$\text{U}_\text{S}\text{TOP}^\text{IA}$ Requirements

—

Thoughts on a User-friendly System for Transformation Of Programs In Abstracto *

E.A. Boiten M.G.J. van den Brand N.W.P. van Diepen
C.H.A. Koster H.A. Partsch N. Völker

University of Nijmegen
Department of Informatics
Faculty of Mathematics and Informatics
Toernooiveld 1
NL-6525 ED Nijmegen
Phone: +31-80-652236
Email: `eerke@cs.kun.nl`

Abstract

Transformational programming is a program development method which is usually applied using "pen and paper". Since this requires a lot of clerical work (copying expressions, consistent substitution) which is tiresome and prone to error, some form of machine support is desirable. We describe a number of systems that have already been built to this aim. Some of their shortcomings and limitations are identified. Based on our experience with program transformation and transformation systems, we present a long list of features that would be useful in an "utopian" transformation system. This list is presented using an orthogonal division of the problem area. A number of problems with the realisation of some aspects of our "utopian" system is identified, and some areas for further research are indicated.

Keywords

transformational programming, transformation systems, programming environments

*Support has been received from the Netherlands Organisation for Scientific Research N.W.O. under grant NF 63/62-518 (the STOP — Specification and Transformation Of Programs — project) for E.A. Boiten, N.W.P. van Diepen and N. Völker, and under grant 612-317-020 for M.G.J. van den Brand.

1 Introduction

One of the main barriers in the manual application of program transformations is the amount of work necessary to ensure correctness of the application of the various transformation rules. These correctness checks are often of a rather trivial, and very tedious, nature. It is hard to remember that a correctness check should be applied. These considerations led to the obvious idea of machine support for various clerical tasks, including automatic verification of trivial applicability conditions, and signaling of the more difficult ones.

In practice, machine support for program transformation in the form of an editor is a minimal requirement. The programmer copies the current version into the editor, and manually applies the transformations which seem to further his or her aim. Of course, this cannot be called proper support, but it provides a starting point in functionality for any system.

This paper has been motivated and stimulated by ongoing research in the STOP (Specification and Transformation Of Programs) project. Participants are the Computer Science Departments of the Universities of Utrecht and Nijmegen and the Algorithmics and Architecture Department of the Centre for Mathematics and Computer Science in Amsterdam. The aim of the project is to further the research into program specification and transformation. Hence the functionality and components of a transformation system form an important issue within these research activities.

It should be noted that we do not aim at describing a practical program transformation support system. Rather, we first try to list things which could conceivably be part of an ideal system. Based on this, we will discuss the realisation of such a system – what has already been done, what cannot be done, and what are the interesting research topics that arise.

1.1 Why program transformation?

The name *software engineering* has been coined in the sixties to characterise attempts to overcome the problem of the so-called *software crisis*. This crisis was caused by the lack of proper techniques to lift the construction of software from the level of art to the level of engineering. Many attempts have been made to handle this problem. A satisfactory solution, however, has not yet been found. Hence new techniques are still looked for to bridge the gap.

One answer can be found in the following way. The software engineer starts with a formal description of the problem, or of the program to be written. This description is then transformed step by step using formal rules towards a final program (which should probably be efficient, etc.), along the way maintaining the correctness by verifying the correct application of the transformations. Many transformation steps lend themselves to formalisation. Also, combinations of steps allow transformation strategies to be formulated and applied. This approach therefore brings two fundamental engineering aspects into play. It is possible to formulate and use standard techniques, thus gaining confidence in the quality of the final product. And it is possible to "compute the strength of the construction" (prove the properties of the program) with respect to the original specification.

Of course, this leaves open the question of how to find such a formal specification. Ideally, it should be provided by the client who wanted the program in the first place. In practice, the process is more difficult, with a recurrent interaction between the client (who supplies the wishes) and the software engineer (who writes the specification). Though such a form of communication is apt to lead to misunderstandings, it is assumed here that the original specification is a correct formalisation of the informally specified problem. Some first ideas on improving the formalisation process itself can be found in [van Diepen and Partsch, 1990].

1.2 Why a program transformation system?

There are several reasons why one would want machine support for program transformation. Firstly, transformational developments show a considerable amount of clerical work. Usually, only a small part of the program is transformed. In that case, the remainder of the program is just copied. Furthermore, the application of a transformation rule requires instantiation of certain parts of the original program in the resulting program, a task in which an automated system is less likely to make errors in. Some people would already be very happy with a system having just these properties.

Another reason for using a transformation system is formality. A transformation system may ensure the check of all details of a derivation, whereas currently many developments are presented with much "hand-waving" and, very often, cheating on essential details.

Also, a transformation system should contain a large body of knowledge, e.g. libraries of data types, rules, strategies etc. This means that much valuable information is available on-line and can be reused.

Finally, a transformation system allows one to treat a development as a formal object. Thus, developments can be "manipulated" to produce nice documentation (cf. [Partsch, 1988]), reused to solve similar problems, or abstracted into new transformation rules or strategies.

1.3 A utopian transformation system

This paper is meant as a reference to support the research into program transformation systems, rather than as a description of an actual system. Therefore, this document describes features of a utopian transformation system. We do not want to restrict ourselves in any way, hence no claim is laid on useful design paradigms like orthogonality, completeness, or even implementability.

