Tool Support for Formal Software Development: A Generic Architecture

Maritta Heisel, Thomas Santen, Dominik Zimmermann

Technische Universität Berlin* and GMD FIRST**

Abstract. We present a formalism independent approach to the design of tools supporting the application of formal methods in software development. It consists of a concept to represent problem solving knowledge, called *strategy*, and a generic architecture showing how to implement tools for strategy-based development. A prototype system for program synthesis demonstrates the practicality of the approach.

1 Introduction

Today, formal methods for software development are at the edge of entering industrial practice. The theory of formal specification and verification of software is well understood, and an increasing number of case studies in industrial context are performed to evaluate cost and use of the application of formal methods [5]. Especially in safety-critical applications they are recognized as one technique to support development of highly dependable software.

In this situation, an increasing number of non-experts in the field have started to use formal methods, and thus tool support is of growing importance. Most existing tools are parsers, type checkers and documentation tools for specifications, or theorem provers for the underlying logics. Only few provide support for the *methodological* aspects of formal methods. But non-experts have to rely on guidance to set up formal specifications, demonstrate their properties, and develop code from specifications in a provably correct way.

The present paper addresses the problem of how to design tools to support the process aspect of software development specific to formal methods. We introduce a concept representing a "method" in a way that allows us to provide machine support for its application. We also present a system design for the implementation of this concept. A prototype system for program synthesis demonstrates the practicality of our approach.

We do not see formal methods as a means to replace traditional software engineering. Put into practice, they will only be one technique among others to enhance software quality. Our approach therefore focuses on tools specific to support application of formal methods. It is not intended to replace but to *complement* existing CASE technology.

* Softwaretechnik (FR5-6), Franklinstr. 28/29, D-10587 Berlin, Germany. e-mail: {heisel,dominik}@cs.tu-berlin.de

** Rudower Chaussee 5, D-12489 Berlin, Germany. e-mail: santen@first.gmd.de

Requirements for Formal Methods Specific Tool Support

In general, there are two conflicting goals in the design of tools specifically for formal methods. In contrast to classical software engineering, such a tool must be designed to guarantee semantic properties of the resulting product, e.g. correctness with respect to a specification. Therefore it must enforce certain ways of procedure. On the other hand, it has to provide as much freedom as possible for the developers and must not hinder creativity. From these goals, we deduce the following requirements:

Guarantee Semantic Properties. A tool must support the development process in a way that eases rigorous mathematical reasoning and establishes confidence that the product indeed fulfills the required properties. Two aspects contribute to establishing confidence: first, there must be a clear identification of the steps in the development process that are crucial to establishing semantic properties. Second, since the development support tool will inevitably contain errors it must be designed to provide insight into the "semantically relevant" components and their interaction.

Balance User Guidance and Flexibility. Formal methods usually consist of some mathematical formalism and a variety of more or less explicitly stated techniques how to use it. Due to syntactic constraints and mathematical rigor, their application tends to be non-trivial. It is therefore important not to leave the user alone with a mere formalism but to develop explicit techniques to guide its use and offer the user choice of tried and tested approaches on how to proceed. In order not to unnecessarily restrict its users, a tool must support the *combination* of such techniques. Furthermore, it must also be *customizable* by informed users who develop specialized techniques for their project contexts.

With or without formal methods, several attempts are usually needed to solve a problem in a satisfying way. A tool for formal methods should provide means to *explore alternative ways* to a solution. It should enable judging the feasibility of an approach as early as possible.

For classical software engineering, support for *multiple developers* is standard. The main problem is to maintain consistency of the resulting documents. For formal methods, the consistency problem appears in a sharper sense: how can work be distributed in a way that ensures the results can be combined and properties guaranteed with reasonable effort? A development tool should provide information about "safe" ways to parallelize work.

Provide Overview of Development. Exactness and rigor entail a higher level of detail that must be handled. It is crucial for developers to have tool support that provides an *overview* of the development process and the relations between subtasks. They must avoid roundabout ways and dead ends in the development that may make proof of properties practically infeasible if not theoretically impossible. The task here is to design the tool so as to maintain the necessary

information that can be used to provide a supportive user interface.

We wish to identify general concepts that are applicable to a variety of formalisms. The contribution of the present paper is a *formalism independent approach* to the design of tools that support the peculiarities of formal methods.

The results of this work are as follows:

- We introduce the concept of *strategy* as a knowledge representation mechanism which makes development knowledge amenable to machine support. Methods are represented as sets of strategies.
- A *uniform interface* between strategies facilitates their modular implementation. It makes the combined application of methods possible and enhances the adaptability of a support tool to new and improved ways of procedure.
- A *generic architecture* shows how to implement support tools for strategy-based development. This architecture is designed to meet the requirements expressed above.

In the rest of the paper, we proceed as follows: in Sect. 2, strategies are introduced. Section 3 presents a general overview of the architecture, followed by a description of its components: implementation of strategies (Sect. 4), internal data structures (Sect. 5), and data and control flow (Sect. 6). We describe an implemented program synthesis system as an instance of the system architecture in Sect. 7. We look at related work in Sect. 8. In the concluding section, we discuss how our approach meets the above requirements and mention implications to future research.

2 Representing Development Knowledge by Strategies

There are two aspects to a method for software development: *strategies* and *heuristics*. Strategies describe possible steps during a development. Examples are how to decompose a system design to guarantee a particular property, how to conduct a data refinement, or how to implement a particular class of algorithms. Strategies are the part of a method that is usually described in text books. In contrast, the ability to decide which strategy may successfully be applied in a particular situation requires human intuition and a deep understanding of the problem at hand. The rules of thumb that experts develop when working with a technique, we call the heuristic part of their method.

While heuristics are hardly mechanizable, strategies can be implemented. Our system architecture is therefore designed to support problem solving by application of strategies in an interactive environment that supports experts in using their heuristic knowledge.

