $\{S_1/G_1, \ldots, S_n/G_n\}$ as input. We select the transformation schema which generates a desired output program schema S together with an abstract literal G.

Specialisation

We apply the substitution $\theta = \theta_1 \ldots \theta_n$ to S/G to get the transformed program $P=S\theta$ and the transformed literal $A=G\theta$.

Once the transformation schemata are available, these three steps can be automated to a large degree. Since the abstraction and the specialisation steps rely on unification, user intervention is necessary only in the selection step, i.e. the transformation of a



program is reduced to the selection of an appropriate transformation schema among a set of schemata suggested by the system. This approach results in a considerable reduction and simplification of user interaction since users do not have to deal with the intricacies of individual transformation rules, but can concentrate on the form of the transformed programs alone. For standard transformations, e.g. the transformations of the elements of the repository, even this user intervention can be replaced by heuristics.

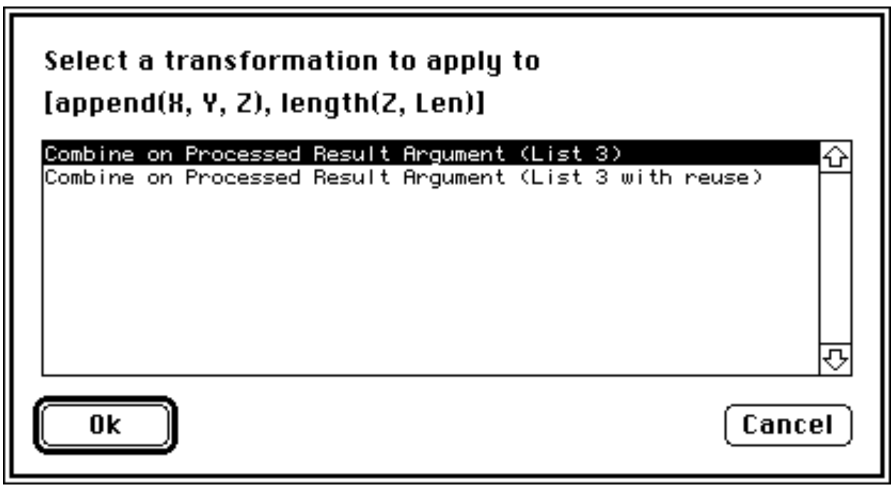
An extensible transformation system based on transformation schemata has been implemented. Some of the transformation schemata are derived from unfold/fold transformations, while others incorporate transformation techniques that are widely used in the logic programming community, e.g. the introduction of accumulators and of difference lists. These transformation schemata are inserted into the repository.

5.3 A Simple Transformation Example

The method will be demonstrated by a simple example. Appending two lists X, Y, and calculating the length Len of the resulting list Z by the conjunction

```
append(X, Y, Z), length(Z, Len)
```

is inefficient since the elements of the list X are processed twice while once would be sufficient. To transform the conjunction and the predicates they call, we invoke the schema transformation system which performs the abstraction step automatically and presents to us the following window for the selection step.



We select the first transformation schema named 'Combine on Processed Result Argument (List 3)'. The specialisation step is automatically performed and we get as result the old and new conjunctions of goals and the newly synthesised predicate append_length/4 which efficiently calculates both the concatenation of two lists and the length of the resulting list.

```
Transformation 'Combine on Processed Result Argument (List 3)' :

Old goal:   append(X, Y, Z), length(Z, Len)
New goal:   append_length(X, Y, Z, Len)


New predicate:

append_length([], [], [], 0).
append_length([], [Y| Ys], [Y| Ys], Len) :-
     append_length([], Ys, Ys, LenYs),
     Len is LenYs+1.
append_length([X| Xs], Y, [X| Zs], Len) :-
     append_length(Xs, Y, Zs, LenZs),
     Len is LenZs+1.
```

With a further schema transformation we could introduce an accumulator argument into append_length/4 making it tail-recursive, and thus even more efficient.