1.4 Organisation of this paper

In the next section we discuss some existing program transformation systems. In section 3 the USTOPIA system is described by means of an informally stated list of requirements. In section 4 we discuss some of the features mentioned in section 3. There, it is pointed out which of these features are present in existing program transformation systems. Also, some thoughts are presented on features that seem to require more research before being implementable. Finally, in section 5 we draw our conclusions and indicate how our research in transformation systems may proceed.

2 Existing transformation systems

Current transformation systems are described in some detail in [Partsch and Steinbrüggen, 1983]. The interested reader is referred to this paper. Some systems not yet available at the time of writing of [Partsch and Steinbrüggen, 1983] have since been studied to some extent. While these systems all have their merits, they are also very much pioneering efforts. Hence they tend to be strong in certain areas, while other areas are not dealt with at all. Still, there is much to be learned from them.

2.1 CIP-S

CIP-S (cf. [CIP, 1987]) is an interactive, language-independent system to support

- the *derivation* of new program(-scheme)s from present ones by the application of transformation rules (which is to include the derivation of new rules within the system),
- the *reduction* of applicability conditions (including support for proofs by induction), and

- the *administration* of all system-specific entities (including the documentation and manipulation of program developments).

In [CIP, 1987], a formal specification of (the kernel of) CIP-S is given. Conceptually, this formal specification is based on the notion of a finite state machine. Following a well-known concept for specifying interactive systems, CIP-S comprises three major components: the user interface, the "core", and the knowledge base. The purposes of these system components are as follows:

- The *user interface* is responsible for the user/system interaction. In particular, it is to manage the translation between internal and external representations (parsing/unparsing) and the compilation of (complex) user requests ("transformation programs") into (basic) system operations.
- The *core* is the central component of the system that provides all basic operations that are needed for each of the above-mentioned activities *derivation*, *reduction*, and *administration*. Additionally it controls the internal system states and prepares reactions of the system to be conveyed as output to the user by the user-interface.
- The *knowledge base* is a collection of data depositaries, each consisting of a number of catalogues for
  - (global or local) transformation rules,
  - (predefined or user-defined) abstract data types, realised as signatures and transformation rules that correspond to the axioms of the respective types,
  - (temporary or permanent) program schemes,
  - program developments (development trees).

The core of CIP-S has been formally specified in CIP-L (cf. [CIP, 1985]) and developed by transformations into a set of programs at the level of PASCAL programs. For this transformational development the CIP prototype transformation system (cf. [Riethmayer et al., 1985]) was used. The extension of the core by language-dependent components (parser/unparser, catalogues of transformation rules, etc.) and a user-interface leading to a running system is the subject of an ongoing cooperation between TUM and Siemens corporation. A pilot version of CIP-S supporting CIP-L as an object language is operational.

In addition to the pure functional requirements, further (non-functional) requirements and constraints have been attempted in the design and the development of CIP-S:

- correctness ("transformational calculus", cf. [Pepper, 1984])
- reliability ("foolproof" system defined as a set of total functions; restricted mode of operation, dependent on current "activity"),
- extensibility with respect to functionality (clean hierarchical specification), and
- language-independence (appropriate parametrisation).

2.2 The PROSPECTRA system

The research topics of the ESPRIT project PROSPECTRA (PROgram SPECification and TRAnsformation, cf. [PROSPECTRA, 1987]) include:

- Engineering discipline for obtaining correct software: integration of program construction, formalisation of knowledge, *method bank*.

- Abstract formal specification, gradual introduction of detail, *replay*.
- Research consolidation and technology transfer. ADA as the centre of a common European technology base. Consolidation of convergent research in specification, verification and implementation.
- Industry of software components: reduction of software production cost by reusable components.

Therefore, a program transformation system for the language PA^{nn}dAS, a combination of ADA and ANNA (an annotation language for ADA, cf. [Luckham et al., 1987]) has been developed. This system consists of an integrated collection of tools, based on ESPRIT PCTE (Portable Common Tool Environment), with a uniform concept of user interaction. Many of these tools are based on the Cornell Synthesizer Generator (CSG, cf. [Reps and Teitelbaum, 1989]). The following components exist:

- a variant of PA^{nn}dAS which is used for controlling the PROSPECTRA system;
- a CSG editor for TRAFOLA-S, a variant of PA^{nn}dAS which can be used to describe transformation rules;
- a "transformer generator" which generates a CSG editor for PA^{nn}dAS with transformation rules from TRAFOLA-S descriptions;
- CEC, a system for completing algebraic specifications;
- an ML-based language for describing transformation rules, which is to enhance or possibly replace TRAFOLA-S;
- various libraries, etc.

A collection of transformation rules, a.o. the basic rules from [CIP, 1987], has been written in TRAFOLA-S. The system is embedded in X-Windows [O'Reilly et al., 1988].

More on the philosophy of the Prospectra project may be found in [PROSPECTRA, 1987]; a short survey of the system may be found in [Boiten et al., 1989].