Technically, the purpose of a strategy is to find a suitable solution to some software development problem. A strategy works by problem reduction. For a given problem, it determines a number of subproblems. From the solutions to

these subproblems the strategy produces a solution to the initial problem. Finally, it tests if that solution is acceptable according to some notion of acceptability of a solution with respect to a problem. The solutions to subproblems are naturally obtained by strategy applications as well.

However, this description is too general to be of much use. It is even inadequate in its simplicity because it says nothing about interdependencies between the various subproblems and solutions. An example from program synthesis serves us to motivate the more detailed description of strategies given in Sect. 2.2.

2.1 An Example: Synthesis of Divide-And-Conquer Algorithms

Before we can describe concrete strategies for a specific area of software development, we have to establish the notions of problem, solution and acceptability. In the context of program synthesis, problems are specifications of programs. Accordingly, solutions are programs in some programming language, and a solution is acceptable with respect to a problem only if the program meets the specification. In general, we do not need more assumptions on the specification language or on what it means that a program meets a specification. A specification may encompass functional requirements as well as constraints on time and space complexity of the resulting algorithm.

As an example, we consider an approach from literature to synthesize divide-and-conquer algorithms [22], where problems are functional requirements. Solutions are programs in some functional programming language, and a program is acceptable if and only if it is totally correct with respect to the specification. A divide-and-conquer algorithm can be represented by a schematic definition of a recursive function:

$$f(x) \equiv \textbf{if } \mathit{primitive}(x) \textbf{ then } \mathit{directly\_solve}(x) \\ \textbf{else } (\mathit{compose} \circ (g \times f) \circ \mathit{decompose})(x)$$

where $g = f$ or $g = id$ (the identity function).

This schema describes a flow of control that is characteristic of divide-and-conquer algorithms: if some *primitive* predicate holds, the problem can be solved directly. Otherwise, the input has to be decomposed into two parts. Depending on the way *decompose* works, the function f is either recursively applied to both parts of the input ($g = f$) or to one part and the other one is left unchanged ($g = id$). Finally, the results yielded by f and g are composed to make up the final result of the algorithm.

Smith [22] describes several "strategic" ideas on how to develop divide-and-conquer algorithms by filling the gaps in the schematic algorithm. One of them is shown in Fig. 1. The idea is to develop the decompose-recursion-compose part from front to back.

Consider the problem of sorting a list.[3] The first thing to do is to find an algorithm in a library that reduces the length of the list. One possible solution is

[3] We are aware of the fact that to solve a problem like this our framework is not necessary. Due to space limitations, it is not possible to present a non-trivial example.

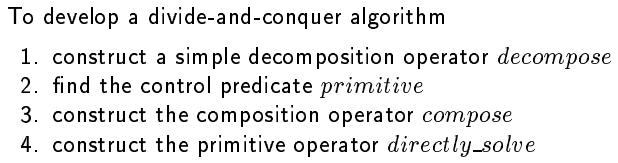

Fig. 1. A divide-and-conquer strategy

listsplit which splits an input list into two halves. Once we have decided on the decomposition function, we can determine the test to terminate the recursion: *listsplit* is applicable only if the input has at least length two.

Selecting *listsplit* also has consequences for the recursive case. Both halves of the input list have to be sorted. Hence, the composition operator must merge two sorted lists to produce the result of the sorting function. This problem again leads to a divide-and-conquer algorithm.

Since a list with at most one element is always sorted, *directly_solve* becomes the identity function. In the end, selecting *listsplit* has lead us to developing a mergesort algorithm.

$$\begin{aligned} mergesort(x) \equiv\ & \textbf{if } length(x) < 2 \\ & \textbf{then } x \\ & \textbf{else } (merge \circ (mergesort \times mergesort) \circ listsplit)(x) \end{aligned}$$

The procedure shown in Fig. 1 is an example of the problem solving knowledge we want to represent as strategies. It gives guidance on what to do in which order, but nevertheless cannot be carried out completely automatically.

2.2 The Structure of Strategies

There is a subtle interference between decomposition, composition and direct solution. The specifications for *compose* and *directly_solve* can be set up only after the code for *decompose* is known. If we choose a different decomposition, not only the algorithms but also the *specifications* for composition and direct solution look different. Assume, for example, we decided to implement decomposition by cutting off the first element of the list. Then we would get only one recursive call and the specification for *compose* would be to produce a sorted list out of a sorted list and a single element. We would end up with an insertion sort algorithm.

Figure 1 suggests a fixed order to find the composition and the primitive operators. This is an over-specification because these steps are independent. Descriptions of strategies serving as a basis for an implementation must not only express dependencies between but also *independence* of subproblems in order to pose as few restrictions as possible on the users and show possibilities to parallelize work.

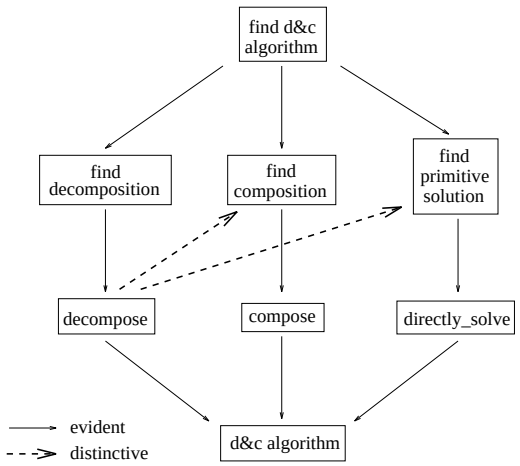

Fig. 2. Example dependency graph

In general, the subproblems of a strategy are not independent of each other and of the solutions to other subproblems. The dependency graph for the divide-and-conquer strategy is shown in Fig. 2. Due to the tight relation between the control predicate and the decomposition algorithm we only get three subproblems. The solution to the decomposition algorithm contains the control predicate (step 2 of Fig. 1).[4]

The arrows denote dependencies. Plain arrows are *evident* dependencies. They reflect our intuition of problem solving: the subproblems depend on the original problem, and their solutions depend on the corresponding problems. The final solution depends on the solutions of the subproblems. The bold dashed arrows are more interesting. They are called *distinctive* dependencies and are characteristic of the divide-and-conquer strategy of Fig. 1. These dependencies induce a partial ordering on the subproblems: it restricts the order in which the various subproblems can be set up and solved. The dependency graph of a valid strategy must not contain cycles. Moreover, problems must not depend on solutions, nothing may depend on the final solution, and the initial problem must not depend on anything.