5.4 Lessons Learned from the Second Prototype

On the positive side we learned that

- *transformation schemata are powerful and flexible,*
- *transformation schemata can be used easily and efficiently,*
- *new transformation schemata can be derived with moderate effort,*
- *schemata can be combined to give more elaborate transformations.*

Since the capabilities of the implemented transformation system are still limited

- *transformation of a complete specification*

remains undecided for the time being.

Our software development method comprises now two phases, one problem-oriented, the other one implementation-oriented. This division corresponds more to the reality of software development than the phases of the standard life-cycle that originated from the idea of mile-stones [Zave & Yeh 81]. Of course, the activities of the phases of the standard life-cycle remain, but they are spread over the two phases of our method, and can no longer be uniquely located [Swartout & Balzer 82].

6 Current and Future Research

The second prototype of our software development method comprises executable specifications and transformational implementation, and thus covers a substantial part of software development. Nevertheless, many of our requirements remain insufficiently fulfilled, and are addressed by current and future research.

6.1 A Further External Representation: Restricted Natural Language

The most glaring shortcoming of the current prototype of our method is the merely partial realisation of the 'hidden formality' requirement – to define the functionality of the specification we have to rely on the formal internal representation since there is no suitable external one.

The use of natural language as data base query language [Androutsopoulos 94], the research in legal reasoning [Kowalski 92], and work on natural language for specifications [Macias & Pulman 92; Pulman 94] encourages us to investigate restricted natural language as additional external representation of specifications, i.e. a subset of natural language that can be translated into Horn clauses. We are currently investigating a structured form of the natural language requirements of the SimpleMat.

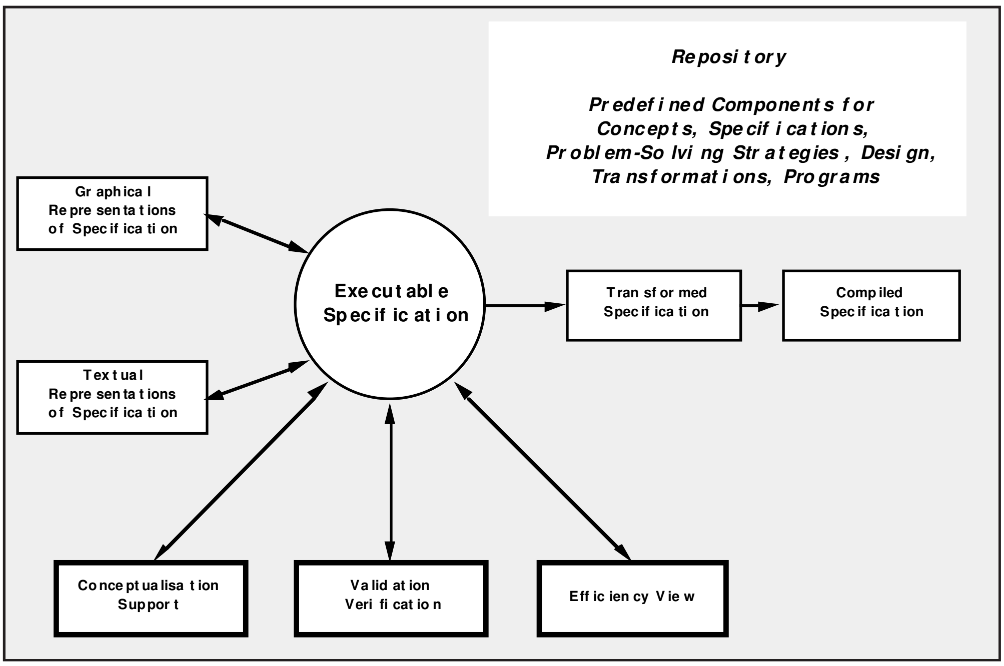
6.2 Conceptualisation Support

Until now we have been proceeding with the assumption that there exists a conceptualisation of the problem domain, that the components of the repository are already given. In general, this will not be the case.

Currently, we are developing an extension of our development method that provides clerical support for requirements engineering and for the conceptualisation of problem domains as a collection of objects, object relations, and constraints. These concepts will be used to define building blocks for the repository.