2.3 The KIDS system

KIDS (Kestrel Interactive Development System, cf. [Smith, 1988]) is an interactive system that provides an open architecture for experimenting with various components for transforming formal specifications into correct and efficient programs. It works fully automatically when optimising programs and needs user interaction only in the "algorithm design" phase.

KIDS is implemented in REFINE which is also used as object language. REFINE is a commercial knowledge-based programming environment which provides

- an object-oriented data base used to represent "domain theories" (domain objects, relationships, constraints, laws),
- a grammar-based parser/unparser, and
- a very-high-level language (including transformation and pattern constructs).

Among others, KIDS provides tools that support

- deductive inference,

- algorithm design,
- expression simplification,
- finite differencing, and
- partial evaluation/specialisation.

The central component of KIDS is the general-purpose deductive inference system RAINBOW II. This subsystem comprises a knowledge-base consisting of approximately 300 rules for reasoning over program expressions and a facility to apply these rules to expressions. RAINBOW II allows the inference of sufficient conditions (by "backward reasoning") as well as necessary conditions (by "forward reasoning") of formulas. Inference of equalities and (lower) bounds are included as special cases. Specifically for program development the following tasks are supported by RAINBOW II:

- canonicalisation,
- formula verification and first-order theorem proving,
- expression and formula simplification,
- constraint propagation,
- finite differencing.

Program development with KIDS starts with an explicit statement of the "domain theory" (i.e., properties reflecting particular knowledge of the respective problem) and a formal specification built on top of it. The system then applies specialised, built-in tactics (e.g. divide-and-conquer or global search), selected by mouse from a menu, to subexpressions also selected by mouse. Partially implemented specifications are augmented with input assumptions, invariants, and output conditions, and shown to the user in a particular window. The result of a development in KIDS is a recursive REFINE program which is then further compiled into COMMON LISP.

KIDS has been used so far for many sample developments, including: enumeration problems involving global search, job scheduling, graph colouring, covers of vertices and sets, knapsack problems, travelling salesman tours, and the k-queens problem.

Although the system worked quite satisfactorily for these examples, it is still an experimental system. For the future it is planned to extend the system by

- more advanced optimisation algorithms,
- automatic data structure selection,
- (ground and parametrised) tactics,
- a more elaborate methodology (checklist for standard scripts), and
- performance-directed design.

3 A list of requirements

Our list of requirements will be divided into three parts. The first one treats the capabilities of the utopian transformation system, i.e. the "heart" of the system. The requirements regarding the user interface are described in the second part. The third part concerns itself with some behavioural aspects of the system. In the fourth part, the role of a transformation system within an integrated project support environment is described. For reasons of readability the list of requirements has an informal nature.

3.1 Capabilities

The desirable capabilities of an "ideal" program transformation system as we envisage it are summed up below. This list is based on our own experiences with "pen and paper" derivations, in which many clerical steps appear to be automatable. In the list below we outline the necessary support for the automation of such clerical tasks. It is based as well on our analysis of existing program transformation systems and other programming environments.

Our ideal of a program transformation system can be viewed along three different dimensions, each of which comprises a well-known subject in computing science:

1. program (or problem) specifications and their analysis, validation and verification;
2. the process of transformational programming;
3. "powerful" computing systems in general.

In these three "dimensions", important parameters are *language*, *logic*, and *model*, respectively.

These dimensions will serve as a guideline when summing up relevant aspects of the program transformation system below.

Projections of the requirements below to any two of these dimensions correspond to:

- 1 & 2: The ("pen and paper") method of transformational programming.
- 1 & 3: A traditional programming environment.
- 2 & 3: An expert system shell.

We are striving to combine 1 & 2 & 3.

3.1.1 Specifications and their analysis

Writing specifications

Any transformational development must begin with some sort of specification of the problem to be solved, or the program to be written. Hence a transformation system should provide support for writing specifications. Since a rigorous development is aimed at [Dijkstra, 1976, Jones, 1980], we envisage support for one or more specification formalisms. To avoid overreaching we restrict ourselves for the rest of the paper, unless explicitly stated otherwise, to two specification and transformation formalisms as *leitmotiv*, viz. CIP-L, including algebraic specifications [CIP, 1985, Bergstra et al., 1989], and the Bird-Meertens formalism [Bird, 1987, Meertens, 1986], referred to as *BMF* in the rest of this paper. The choice of these two formalisms is rather pragmatic: with these two the authors have built up extensive experience. Apart from that, these formalisms take extreme positions in the area of specification. BMF specifications are usually defined and determinate; CIP-L specifications need not be. BMF has a lazy semantics, while CIP-L has a strict one. BMF is mostly used to describe algorithms on lists and related structures, while CIP-L is intended to cover arbitrary descriptive and operational specifications over all possible data types.

Support for both of these formalisms could be provided by:

- a pre-defined collection of basic data types, to avoid the tedious job of specifying yet again e.g., the Booleans. This collection can be viewed as a library for software reuse;
- basic operators and laws of BMF [Bird, 1987];
- support for constructing specifications (by combination or extension) from existing ones;

- some specialised support, e.g., for algebraic specifications, checks on consistency and completeness of abstract data types;
- support for formalising informal requirements.