For a strategy to work, we need to know not only its dependency relation but also exactly how the subproblems are constructed, how the final solution is assembled from the solutions to the subproblems, and how to check if this solution is acceptable. A strategy is described by the following items:

- the number of subproblems it produces,
- the dependency relation on them and their solutions,

[4] For program synthesis, solutions do not consist of just program code. They contain additional information about the behavior of the program, see Sect. 7.

- for each subproblem, a procedure how to set it up using the information in the initial problem and the subproblems and solutions it depends on,
- a procedure describing how to assemble the final solution,
- a test of acceptability for the assembled solution, and
- optionally a procedure providing an explanation *why* a particular solution is acceptable.

The last item is not strictly necessary for a strategy to work. Still, one might be interested in a more detailed documentation of why a particular solution "works" for a given problem. An explanation may be a formal proof or an informal description, depending on the required degree of mathematical rigor.

The above description of what a strategy consists of is parameterized by the notions of problem, solution, and acceptability. Problems and solutions provide a common interface between strategies. It is therefore possible to design a system architecture for strategy-based problem solving that is generic in the exact definition of problems and solutions. This system architecture is described in the following sections. The notion of strategy sketched here can be precisely defined in terms of relational calculus and partial orders [10].

3 The System Architecture

To begin with, we give an overview of the central components of the system architecture and sketch by way of an example how strategy-based problem solving proceeds. Sections 4 through 6 provide a more detailed description of the architecture's components.

3.1 Overview of the Architecture

Figure 3 gives a general view of the architecture. Two global data structures represent the state of development: the *development tree* and the *control tree*. The development tree represents the entire development that has taken place so far. Nodes contain problems, information about the strategies applied to them, and solutions to the problems as far as they have been found. Links between siblings represent dependencies on other problems or solutions. The data in the control tree is concerned only with the future development. Its nodes represent open tasks. They point to nodes in the development tree that do not yet contain solutions. The degrees of freedom to choose the next problem to work on are also represented in the control tree.

The third major component of the architecture is the strategy base. It represents knowledge for strategy-based problem solving by modules implementing strategies. Each module consists of a set of functions that realize the tasks comprising a single strategy.

A development roughly proceeds as follows: the initial problem is the input to the system. It becomes the root node of the development tree. The root of the control tree is set up to point to this problem. Then a loop of strategy applications is entered until a solution for the initial problem has been constructed.

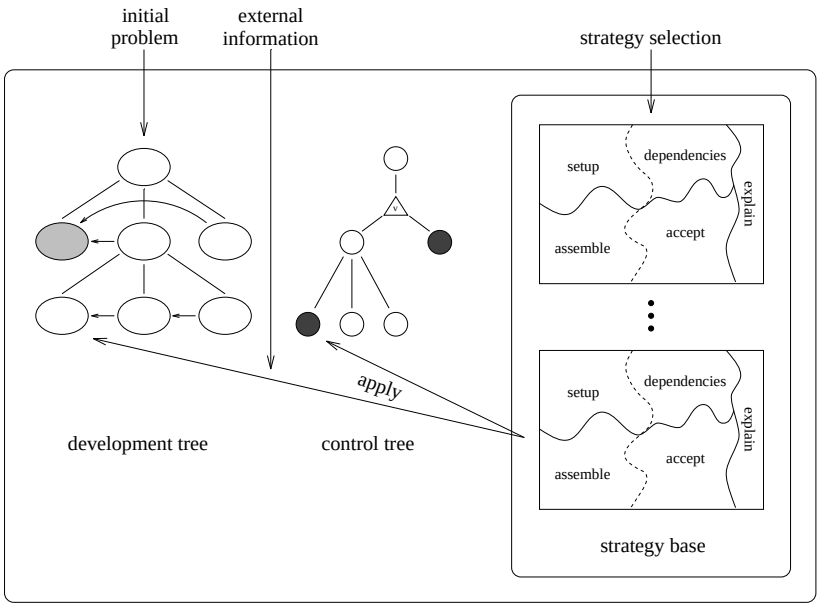

Fig. 3. General view of the system architecture

To apply a strategy, first the problem to be reduced is selected from the leaves of the control tree. Second, a strategy is selected from the strategy base. Strategy selection will usually be interactive but implementations of heuristics to choose a strategy or to suggest a set of applicable ones are also conceivable. Applying the strategy to the problem means to extend the development tree with nodes for the new subproblems, install the functions of the strategy in these nodes, and set up dependency links between them. The control tree is also extended according to the dependencies between the produced subproblems. Application of a strategy may need more information than is provided by the problem it is applied to, e.g. information needed to prove termination. Like strategy selection, providing external information usually encompasses user interaction or heuristics.

If a strategy immediately produces a solution and does not generate any subproblems, or if solutions to all subproblems of a node in the development tree have been found, the functions to assemble and accept a solution are called, and, if successful, the solution is recorded in the respective node of the development tree. When a solution is produced the control tree shrinks because it only contains references to unsolved problems. The process terminates when the control tree vanishes, because then the solution for the initial problem has been found. The result of the process is a development tree where all nodes contain acceptable solutions.

3.2 Example: Mergesort Revisited

To illustrate how the notion of strategy introduced in Sect. 2 is supported by the architecture of Fig. 3, we reconsider the example of Sect. 2.1 and sketch how an instance of the architecture for program synthesis works when developing the mergesort algorithm. Figure 4 shows a snapshot of the development.