We start with the assumption that user requirements – as far as they are already known – are given in the form of texts in natural language with a relatively small vocabulary and a restricted syntax, or that the requirements can be transformed into this form. These assumptions are fulfilled for most technical problems, specifically for our two standard examples – the automated teller machine and the library data base. We are separating the domain knowledge into a problem-independent executable model – a theory of the domain – and a problem description, both expressed in the concepts of the domain which are defined in a domain-dependent glossary.

The choice of the concepts has indirect consequences for the validation and the transformation of specifications, since these operations make use of the building blocks of the repository.



6.3 Transformation Schemata

More work needs to be done for transformation schemata. Specifically, we will investigate the systematic development of transformation schemata, proofs of useful transformation schemata that cannot (easily) be derived by known transformation techniques, transformation schemata for the building blocks of the repository, and heuristics to eliminate user intervention in the selection step thus rendering transformations automatic.

Transformation schemata originally devised for unfold/fold transformations are powerful and general enough to introduce into specifications problem-solving strategies, like divide-and-conquer, data type refinement, simplification, search, recursion, finite differences, abstraction, specialisation, partial evaluation, and memoing.

In the same way we can introduce problem-specific algorithms. This is possible because transformation schemata can be combined with selective preconditions.

Furthermore, these transformations can help to solve another problem. Property-oriented specifications are often only inefficiently executable. To validate them by execution, we may have to transform them into a more efficient form without sacrificing their characteristics as specifications. One might consider these (temporary) transformation as another external representation of the specifications providing an 'efficiency view'.

6.4 Validation

We will develop a practical validation method based on proofs, execution and declarative debugging.

Logical inference is a powerful method for validating specifications with respect to requirements. Unfortunately, inference fails for non-functional requirements like user interfaces. Another validation method is the execution of specifications. Being a kind of test, this method is less powerful than logical inference, but very effective for the validation of non-functional requirements and for the completion and clarification of vague requirements. Both methods are complementary and should be combined for optimal results. For specifications in logical languages we have a special situation - execution is a form of inference, i.e. validation by inference and by execution coincide if the declarative and procedural meaning of the specification coincide.

In the case of validation errors, the specification may have to be modified. To localise the discrepancy in the specification we can use declarative debugging [Lloyd 87; Nadjm-Tehrani 89; Ducassé et al. 90]. To make declarative debugging practical, we will employ the fact that a specification is composed from components of the repository which are considered as bug-free. This suggests a bottom-up debugging strategy. Combining top-down and bottom-up debugging could eliminate the disadvantage of declarative debugging – the extensive and detailed dialog that can leave the user exhausted and disoriented.

6.5 Repository and Internal Representation

Each of the preceding extensions of our method will introduce appropriate components into the repository that will eventually contain predefined components for concepts, specifications, algorithms, problem-solving strategies, and transformations.

The repository resembles a framework of 'components of expertise' [Steels 88]. Knowledge-level descriptions of expertise in the problem domain and the program development domain lead to principled design and implementation, to reusability and, eventually, to the configuration of new applications by domain specialists.

The extension of our method will also have consequences for the common internal representation that initially consisted of (partially incomplete) specifications only. With the addition of (partially formal) conceptual models and (partially transformed) specifications the internal representation becomes very heterogeneous though it remains executable and mappable to external representations. Also, developers can, but need not, work at several levels of abstraction of the specification at the same time.

7 Related Work

Our work fits into several fields, two of which come to mind immediately – schema-guided software development [Flener 93; Flener & Popelinsky 94], and knowledge-based software development [Rich & Waters 86; Lowry & Duran 90; Lowry & McCartney 91]. Both fields have been very active since many years. Since any attempt to do justice to relevant research is doomed to fail, we will only mention a few projects with intentions similar to ours.