Analysing specifications

Since a formal specification is generally derived from an informal one, it needs to be validated. This can be done in a number of different ways. One may want to prove additional properties of a specification. Thus, the following activities need to be supported:

- reasoning about definedness, determinacy, or other aspects of formal specifications;
- the analysis of operational specifications w.r.t. complexity, etc.;
- formal interpretation of specifications, including non-operational constructs like **some**-expressions, and with possibilities for *tracing* for "debugging" specifications;
- (rapid) prototyping of abstract data types for the validation of informal specifications, and to provide a check on the practical value of the final program;
- in combination with the former, interpretation (reduction) of *program schemes*;
- compilation of a reasonable subset of specifications;
- change of representation, like the translation to English or graphical representation of a specification, to provide a version readable to the customer. For graphical representations, one could also think of structure diagrams, signature diagrams, etc.

3.1.2 Transformational programming

Derivation

Since the derivation process is the main activity of the programmer, tools for its support are desirable. Support should be given for the:

- explicit application of one transformation rule, viz.
  - the selection (graphical, path expressions) of a program fragment to which the transformation is to be applied;
  - the selection of a rule to be applied or proposed for application, e.g., with pattern recognition;
  - the reduction or proof of applicability conditions, or at least recording the need for the proof of these conditions.
- implicit application of one transformation rule, i.e. by stating a program equivalent to the current one.
- combined application of multiple transformation rules, viz.
  - by way of transformational expressions or a transformation language;
  - the application of standard strategies, like divide and conquer [Smith, 1988], or the elimination of tail recursion;
  - the notions of focus and status, i.e., the localisation of the program fragment under consideration and the part of the strategy currently elaborated and the possibilities of user interaction.

- introduction of transformation rules.
  During a derivation one often wants to introduce and prove some specialised transformation, comparable with a lemma during a mathematical proof. One should be reminded by the system of the proof obligation, and aided in delivering the proof.

Furthermore, it should be possible to do the development in a way that reflects the logical structure of the derivation.

Verification

This concept plays a central rôle in program transformation, since many transformation rules are only valid under certain applicability conditions. Therefore, the following activities need to be supported:

- the proof of
  - applicability conditions, and
  - transformation rules,

  using a proof system including comprehensive possibilities for backward reasoning.
- delayed proof of applicability conditions and keeping track of these proof obligations.

Advice & automation

In order to reduce the amount of work to be done by the user, the system should provide hints and advice on possible directions for a derivation. For this one could think of:

- options for automatic canonicalisation ("simplification");
- "jittering", automatic adaptation of a program to fit a transformation rule;
- the automated checking of applicability of a rule, e.g. by pattern matching and reducing applicability conditions, and "proposals" for possible transformation steps;
- guidance for the selection of rules and strategies, based on complexity checks or other heuristics.

Evaluation of developments

It should be possible to evaluate the development process, in order to learn from mistakes and to add successful transformation rules or strategies to the available repertoire. In this context, the following activities should be supported:

- manipulation of formal derivations to adapt them to new circumstances ("reusability");
- the generalisation of developments, in order to abstract strategies or rules for more general use;
- collecting all laws and assumptions used in a derivation;
- maintenance of a history in order to aid modification of the program and/or its development (e.g., by keeping track of the strategies which have been tried or could still be tried);
- replay of parts of the development process to aid the above and to follow alternative strategies if desired or needed.
- the generation of papers or other written documentation during the transformational development in a convenient way ("literate programming" [Knuth, 1984]).

3.1.3 A powerful system

Knowledge base

One of the main advantages of a program transformation system is the availability of a large amount of knowledge on transformation techniques, etc. (a "method bank"). Not only should data types, transformation rules, strategies, etc. be stored, but they should also be accessible in a convenient way. Furthermore, they should be appropriately documented, in such a way that they can be used by any user of the system. It is insufficient to rely on the knowledge of transformation strategies of all but the most expert user.

Also, the program to be produced itself should be documented. This should be automated as much as possible. One needs:

- a library or database of (basic) data types, laws, operators, rules, and strategies, etc.;
- a tool to provide and store documentation for (new) rules and strategies with the possibility of generating dictionaries, indexes, cross-references, etc.

The system should also support the development of large programs, so some form of version management should be available.

Editing facilities

A powerful language-based editor is necessary for entering specifications, transformation rules, etc. It should also allow easy modification of specifications (e.g., restructuring, generalisation, specialisation). Furthermore, it should also recognise *program schemes* ("contexts").

3.2 User interface

A system with a user interface which is both pleasant and easy to use will attract more users. Hence effort put into the user interface is effort well spent. On the other hand, a full scale development effort on the user interface is not desired, since the focus of our research is on capabilities. Existing software should therefore be used wherever possible. A list of interface aspects is given below.

- A *windows* based system is clearly an advantage here, since subdevelopments and data base references for rules or applications could use their own window. For portability reasons some standard system seems to be the best choice, e.g., X-Windows [O'Reilly et al., 1988]. Furthermore, it would be useful if a hypertext-like facility were available. This allows the uncomplicated unfolding of information on the screen.
- The user interface should be *adjustable*, to accommodate for personal taste and for research into different views on the ease of use.
- A *focusing* facility is needed to switch from the global development to subdevelopments and back.
- It should be possible to perform some simple operations in a *graphical* way by highlighting the components of common transformations. It may be rather too futuristic at the moment, but a holographic user interface could actually perform fold/unfold transformations on the screen.
- The *metaphor* employed by access facilities should be such that every kind of user can do at least anything he could do with pen & paper. In connection with the latter point good *help facilities* are needed.