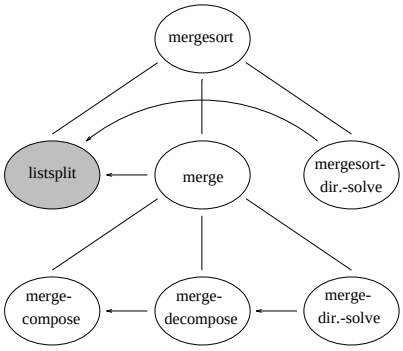

Fig. 4. Development tree for a mergesort algorithm

The initial problem is to sort a list. We decide to apply the divide-and-conquer strategy to it. Upon invocation of the strategy, we already know the dependencies between the children of the initial problem: the exact forms of both the composition and direct solution problems depend on the algorithm constructed to decompose the list. The dependencies are recorded in the development tree (pointed arrows).

The only problem that can be tackled now is the decomposition. Having found *listsplit* as a solution to the decomposition problem (shaded node), we are free to choose which one of the two remaining problems to reduce first: they are independent of each other. In Fig. 4, the composition problem is reduced first.

To develop the *merge* algorithm, an alternative divide-and-conquer strategy to the one described in Sect. 2 is applied. It constructs the algorithm "backward": first the composition part, then the decomposition, and finally the primitive solution part.[5] Applying this strategy to the problem for *merge* produces the development tree shown in Fig. 4.

Suppose now, the *merge* and *mergesort-directly-solve* algorithms have been developed. Then the final step of the application of the divide-and-conquer strategy to the initial sorting problem is to assemble the solutions *listsplit*, *merge* and *merge-directly-solve*.

[5] Extending the result list by one element is a simple composition operation for *merge*, while deciding which element of the two input lists to put next into the result is more complex. Thus a function for *decompose* is unlikely to be found in a library and the strategy of Fig. 1 is not applicable.

4 Strategy Implementation

Implementations of strategies should be independent of each other with a uniform interface between them. Thus, the implementation of a strategy is a module with a clearly defined interface to other strategies and the rest of the system. In the following, we call an implementation of a strategy a *strategy module*. The signature shown in Fig. 5 corresponds to the components of a strategy as described in Sect. 2.2.

$$
\begin{aligned}
\mathit{subpr} &: \mathit{Nat} \\
\mathit{dependency} &: \mathbf{array}\,[1 \ldots \mathit{subpr}, 1 \ldots \mathit{subpr}]\,\mathbf{of}\ \mathit{Bool} \\
\mathit{setup} &: \mathbf{array}\,[1 \ldots \mathit{subpr}]\,\mathbf{of}\ (\mathcal{P} \times \mathbf{list}(\mathcal{P} \times \mathcal{S}) \to \mathcal{P}) \\
\mathit{assemble} &: (\mathcal{P} \times \mathbf{array}\,[1 \ldots \mathit{subpr}]\,\mathbf{of}\ \mathcal{S}) \to \mathcal{S} \\
\mathit{accept} &: (\mathbf{array}\,[0 \ldots \mathit{subpr}]\,\mathbf{of}\ (\mathcal{P} \times \mathcal{S})) \to \mathit{Bool} \\
\mathit{explain} &: (\mathbf{array}\,[0 \ldots \mathit{subpr}]\,\mathbf{of}\ (\mathcal{P} \times \mathcal{S})) \to \mathcal{E}
\end{aligned}
$$

Fig. 5. Interface of a strategy module

The natural number *subpr* is the number of subproblems generated by a strategy. The Boolean matrix *dependency* represents dependencies between the children nodes 1 through *subpr*. While we can describe *dependency* as an array, the remaining elements of the module are proper functions or procedures because they represent the algorithmic content of the strategy. For each subproblem, we need to know how to set it up. Thus *setup* is an array of functions. The function *setup*[i] produces the i^{th} subproblem from the initial problem and a list of problems and solutions. This list contains the sibling problems and their solutions on which problem i depends.

The *assemble* function computes the final solution from the initial problem and the solutions to all subproblems. The *accept* and *explain* functions are concerned with the final solution. This solution is checked by *accept* for acceptability with respect to the initial problem. Optionally, *explain* may provide an explanation of type $\mathcal{E}$ to further document why the solution is acceptable.

5 The Structure of Development and Control Trees

We describe the internal structure of the development and the control tree, and their interaction with the strategy base.

5.1 Development Tree

Two strategies are involved in processing one node of the development tree: a *creating* and a *reducing* strategy. Figure 6 shows the internal structure of a node of the development tree and its relation to the creating and reducing strategies. The flow of information is indicated by pointed arcs. A node contains a problem and its solution, and references to its children and to siblings it depends

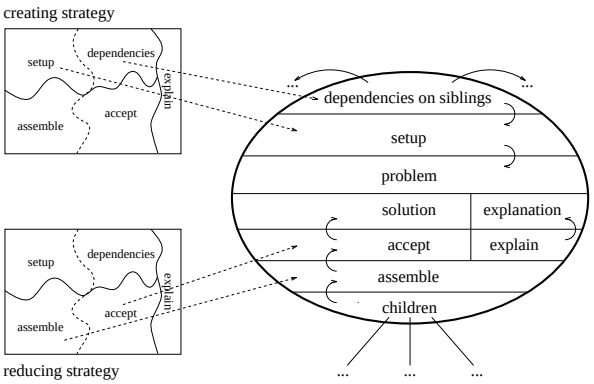

Fig. 6. Structure of a node in the development tree

on. Furthermore, it contains the functions needed to set up the problem and determine its solution. These functions stem from the strategy modules involved.

The development tree as a data structure contains all information about the process, the open problems and the result of the current development. Thus it is the basis to browse and provide views of developments, switch between developments, and analyze them for replay and re-use.