The *Requirements Apprentice* represents an attempt to develop a knowledge-based system for the requirements analysis [Reubenstein & Waters 91]. The Requirements Apprentice is a component of the *Programmer's Apprentice*, a system intended to intelligently assist the software engineer [Rich & Waters 90]. The Requirements Apprentice contains domain specific clichés describing and interrelating knowledge of the domain, and uses these clichés to check pieces of information entered by an analyst for consistency and coherence. Checking is done by dependency-directed reasoning and other inference techniques.

Another knowledge-based system for the requirements analysis is *ARIES* [Johnson et al. 93] which is part of the *Knowledge-Based Software Assistant KBSA* [Green et al. 86]. ARIES resembles our method since it also separates the (external) presentation and the (internal) representation thus providing different points of view. The features of ARIES are: a modularised central repository of requirements information, a single highly expressive representation scheme, communication between system and analyst in terms familiar to the analyst, and tool support to manage, analyse, and evolve requirements information.

In our approach we assume that the domain specialist decides which concepts will be used, not the software engineers, or the development environment. In this aspect we differ from the approaches taken by the Requirements Apprentice, or by ARIES.

Of special interest to our project is the powerful *KIDS* system [Smith 90a; Smith 90b] which provides knowledge-based support for the derivation of correct and efficient programs from formal specifications. High-level transformations are available to derive programs: design tactics for algorithms, deductive inference, program simplification, finite-differencing, data type refinement, and partial evaluation. Transformations are seen as high-level design decisions selected interactively by the user, and applied by the KIDS system to user-selected parts of the specification.

OBSERV is a software development method for embedded systems [Tyszberowicz & Yehudai 92]. The method is based on executable specifications and combines object-orientation, finite-state machines and logic programming. Important goals of the method are the smooth transition to the target system, and the reuse of specifications, design and code. The method is basically text-oriented, an improved version includes a graphic front-end CRUISE.

LOPS is a system that resembles KIDS [Bibel 91]. In LOPS one transforms a logic specification into a form that is amenable to known algorithmic methods, e.g. the introduction of recursion.

FOLON [Henrard & Le Charlier 92] is an environment for the declarative construction of logic programs based on Deville's work [Deville 90]. Deville devises specifications of logic programs in a semiformal notation. Relations are specified in natural language, while formal parts of the specification describe the types and the directionality of the arguments, possible restrictions, and side-effects. Parts of the specification are interpreted as pre- and postconditions of the program. By structural induction and generalisation, Deville derives from the specification a logic description of a predicate expressed as equivalencies in untyped first-order predicate logic. In a further step, the logic description is transformed into a Prolog program. The FOLON system provides efficient clerical support for these steps.

8 Acknowledgements

We would like to thank Hubert Hofmann, Ina Kraan, Dave Robertson and Rolf Schwitter for stimulating discussions, and for valuable comments on an earlier draft of this paper.

9 References

[Agresti 86] W. W. Agresti (Ed.), New Paradigms in Software Development, IEEE Computer Society Press, 1986

[Androutsopoulos 94] I. Androutsopoulos, Natural Language Interfaces to Databases: An Introduction to the Current Technology, Technical Report, Department of Artificial Intelligence, University of Edinburgh, 1994

[Bibel 90] W. Bibel, Towards Predicative Programming, in: M. R. Lowry, R. D. McCartney (Eds.), Automating Software Design, AAAI Press and MIT Press, pp. 1990

[Booch 91] G. Booch, Object-Oriented Design with Applications, Benjamin & Cummings, 1991

[Bundy 92] A. Bundy, Tutorial Notes: Reasoning about Logic Programs, in: G. Comyn, N. E. Fuchs, M. Ratcliffe (Eds.), Logic Programming in Action, Proceedings of the Logic Programming Summer School LPSS '92, Lecture Notes in Artificial Intelligence 636, Springer Verlag, pp. 252-277, 1992

[Clement & Lau 92] T. Clement, K.-K. Lau (Eds.), Proceedings of LOPSTR '91, Springer Verlag, 1992

[Deville 90] Y. Deville, Logic Programming, Systematic Program Development, Addison-Wesley Publishing Company, 1990