- In order to allow for more concise notations (e.g., in a "squiggly" kind of transformational development [Bird, 1987]), a *large* and *extensible character set* (e.g. *Metafont*) should be available.

3.3 System aspects

The system should exhibit all the attributes of a well engineered software product, as stated for example in [Fairley, 1985, Sommerville, 1989]. This is especially important, because the system should run on different machines, and it should be possible to instantiate it with different specification languages. We list a number of main points:

- The system should be *modular*, and well *integrated*.
- It should be *extensible* and *modifiable*.
- The system should be *robust* and *reliable*.
- It should be *integratable* with relevant other systems, such as editors, compiler generators, file systems, etc.
- The system should be *fast*. If and when it cannot be, the user should get something to read every now and then, to "prove" that the system is doing hard work.
- It should be *portable* (that seems to imply **C** under UNIX and X-Windows). Also, it should be relatively easy to install the system.
- Nevertheless, the size of the system should be such that it can be
  - installed on every reasonably powerful machine (workstation) and
  - run without causing innumerable page faults.

3.4 USTOPIA within an Integrated Project Support Environment

The main idea of this section is to investigate whether it is useful to consider the possibility of having USTOPIA as a subsystem of an Integrated Project Support Environment (IPSE). The aim of an IPSE is to provide an environment for developing large software systems by integrating a set of tools which support a certain development methodology [Brown, 1988]. An IPSE supports both program development and the management aspects of software development of several people at the same time. USTOPIA as described in the previous sections does not support the management aspects of the software development and it will also be a single user system. It could be possible to have USTOPIA as a subsystem within some IPSE.

The trend in software engineering is towards the development of IPSE's. The base of an IPSE is a database in which all relevant information about the software project is recorded, such as the relationship between a specification and its implementation in some programming language. Furthermore it contains facilities for the communication between the project members. Only a few of these IPSE's incorporate a transformational tool, to allow the formal derivation of software from a specification. However, there are already some systems which allow the formal specifications of software requirements.

The USTOPIA system in an IPSE could be used as a tool to support the formal specification and derivation of software by transformations. The transformation rules used should be stored in the underlying database of the IPSE. New correct transformation rules, strategies and tactics can also be stored in this database, so each user of the USTOPIA tool can make use of the rules derived by fellow users. It may even be possible to store parts of the derivation in order to reuse them later in another derivation.

4 Realisation aspects

From the implementation point of view, the requirements presented in the last section range from easily implementable to unsolvable tasks. Also, it is obvious that some of the requirements are contradictory. In the following section, we will list a number of implementation issues of existing resp. futuristic systems, which promise to be interesting for further research. Following the ordering of the previous section, we will first look at single requirements, and then comment on aspects concerning the whole system.

4.1 Tools for the design of specifications

Every implementation of a transformation system will in the end support some specification formalism, and will hence provide the basic constructs of that formalism. Following the paradigm of reuse, the user should also have access to already existing specifications. This leads to a process of developing formal specifications which should help to lessen the gap between formal and informal requirements. At present only a few systems support the specification process at all (PLUSS: [Bidoit et al., 1987], SAFE: [Balzer et al., 1980]). However, this seems to reflect a lack in the underlying methodology, rather than basic difficulties in the realisation of appropriate tools.

The efficient incorporation of existing specifications makes particular demands on the specification formalism, such as possibilities for parametrisation and modularisation. For algebraic specifications, there has been much research on this issue, and a number of textbooks which include sections on it have recently been published [Ehrig and Mahr, 1985, Ehrich et al., Bergstra et al., 1989]. Nevertheless, there still seems to be work left with respect to the integration and, even, standardisation of the techniques.

The system should ensure wellformedness of the initial specification. The required methods and tools are fairly well understood in the case of classical syntactical correctness, i.e. the building of parsers, syntax directed editors, etc. There are still a number of interesting open problems in connection with more recently developed formalisms. For polymorphic, functional languages which allow higher order functions, there are for example questions surrounding the implementation of efficient typing algorithms, see for example [Henglein, 1989]. For algebraic specifications a number of open issues concerning completeness and consistency remains. An overview of recent results and problems in this area can be found in [COMPASS, 1989, chapter 2.5.4].

4.2 Realisation of specification analysis

It is well known that most semantic properties of specifications such as definedness, determinacy, strictness or complexity behaviour are in general undecidable. Hence, only results within a specific area of specifications, or of a stochastic or otherwise restricted nature can be expected. The first path has for example been taken by the builders of the RAPTS-system [Paige and Cai, 1987]. In this system, it is possible to construct functions which can automatically be shown to be of linear time and size complexity (in the input and output space).