5.2 Control Tree

The purpose of the control tree is to keep track of unsolved problems and their dependencies. It provides a basis to choose the next problem to reduce. Figure 7 shows how the nodes of the control tree point to unsolved nodes in the development tree. There are two kinds of branchings in the control tree that stem from the dependencies between the development nodes. They indicate whether siblings have to be solved in a fixed left-to-right order or if they may be solved in an arbitrary order. The "normal" branching in the left subtree of the control tree in Fig. 7 represents a fixed order in which the problems have to be solved. On the other hand, the triangle v in the upper branching represents an arbitrary order for the two children of the root. The leaves of the control tree point to unreduced problems. The shaded leaves may be tackled in the next step.

As far as possible, selection of the next problem should be left to the developer. When selecting a strategy to reduce a particular problem, it is usually not obvious if the strategy will succeed in producing a solution. Therefore developers might try to tackle the "hardest" subproblem first and reduce it until they can decide if a solution is possible. Then they might concentrate on the next "hard" problem in some other branch of the development. In this way, the architecture makes it possible to focus development on the critical tasks first.

All information the control tree represents is contained in the development tree. Still, for efficiency reasons, it is useful to maintain control information

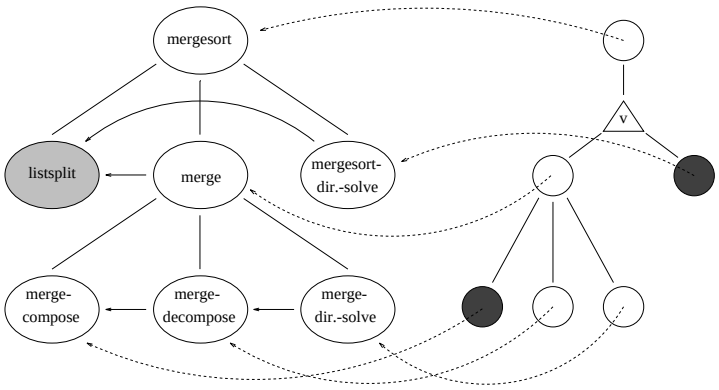

Fig. 7. The control tree tracks unresolved problems

explicitly. The development tree grows with each strategy application while the control tree shrinks whenever a solution is found. Without an explicit control tree, the set of reducible nodes would have to be re-computed for each strategy application.

6 Data Flow

The data flow diagram in Fig. 8 describes how the global data structures are manipulated. The main control flow is a loop of strategy applications. Upon each entrance of the loop body, a backtrack point is set. The strategy application cycle consists of selecting a problem and a strategy, reducing that problem by the strategy, and assembling solutions.

Node Selection. The set of reducible leaves can be determined by considering the control tree's two kinds of branchings. The reducible leaves of a tree with normal root branching are the reducible leaves of the leftmost subtree. For a triangle branching, they are the *union* of the sets of reducible leaves of all subtrees. Users may choose from the set of reducible leaves. The chosen node becomes the *current node*. It is possible to enhance flexibility of node selection and try to set up problems that depend on incomplete solutions. Such problems are not in the set of reducible leaves determined from the control tree.

Strategy Selection. Like selecting a node, choosing a strategy is typically a user decision which may be assisted by heuristics. For example, some strategies are applicable only to problems with certain properties. One heuristic might be to search the strategy base for strategies particularly suited for the current problem.

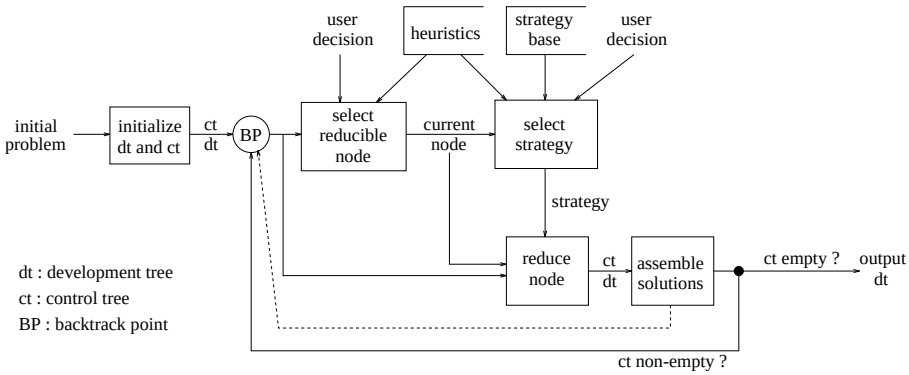

Fig. 8. Data flow diagram for the architecture

Solution Assembly. If the selected strategy creates no subproblems, the solution to the current node can be immediately determined: *assemble* is called for the current node, and the *accept* test is applied. If the test fails, the most recent cycle of problem selection and strategy application is undone. The system backtracks to the state of development before selection of the current node, symbolized by the dashed arrow in Fig. 8.

If the solution is acceptable, *explain* fills in the *explanation* field of the current node (cf. Fig. 6). The current node of the control tree is deleted. If the parent node of the deleted one has no other children, the process of solution assembly is recursively applied to that node.

Even if a solution is acceptable for the selected strategy, it may be inadequate as part of the solution to a problem higher up in the development tree. Any failure of *accept* functions during recursive solution assembly therefore causes a backtrack, where the most recent strategy application is undone.

Backtracking may be initiated by the users as well, e.g. if they decide that a strategy application leads nowhere because the generated subproblems cannot be solved. User-driven backtracking is possible during both node and strategy selection.

The loop of strategy applications terminates when the control tree is empty, yielding a development tree in which all nodes have successfully been solved. Its root contains the solution to the initial problem.

7 IOSS - A Prototypical Implementation

IOSS is an instantiation of the described architecture. It supports synthesis of provably correct imperative programs. The basis for the implementation of IOSS is the *Karlsruhe Interactive Verifier* (KIV), a shell for the implementation of

proof methods for imperative programs [11]. It provides a functional *Proof Programming Language* (PPL) with higher-order features and a backtrack mechanism. Strategies are implemented as collections of PPL-functions in separate modules. New strategies can be incorporated in a routine way. Currently a template file for new strategies supports this process; for the future, we envision tool support relieving the implementor of anything but the peculiarities of the newly implemented strategy. The graphical user interface of IOSS (see Fig. 9) is written in `tcl/tk` [17] and integrates the graph visualization system `daVinci` [7] to display the development tree (for details see [12]).