[Deville 94] Y. Deville (Ed.), Proceedings of LOPSTR '93, Springer Verlag, 1994

[Ducassé et al. 90] M. Ducassé, A.-M. Emde, T. Kusalik, J. Levy, Logic Programming Environments, ICLP '90 Preconference Workshop, Eilat, 1990

[Flener 93] P. Flener, Logic Algorithm Synthesis from Examples and Properties, PhD, Université Catholique de Louvain, 1993

[Flener & Popelinsky 94] P. Flener, L. Popelinsky, On the Use of Inductive Reasoning in Program Synthesis: Prejudice and Prospects, in: Proceedings of LOPSTR '94, Pisa, 1994

[Floyd et al. 89] C. Floyd, F.-M. Reisin, G. Schmidt, STEPS to Software Development with Users, in C. Ghezzi, J. A. McDermid (eds.) Proceedings ESEC '89, Lecture Notes in Computer Science, no. 387, Springer Verlag, 1989

[Fraser et al. 91] M. D. Fraser, K. Kumar, V. Vaishnavi, Informal and Formal Requirements Specification Languages: Bridging the Gap, IEEE Transactions on Software Engineering, 17, 5, pp. 454 - 466, 1991

[Fromherz 89] M. P. J. Fromherz, A Survey of Executable Specification Methodologies, University of Zurich, Computer Science Dept., TR no. 89.05, June 1989

[Fromherz 91] M. P. J. Fromherz, Explore/L — An Object-Oriented Logic Language; Tutorial, Reference Manual, Declarative Semantics, Implementation, University of Zurich, Computer Science Dept., TR no. 91.06, June 1991

[Fromherz 93] M. P. J. Fromherz, A Methodology for Executable Specifications — Combining Logic Programming, Object-Orientation and Multiple Views. University of Zurich, Computer Science Dept., Ph.D. thesis, 1993

[Fuchs 89] N. E. Fuchs, SimpleMat (Customer's View), University of Zurich, Computer Science Department, Working Paper, 1989

[Fuchs 92a] N. E. Fuchs, Specifications Are (Preferably) Executable, Software Engineering Journal, September 1992, pp. 323-334

[Fuchs 92b] N. E. Fuchs, Hoare Logic, Executable Specifications and Logic Programs, Structured Programming, 13, pp. 129-135, 1992

[Fuchs & Fromherz 91] N. E. Fuchs, M. P. J. Fromherz, Schema-Based Transformations of Logic Programs, in: K.-K. Lau, T. Clement (Eds.), Proceedings of LOPSTR '91, University of Manchester, 1991, Springer, 1992

[Gardner & Shepherdson 89] P. A. Gardner, J. C. Shepherdson, Unfold/Fold Transformations of Logic Programs, Report PM-89-01, School of Mathematics, University of Bristol, 1989

[Gegg-Harrison 89] T. S. Gegg-Harrison, Basic Prolog Schemata, CS-1989-20, Department of Computer Science, Duke University, September 1989

[Gibson 90] E. Gibson, Objects - Born and Bred, Byte, 15, 10, pp. 245-254, 1990

[Green et al. 86] C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich, Report on a Knowledge-Based Software Assistant, in: C. Rich, R. C. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann Publisher, 1986

[Hayes & Jones 89] I. J. Hayes, C. B. Jones, Specifications are not (necessarily) executable, Software Engineering Journal, November 1989, pp. 330-338

[Henrard & Le Charlier 92] J. Henrard, B. L. Charlier, FOLON: An Environment for Declarative Construction of Logic Programs, in: M. Bruynooghe, M. Wirsing (Eds.), Proceedings of PLILP '92, Programming Language Implementation and Logic Programming, Springer Verlag, 1992

[Hoare 87] C. A. R. Hoare, An overview of some formal methods for program design, in: C. A. R. Hoare, C. B. Jones, Essays in Computing Science, Prentice Hall, 1987, pp. 371-387

[Johnson et al. 93] W. L. Johnson, K. M. Benner, D. R. Harris, Developing Formal Specifications from Informal Requirements, IEEE Expert, 8, 4, pp. 82 - 90, 1993