One of the first systems aimed at average case complexity analysis of functional programs was [Wegbreit, 1975]. In [Zimmermann, 1988], its ideas are transported to the context of a simple *typed* functional language. These kinds of systems can usually also be used to obtain upper bounds for the worst case behaviour of programs. However, without user support, these bounds tend to be fairly imprecise even with relatively simple programs. In [Hickey and Cohen, 1988], some of the problems with the complexity analysis of programs have been attacked from the theoretical side by giving a probabilistic semantics for (functional) programs and using probabilistic attribute grammars to model input distributions. Looking at the equations generated by their hypothetical system, it seems that support by a symbolic processor would be an indispensable requisite of any system using this approach.

Up to now, these approaches seem to have a fairly experimental nature and serve, in a way, to show the problems arising during the analysis of a program. In the context of the synthesis of programs, this seems to make the case for tools which do the complexity analysis of programs alongside the transformation process. The effect of many transformations with respect to definedness and determinacy can be determined, see for example the theory developed in the frame of the CIP-project [CIP, 1985, CIP, 1987]. Hence, systems which help the user to keep track of at least these properties seem to be within reach. However, at present we do not know of any general purpose transformation system which supports this kind of activity. Despite the fact that the effect of transformations on the complexity is in general undecidable, future research should try to establish a calculus measuring more quantitative effects of (certain) transformations.

In recent years, it has emerged that one of the major benefits of formal specifications can be their use for prototyping and, hence, early validation of specifications. Such a prototype is of course immediately obtained as a byproduct of a specification in an executable specification language such as a functional language. The problem with the use of such a deterministic formalism for specification is of course the *overspecification* which is often implied by determinacy. Hence, in recent years there has been research about the integration resp. extension of such formalisms to relational calculi, see for example the enrichment of the BMF-formalism discussed in [Bird et al., 89] or the Ruby language [Sheeran, 1990]. Prototyping in such a language is of course more difficult. It seems reasonable to expect that many of the techniques employed in the interpretation of PROLOG programs could be used in this context as well. Note that because of the correspondence between relations and many-valued functions, this would also show the way to the interpretation of choice-constructs like the **some**-expressions in CIP-L.

Quite a number of implementations for 'executing' algebraic specifications have been developed in the last few years. These include interpreters as well as – more recently – compilers. Most of these systems have been based on term rewriting and narrowing. As an attempt to make use of common subterms, graph grammars (containing *graph* transformation rules) have also been introduced. They can reduce the number of rewrite steps and amount of space needed during a rewrite process at the expense of some administrative overhead. Despite all this, it seems that a lot more effort will be needed to increase the efficiency of the prototypes based on either of these techniques. For a list of references, see [COMPASS, 1989, chapter 2.3.2].

One of the more general transformations which can be useful for the evaluation of specifications is partial evaluation [Bjørner et al., 1988]. A related principle is the symbolic evaluation propagated in [Cheatham et al., 1979]. We will make more comments on the techniques related with these kind of transformations in the following section.

There are a number of tools which have been useful to make formal specifications more understandable to 'non experts'. This includes translators to English [Swartout, 1982, Ehler, 1985] as well as tools to give more general behaviour explanations [Swartout, 1983].

4.3 Support for the transformational process

By definition, transformation systems should support the basic activities occurring during a transformational development, such as the application and introduction of transformation rules. Most existing systems perform these and other clerical jobs more or less satisfactorily. Experience with their use has revealed a number of inconveniences.

- The amount of user interaction needed can be fairly high. This is partly due to the length of transformational developments. Partly this seems a user interface problem.

- The interaction can sometimes be technically difficult. A typical problem is the proof of applicability conditions.
- It is not easy to choose the 'right' transformation, in particular if this is to be chosen out of some huge catalogue.

One method of trying to solve these problems lies of course in letting the machine do more work. We list in the following some of the approaches which have been made in this direction. We will group them according to the framework they are based on.

1. Language and type.
   One way of shortening the transformational development is by adding an extra language level, i.e. by establishing a transformational language. This has been done in a number of systems including the Prospectra System described in section 2, and the DEVA system developed in the course of the ToolUSE project [DEVA, 1989]. In these projects, a uniform approach to program and meta-program development is advocated. More remarks on the underlying methodology can be found in [Krieg-Brückner, 1988]. In [Darlington, 1981], the functional language HOPE is used for similar purposes. Despite all this, it should be mentioned, that there has not been much practical experience with this method yet.

   Most systems provide a tool to signal rules which are applicable within a certain part of the specification. This can be implemented with the help of pattern matching based on the type system. Of course, this only concerns those rules whose applicability can be derived from syntax and type information only, i.e. without semantic checks. Hence, in general it will only be possible to exclude certain transformation rules which cannot be applied. It is not clear to which extent this is done incrementally in existing systems.

2. Rewriting.
   Provided the question of applicability can be solved, it is of course possible to perform certain transformations automatically. Especially if this is done conditionally, it has strong relations with existing rewriting. The resulting processes can be used to simplify expression for the user. (Semi) automatic transformation systems can be seen as rewrite systems. This includes, as an extreme case, compilers which automatically perform optimisations. Jittering, i.e. the slight modification of programs to fit certain transformation rules [Fickas], is a related application.

3. Theorem proving.
   Most specification languages contain logical constructs. Hence the transformation system should support calculations within a general logical framework. This could be provided by the integration of a conventional theorem prover such as the LCF or the Boyer-Moore theorem proving system [Paulsen, 1987, Boyer and Moore, 1979].