In this section, we first establish the notions of problem, solution, acceptability, and explanation that are used for IOSS. We then give an overview over its strategy base. Finally, we sketch an example development.

7.1 Problems, Solutions, and Explanations

In IOSS, *problems* are specifications of programs, expressed as pre- and postconditions that are formulas of first-order predicate logic. To aid focusing on the relevant parts of the task, the postcondition is divided into two parts, *invariant* and *goal*. In addition to these we have to specify which variables may be changed by the program (result variables), which ones may only be read (input variables), and which variables must not occur in the program (state variables). The state variables are used to store the value of variables before execution of the program for reference of this value in its postcondition.

Solutions are programs in an imperative Pascal-like language. Additional components are additional pre- and postconditions, respectively. If the additional precondition is not equivalent to true, the developed program can only be guaranteed to work if both the originally specified and the additional precondition hold. The additional postcondition gives information about the behavior of the program, i.e. it says *how* the goal is achieved by the program. If, e.g., the specification requires the value of variable x to be increased, the additional postcondition might contain the equation $x = x' + 4711$ which means that x is increased by 4711.

A solution is *acceptable* if and only if the program is totally correct with respect to both the original and the additional the pre- and postconditions, does not contain state variables, and does not change input variables. Thus, checking for acceptability of a solution amounts to proving verification conditions on the constructed program.

Explanations for solutions are provided as formal proofs in dynamic logic [8]. This is a logic designed to prove properties of imperative programs. Proofs are represented as tree structures that can be inspected at any time during development.

7.2 The Strategy Base

A number of interactive, semi-automatic and fully automatic strategies have been implemented. In the current version, they are oriented toward programming

language constructs.

Three strategies solve a problem directly: one for developing the empty program `skip` (*skip* strategy), two for developing assignments (*manual assignment* and *automatic assignment* strategy).

Two strategies can be applied to modify a problem: the *state variable* strategy introduces a new state variable for some result variable. The *strengthening* strategy is needed to use domain-specific knowledge in the problem solving process. The idea is to replace the goal of the problem by a stronger one, i.e. a formula which entails the old goal in the model under consideration.

Three strategies are available for developing compound statements: one corresponds to the rule for compound statements in the Hoare calculus (*intermediate assertion* strategy), the two others are based on Dershowitz' approach for conjunctive goals [6]. The *disjoint goal* strategy can be applied if the goal can be divided into two independent subgoals. The *protection* strategy can be applied when the subgoals are not independent as required for the disjoint goal strategy. In this case, the goal for the first statement must be an invariant for the second one.

Two strategies can be used to develop conditionals: the *conditional* strategy reflects the rule for conditionals of the Hoare calculus, the *disjunctive conditional* applies if the goal is of disjunctive form.

The *loop* strategy is currently available to develop a loop. Since it does not consider the initialization of the loop and the development of the invariant, it is usually applied in combination with the *strengthening* and *protection* strategies.

In the near future, higher level strategies will be built in. For example, an additional loop strategy that performs the development of the invariant, the initialization, and the loop body in a single reduction step, according to Gries' method [9]. Also, strategies for the development of divide-and-conquer algorithms or reusable procedures have been defined.

A complete description of these strategies can be found in [10].

7.3 IOSS in Practice

Figure 9 shows a snapshot of the synthesis of a heapsort algorithm. On the left-hand side of the window the development tree is displayed. Different colors and shades of the nodes visualize different states. The user can re-scale the tree, hide subgraphs, or view nodes. A separate window pops up for each node; several nodes can be inspected at the same time. On the right-hand side the user sees the current problem. To apply a strategy one chooses one from the menu of strategies shown in the center.

The task is to sort an array `a` of integers. The concept of the heapsort algorithm is to first build a heap[6], and then level down the heap putting the

[6] A heap is a binary tree of numbers where each node is greater than or equal to both of its successors. Such a tree can be stored in an array: the successors of node i are stored under the addresses $2i+1$ and $2i+2$.

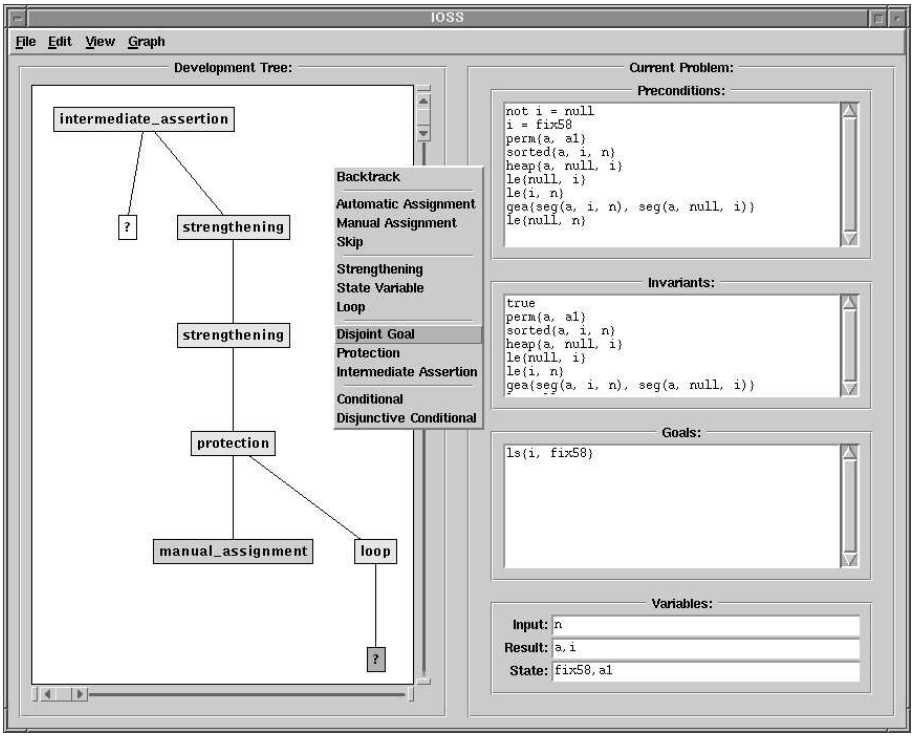

Fig. 9. The IOSS interface

root (maximum) element at the end of the array and restoring the heap for the remaining unsorted segment of the array.