[Kemmerer et al. 87] R. A. Kemmerer, S. White, A. Mili, N. Davis, Problem Set for the Fourth International Workshop on Software Specification and Design, in: IEEE, ACM, Proceedings Fourth International Workshop on Software Specification and Design, Monterey, CA, 1987, pp. ix - x

[Kowalski 85] R. A. Kowalski, The relation between logic programming and logic specification, in: C. A. R. Hoare, J. C. Shepherdson (Eds.), Mathematical Logic and Programming Languages, Prentice-Hall International, 1985

[Kowalski 92] R. A. Kowalski, Legislation as Logic Programs, in: G. Comyn, N. E. Fuchs, M. Ratcliffe (Eds.), Logic Programming in Action, Proceedings of the Logic Programming Summer School LPSS '92, Lecture Notes in Artificial Intelligence 636, Springer Verlag, pp. 203-230, 1992

[Kowalski 93] R. A. Kowalski, Logic without Model Theory, Technical Report, Imperial College, 1993

[Lakhotia & Sterling 88] A. Lakhotia, L. Sterling, How to Control Unfolding when Specializing Interpreters, in L. Sterling (ed.), The Practice of Prolog, MIT Press, 1990, pp. 171

[Lau & Clement 93] K.-K. Lau, T. Clement (Eds.), Proceedings of LOPSTR '92, Springer Verlag, 1993

[Lloyd 87] J. W. Lloyd, Declarative Error Diagnosis, New Generation Computing, 5 (1987), pp. 133-154

[Lloyd & Topor 84] J. W. Lloyd, R. W. Topor, Making Prolog More Expressive, Journal of Logic Programming, 1, 3, pp. 225-240, 1984

[Lowry & Duran 90] M. Lowry, R. Duran, Knowledge-Based Software Engineering, in: A. Barr, P. R. Cohen, E. A. Feigenbaum (Ed.),^(Eds.), The Handbook of Artificial Intelligence, Addison-Wesley, Reading, pp. 242-322, 1990

[Lowry & McCartney 91] M. R. Lowry, R. D. McCartney, Automating Software Design, AAAI Press and MIT Press, 1991

[Macias & Pulman 92] B. Macias, S. Pulman, Natural Language Processing for Requirements Specifications, in: F. Redmill, T. Anderson (Eds.), Safety-Critical Systems, Current Issues, Techniques and Standards, Chapman & Hall, pp. 67-89, 1993

[Meyer 85] B. Meyer, On Formalisms in Specifications, IEEE Software, vol. 2, no. 1, pp. 6-26, 1985

[Nadjm-Tehrani 89] S. Nadjm-Tehrani, Contributions to the Declarative Approach to Debugging Prolog Programs, Thesis 187, Department of Computer and Information Science, University of Linköping, 1989

[Nielson & Nielson 90] H. R. Nielson, F. Nielson, Eureka Definitions for Free – Disagreement Points for Fold/Unfold Transformations, Proceedings of ESOP '90, LNCS 432, 1990, pp. 291-305

[Pereira & Shieber 87] F. C. N. Pereira, S. M. Shieber, Prolog and Natural-Language Analysis, CSLI, Lectures Notes 10, 1987

[Proietti & Pettorossi 90] M. Proietti, A. Pettorossi, Synthesis of Eureka Predicates for Developing Logic Programs, Proceedings of ESOP '90, LNCS 432, 1990, pp. 306-325

[Pulman 94] S. Pulman, Natural Language Processing and Requirements Specification, Presentation at the Prolog Forum, Department of Computer Science, University of Zurich, February 1994

[Reubenstein & Waters 91] H. B. Reubenstein, R. C. Waters, The Requirements Apprentice: Automated Assistance for Requirements Acquisition, IEEE Transactions on Software Engineering, 17, 3, pp. 226-240, 1991

[Rich & Waters 86] C. Rich, R. C. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann Publishers, Inc., 1986

[Rich & Waters 90] C. Rich, R. C. Waters, The Programmer's Apprentice, ACM Press & Addison-Wesley Publisher. Corp., 1990