   More specific needs for deductive capabilities arise during the verification of applicability conditions and transformation rules. Resolving these questions involves semantic issues. A deductive system to support these activities will be dependent on the semantics and the logic of the specification language, i.e. the possible transformations.

   Further tasks for deductive systems arise from complexity analysis. As argued above, within a transformational style of programming this is probably best done incrementally. Hence, complexity can be seen as an annotation of a specification, cf. [Wupper and Vytopil, 1989]. The updating of this or other annotations of a specification will usually demand deductive capabilities.

Within (semi)-automatic systems, all of the above deductive tasks occur. Hence, for example 'RAINBOW II', the 'inference engine' of the KIDS system provides tools for them as well as for optimising the resulting trees of developments.

The extreme length of transformational developments is partly due to the fact that they often consist of many, small transformation steps. One possible approach to overcome this is the use of more compact transformation rules. However, this has the disadvantage of increasing the knowledge and choice expected from the user. This has motivated the introduction of 'transformation strategies', such as 'try finite differencing'. The transformation languages introduced above can be seen as a tool to explicitly formulate such strategies. In (semi) automatic systems such as KIDS or RAPTS, the user usually only has the choice between the built in strategies.

4.4 Support for the evaluation of developments

No serious work has been done on supporting the evaluation of complex developments, other than traditional clerical support. This is due to the fact that the work on the underlying methodology has only just begun.

4.5 System aspects

It is a well-known fact that the requirements of a well engineered software product cannot all be met to the same degree, because they are inherently contradictory. So, we are faced here with the dilemma between formalism independence (parametrisation), flexibility and convenience on the one side, and efficiency, compactness, and portability on the other.

5 Conclusions and further work

We hope that this survey has been of help in identifying important problems and design decisions in the area of program transformation systems. To us, it has certainly become clearer what constitutes a program transformation system.

Also, we have pointed out a number of principal problems, and other problems that may instigate further, theoretical and applied, research.

In our opinion, the construction of yet another transformation system will not be a very useful activity. Many basic systems exist, and we hope to gain experience with more of them in the nearby future. It appears to be more interesting to extend an existing system with some more advanced features, like those mentioned in section 3.1.2.

Acknowledgments

We would like to thank H. Meijer and J. Sarbo for their contribution to the discussions on transformation systems, and L. Meertens for completing the USTOPIA-acronym.

References

[Balzer et al., 1980] R. Balzer, N. Goldman, and D. Wile. Informality in program specifications. *IEEE Transactions on Software Engineering*, 4(2):94–103, 1980.

[Bergstra et al., 1989] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. Addison-Wesley, New York, 1989.

[Bidoit et al., 1987] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? An experiment with the PLUSS specification language. Rapport Interne N.343, LRI, April 1987.

[Bird, 1987] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design. NATO ASI Series Vol. F36*, pages 5–42. Springer-Verlag, Berlin, 1987.

[Bird et al., 89] R.S. Bird, R.C. Backhouse, G. Malcolm, O. de Moor, J.T. Jeuring, G. Jones, M.M. Fokkinga, M. Sheeran, and L.G.L.T. Meertens. International summer school on constructive algorithmics, Ameland, September 1989. Lecture notes, 1989.

[Bjørner et al., 1988] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland Publishing Company, 1988.

[Boiten et al., 1989] E. Boiten, S. Pemberton, and H. Vogt. Het Prospectra systeem in Bremen. STOP Internal Note (in Dutch), March 1989.

[Boyer and Moore, 1979] R.S. Boyer and J.S. Moore. *A Computational Logic.* ACM Monograph. Academic Press, 1979.

[Brown, 1988] A.W. Brown. A view mechanism for an integrated project support environment. Report 275, University of Newcastle upon Tyne, 1988.

[Cheatham et al., 1979] T.E. Cheatham Jr., G.H. Holloway, and J.A. Townley. A system for program refinement. In *Proceedings of 4th International Conference on Software Engineering, Munich, West Germany, September 1979*, New York, 1979. IEEE.

[CIP, 1985] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, *Lecture Notes in Computer Science* **183**. Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[CIP, 1987] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Volume II: The Transformation System CIP-S*, *Lecture Notes in Computer Science* **292**. Springer-Verlag, Berlin, 1987.

[COMPASS, 1989] ESPRIT Basic Research Working Group No. 3264 COMPASS. A COMprehensive Algebraic approach to System Specification and development. Objectives, state of the art, references. Bericht 6/89, Universität Bremen, 1989.

[Darlington, 1981] J. Darlington. The structured description of algorithm derivations. In J. de Bakker and H. van Vliet, editors, *Algorithmic Languages*, pages 221–250. North-Holland Publishing Company, Amsterdam, 1981.

[DEVA, 1989] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. Definition 1.1 of the generic development language Deva. Report, Université Catholique de Louvain, Faculté des Sciences Appliquées, Unité d'Informatique, 1989.

[van Diepen and Partsch, 1990] N.W.P. van Diepen and H.A. Partsch. Formalising informal requirements, some aspects. In L. Feijs and J. Bergstra, editors, *Proc. METEOR Workshop, Mierlo 1989*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1990.