Hence, we first apply a compound strategy to the original problem[7] (shown in Fig. 10(a)). In the first part of the compound, the heap is built up, in the second part, the sorted array is established as mentioned above. We choose the *intermediate assertion* strategy with the intermediate assertion[8]

`heap{a, null, n} and perm{a, a1}`

Since we can expect to get hints how to build up a heap from the transformation of a heap into a sorted array we decide to develop the second part of the algorithm first (see Fig. 9). IOSS supports this kind of approach: the *intermediate assertion* strategy allows us to choose the subproblem to tackle first. With left-to-right processing of the subproblems enforced, the system would have hindered instead of helped.

[7] IOSS uses a prefix-ascii notation for functions and predicates. Variables, constants, predicates, and functions as well as non-logical axioms about them are defined in a theory file read in by IOSS.

[8] $\mathtt{a}[0..n-1]$ is a heap and a permutation of the original array, denoted by the state variable `a1`.

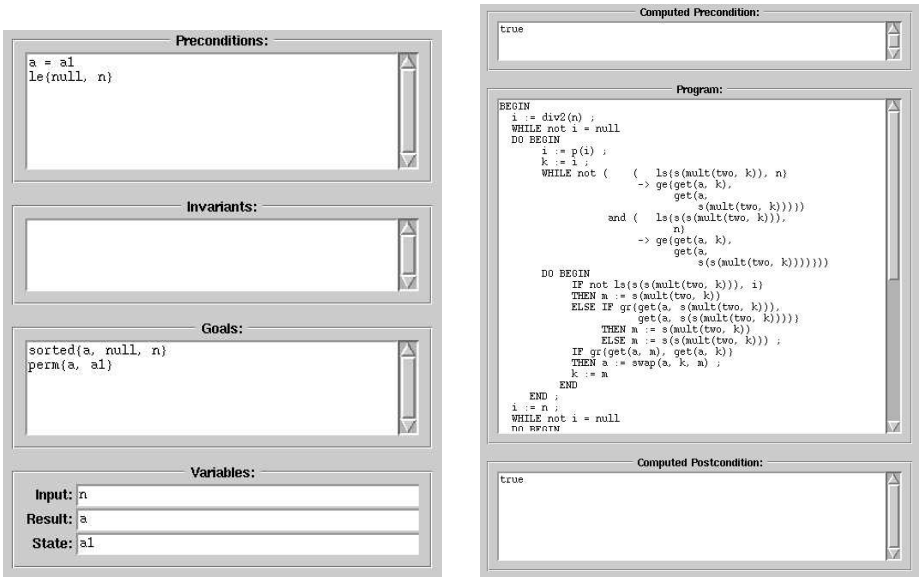

Fig. 10. The initial problem (a) and its solution (b)

Heapsort is a non-trivial algorithm. We needed a total of 54 strategy applications to synthesize its two doubly-nested loops. The result is shown in Fig. 10(b). To find the right loop condition and to avoid "off-by-one" errors in indexing the array is not easy. The possibility to examine the development tree during the development proved most valuable. First, we were always able to put the current problem into context and decide what to do next. Second, IOSS proved our first paper-made concept of the algorithm to be incorrect. Reviewing the erroneous development tree and analyzing the false verification conditions generated by IOSS gave the clue to correct the development.

With higher-level strategies than the ones currently available, this work could be further reduced. Since the program section that builds up the heap is almost identical in both parts of the algorithm, a strategy to encapsulate code in procedures would reduce the required strategy applications by half. Such higher-level strategies are already designed. It is just a matter of time before we incorporate them into the system.

8 Related Work

Our work relates to knowledge representation techniques and process modeling in classical software engineering, program synthesis and automated theorem proving.

Knowledge-Based Software Engineering (KBSE). A prominent example of KBSE which is close to our aims is the Programmer's Apprentice project [20]. There, programming knowledge is represented by *clichés*. These are prototypical examples of the artifacts in question, e.g. programs, requirements documents or designs. The Apprentice approach assumes that a library of prototypical examples provides better user support than the representation of general-purpose knowledge. We find it difficult to set up a sufficiently complete cliché library that does not need to be extended for each new problem.

Representation of Design and Process Knowledge. Wile [26] presents the development language Paddle. Paddle programs express procedures to transform a specification into a program. The procedural representation of process knowledge has the disadvantage that it enforces a strict depth-first left-to-right processing. This restriction also applies to more recent procedural approaches to represent software development processes [16, 21].

Potts [19] aims at capturing not only strategic but also heuristic aspects of design methods. He uses *Issue-based Information Systems* (IBIS) [4] as a representation formalism. IBIS representing heuristics tend to be specialized for a particular application domain. Our approach, in contrast, aims at representing general, domain independent problem solving knowledge.

Souquières [24, 25] has developed an approach to specification acquisition whose underlying concepts have much in common with the ones presented here. Specifications acquisition is performed by solving *tasks*. The agenda of tasks is called a *workplan* and resembles our development tree. A workplan is an AND/OR-tree where OR-nodes represent alternative developments. Tasks can be reduced by *development operators* similar to strategies. Development operators, however, do not guarantee semantic properties of the product. Therefore, incomplete reductions and a variable number of subtasks for the same operator can be admitted.