[Scacchi 94] W. Scacchi, Process Models in Software Engineering, in J. J. Marciniak (Ed.), Encyclopedia of Software Engineering, John Wiley & Sons Inc., 1994

[Slonneger 93] K. Slonneger, Executing Continuation Semantics: A Comparison, Software - Practice and Experience, 23, 12, pp. 1379-1397, 1993

[Smith 90a] D. R. Smith, KIDS – A Knowledge-Based Software Development System, in: M. R. Lowry, R. D. McCartney (Eds.), Automating Software Design, AAAI Press and MIT Press, 1990

[Smith 90b] D. R. Smith, KIDS: A Semiautomatic Program Development System, IEEE Transactions on Software Engineering, 16, 9, pp. 1024-1043, 1990

[Steels 88] L. Steels, Components of Expertise, AI Magazine, 11(2), pp. 28-49, 1990

[Swartout & Balzer 82] W. Swartout, R. Balzer, On the Inevitable Intertwining of Specification and Implementation, CACM, vol. 25, no. 7, pp. 438-440, 1982

[Tyszberowicz & Yehudai 92] S. Tyszberowicz, A. Yehudai, OBSERV - A Prototyping Language and Environment, ACM Transactions on Software Engineering and Methodology, 1, 3, pp. 269 - 309, 1992

[Wing 90] J. M. Wing, A Specifier's Introduction to Formal Methods, IEEE Computer, 23, 9, pp. 8-24, 1990

[Zave & Yeh 81] P. Zave, R. T. Yeh, Executable Requirements Specification for Embedded Systems, Proceedings of the 5th International Conference on Software Engineering, pp. 295-304, San Diego, 1981

10 Appendix

SimpleMat [Fuchs 89]

This is a description of a simple money-dispenser called *SimpleMat*. The description specifies the user interface, and in addition makes some assumptions about the mechanisms behind the user interface.

Cards

The customer has a card that carries the following machine-readable data: the card number, the check code, and the maximum amount of money that can be withdrawn.

Personal Code

Customers have personal codes for their cards. The personal code is a fixed length (n) number that is used by a trap door algorithm to calculate a number which should be identical to the check code on the card.

Access

The customer introduces his card, which is read by the SimpleMat. The card number is checked for validity and whether it is currently active. If the card cannot be read, has an invalid card number, or is not currently active it is rejected with the message 'Invalid Card'. Otherwise the access door of the SimpleMat is opened and the message 'Enter Your Personal Code' is displayed. The customer enters the personal code. The code is checked for validity by a trap door algorithm that should result in the check code stored on the card. If the personal code entered is not correct the SimpleMat displays the message 'Incorrect Code. Transaction Terminated. Card Retained' and the door of the SimpleMat is closed. The SimpleMat retains the card.

Transactions

If the correct code has been entered the SimpleMat displays the following message

- Withdraw Money Without Receipt - Key A
- Withdraw Money With Receipt - Key B
- Terminate Transaction - Terminate

Selection B will not be displayed if there is no paper to print the receipt.

By hitting the appropriate key the customer can execute one of the following transactions.

- Key A: Withdraw money without receipt.

  The customer pushes key A and the message 'Enter Amount To Be Withdrawn' is displayed. If the amount entered exceeds the maximum amount the SimpleMat displays 'Enter Smaller Amount' and the customer can reenter the amount s/he wants to withdraw. If the amount entered does not exceed the maximum amount the SimpleMat displays 'Amount is Being Prepared. Retract Card' and expels the card. Finally, the amount of money is prepared and presented to the customer. Then the SimpleMat closes the door.

- Key B: Withdraw money with a receipt.

  As in point 1, only that after expelling the card and before the money is presented a receipt is printed and expelled which contains the amount of money withdrawn, the card number, the SimpleMat number and address, and the time and date. The SimpleMat closes the door.

- Terminate: Terminate transaction.

  If the customer pushes the key 'Terminate' the message 'Transaction Terminated. Retract Card' is displayed, and the card is expelled. The SimpleMat closes the door.