[Dijkstra, 1976] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Ehler, 1985] H. Ehler. Making formal specifications readable. Technical Report TUM-I8527, Institut für Informatik der T.U.München, 1985.

[Ehrich et al.] H.-D. Ehrich, U.W. Lipeck, and M. Gogolla. *Algebraic Specifications.*

[Ehrig and Mahr, 1985] H. Ehrig and B. Mahr. *Foundations of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[Fairley, 1985] R.E. Fairley. *Software Engineering Concepts*. McGraw-Hill, New York, 1985.

[Feather, 1987] M.S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T Meertens, editor, *Program Specification and Transformation. Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation*, pages 165–196. North-Holland Publishing Company, Amsterdam, 1987.

[Fickas] S.F. Fickas. *Automating the Transformational Development of Software*. PhD thesis, Univ. of California, Irvine.

[Henglein, 1989] F. Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, 1989. Technical Report 443.

[Hickey and Cohen, 1988] T. Hickey and J. Cohen. Automating program analysis. *Journal of the ACM*, 35(1):185–220, January 1988.

[Jeuring et al., 1988] J. Jeuring, L. Meertens, and S. Pemberton. A transformation system. STOP Internal Note, 1988.

[Jones, 1980] C.B. Jones. The role of formal specifications in software development. In P.J. Wallis, editor, *Life-cycle Management, Infotech State of the Art Report*. Pergamon Infotech Ltd., Maidenhead, 1980.

[Krieg-Brückner, 1988] B. Krieg-Brückner. Algebraic formalisation of program development by transformation. In *Proc. European Symposium On Programming '88, Lecture Notes in Computer Science* **300**, pages 34–48, 1988.

[Knuth, 1984] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Luckham et al., 1987] D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, and O. Owe. *Anna, a Language for Annotating Ada Programs, Reference Manual*, *Lecture Notes in Computer Science* **260**. Springer, Berlin, 1987.

[Meertens, 1986] L.G.L.T. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, *CWI Monographs* **1**, pages 289–334, 1986.

[O'Reilly et al., 1988] T. O'Reilly, V. Quercia, and L. Lamb. *The Definite Guide to the X-Windows System*. O'Reilly & Associates, Inc., 1988.

[Paige and Cai, 1987] R. Paige and J. Cai. Building performance at language design time. In *Proc. ACM Principles Of Programming Languages*, 1987.

[Partsch, 1988] H.A. Partsch. Generalize and reuse. An exercise in reusing transformational developments. In M. Broy and M. Wirsing, editors, *Methodik des Programmierens. Eine Festschrift zu Ehren von F.L. Bauer, MIP-8915*. Fakultät für Mathematik und Informatik, Universität Passau, 1988.

[Partsch and Steinbrüggen, 1983] H.A. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, 1983.

[Paulsen, 1987] L.C. Paulsen. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pepper, 1984] P. Pepper. A simple calculus for program transformation (inclusive of induction). Technical Report TUM-I8409, Institut für Informatik der T.U.München, 1984.

[PROSPECTRA, 1987] B. Krieg-Brückner, B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möncke, B. Weisgerber, A. McGettrick, I.G. Campbell, and G. Winterstein. PROgram development by SPECification and TRAnsformation. In *Proc. ESPRIT Conf. '86*. North-Holland Publishing Company, 1987.

[Reps and Teitelbaum, 1989] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a system for constructig language-based editors*. Springer Verlag, Berlin, 1989.

[Riethmayer et al., 1985] H.O. Riethmayer, F. Erhard, and H. Ehler. User manual for the CIP-System-Prototype. Technical Report TUM-I8511, Institut für Informatik der T.U.München, 1985.

[Sheeran, 1990] M. Sheeran. Ruby - a language of relations and higher order functions. In *Proc. 3rd Banff workshop on Higher Order*, Lecture Notes in Computer Science. Springer, 1990. To appear; also in [Bird et al., 89].

[Smith, 1988] D.R. Smith. KIDS – a knowledge-based software development system. In *Proc. Workshop on Automating Software Design*, pages 129–136. Morgan-Kaufmann, 1988.

[Sommerville, 1989] I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, Mass., 1989. 3rd edition.

[Swartout, 1982] W. Swartout. GIST English generator. In *Proc. of Amer. Assoc. for Art. Int. 82*, pages 404–409, August 1982.

[Swartout, 1983] W. Swartout. The GIST behavior explainer. In *Proc. of the 1983 National Conference on Artificial Intelligence*, pages 402–407, Washington, D.C., 1983. AAAI.

[Wegbreit, 1975] B. Wegbreit. Mechanical program analysis. *Journal of the ACM*, 18(9):528–539, 1975.

[Wupper and Vytopil, 1989] H. Wupper and J. Vytopil. A specification language for reliable real-time systems. In J. Zalewski and W. Ehrenberger, editors, *Hardware and Software for Real Time Process Control*. Elsevier Science Publishers B.V., 1989.

[Zimmermann, 1988] W. Zimmermann. How to mechanize complexity analysis. Technical report, GMD Forschungsstelle Karlsruhe, 1988.