In the German project KORSO [2], the product of a development is described by a *development graph* [14]. Its nodes are specification or program modules whose static composition and refinement relations are expressed by two kinds of vertices. There is no explicit distinction between "problem nodes" whose contents are not completely known and "solution nodes". In contrast to the development tree the KORSO development graph does not reflect single development steps. A branching in our development tree maps to a subgraph in their development graph where process information like dependencies between subproblems cannot be represented.

Program Synthesis. IOSS serves to integrate a variety of methods which can be expressed in its basic formalism. The synthesis systems CIP [3], PROSPECTRA [13] and LOPS [1], in contrast, are all designed to support specific methods. It is not intended to integrate these methods with other ones.

The approach underlying the system KIDS [22, 23] is to fill in algorithm schemas by constructive proof of properties of the schematic parts. This is achieved by highly specialized code (*design tactics*) for each schema. Section

2.1 shows how design tactics can be expressed as strategies. In KIDS however, there is no general concept of design tactics or how to incorporate a new one into the system. Information about the development process is maintained implicitly. Working with KIDS, it is hard to keep track of "where" one is in a development. There is a logging and replay facility, but this provides no possibility to browse the state of development. Since design tactics are linearly programmed, there is no way to change the order of independent design steps or "interleave" tactics applications.

Tactical Theorem Proving. Tactical theorem proving has first been employed in Edinburgh LCF [15]. The idea is to conduct interactive, goal-directed proofs by backward chaining from a goal to sufficient subgoals. *Tactics* are programs that implement "backward" application of logical rules. The functional programming language ML evolved as the tactic programming language of LCF. Tactical theorem proving is also used in the generic interactive theorem prover Isabelle [18] and in KIV [11], the theorem proving shell underlying IOSS.

The goal-directed, top-down approach to problem solving is common to tactics and strategies. Nevertheless, there are some important differences. First, a tactic is one monolithic piece of code. All subgoals are set up at its invocation. Dependencies between subgoals can only be expressed by the use of *metavariables*. These allow one to leave "holes" in a subgoal that are "filled" during proof of another subgoal by unification on metavariables. Dependencies not schematically expressible by metavariables are not possible with tactics. Since tactics only perform goal reduction, there is no equivalent to the *assemble* and *accept* functions of strategies. They are not necessary for the tactic approach because problems and solutions are identical except for instantiation of metavariables. In contrast, problems and solutions of strategies may be expressed in different languages, and the composition of solutions by *assemble* may not be expressible schematically.

Theorem proving systems like Isabelle usually do not maintain a data structure equivalent to the development tree. Isabelle only maintains a stack of proof states containing the results of tactic applications in chronological order. They are discarded upon completion of the proof. No information is given about the tactics that produced a proof state or the dependencies between proof states. It is the users' responsibility to record their proof steps textually outside of the system.

9 Discussion

Most of the tools supporting formal methods today deal with single documents and not with the process aspect of a development. They are used to check static semantics of the documents or to discard proof obligations obtained without tool support. The few tools we know of that support the process aspect, e.g. KIDS, enforce one fixed way of procedure on their users and do not provide an overview of the state of development (see Sect. 8).

Existing tools often are monolithic systems and hardly modifiable except by their developers. This prevents incorporation of new problem solving knowledge by local modifications. It also reduces confidence in the tools, because is it not clear which pieces of code are responsible to guarantee semantic properties of the products.

The concept of *strategy* and the generic system architecture based on *uniform interfaces* of strategy modules and centered around the data structure of a *development tree* contribute to overcome these deficiencies. The product of a development is a development tree with acceptable solutions in all nodes. It contains explanations for all strategy applications and documents design decisions and their justifications. This improves comprehensibility of the product and may be used as a basis to conduct inspections by certification authorities. Our work relates to the requirements stated in Sect. 1 as follows.

Guarantee Semantic Properties. The function *accept* is the only component of the interface of a strategy module that is concerned with semantic properties. Only this function determines if a candidate solution is acceptable for the given problem. How the other components – and other strategies – contribute to the evolution of a candidate solution has no influence on this process. This is important for the design of strategies: they need not produce solutions that are acceptable in every context because a strategy using their output to compose a solution will check for acceptability of the composed solution. There is a single point in a strategy implementation that is responsible for the semantic properties of the produced solution. This enhances confidence in the development tool because only the *accept* functions have to be verified to ensure that the tool truly guarantees acceptability of the produced solutions.

Balance User Guidance and Flexibility. Methods are uniformly represented as sets of strategies. Their common interface to the system kernel makes method combination possible: strategies of different methods can be interleaved to solve a problem, e.g. the Gries' method can be used to solve the subproblems created by Smith's divide-and-conquer-strategy. To incorporate a new method into the system, the strategy base only has to be extended by the new strategies. This involves only local changes that do not affect existing components.

More work is necessary if the notions of problem, solution or acceptability have to be changed. One example is to extend the problems of IOSS by an additional invariant that must not be destroyed even in intermediate states of the synthesized program. This kind of invariant is useful for enforcing safety requirements. In this case, all strategies have to be revised, but the clear modularization still helps in identifying the code that has to be changed.

The development tree allows for multi-developer environments and explorative procedures. Independent leaves can safely be worked on in parallel while the global context is still accessible by all developers.

Provide Overview of Development. By maintaining the open subproblems and their dependencies in the development tree we get not only an overview of the state of the development but the entire development is mirrored in this data structure. It can be browsed to find out interrelations between subproblems and

thus to get insight into the role a certain component plays. This possibility is particularly useful where creative design decisions have to be taken. They do not only depend on the formal requirements as stated in the problem description, but must consider the net effect a decision may have. Browsing is all the more essential when using formal methods because of the increased level of detail in formal documents. In case of a dead end in a development, it supports analysis of the steps that led to the error. The behavior of most theorem provers that just say "no" without further explanation why a proof attempt failed is not acceptable in software development.

Implications to Future Research and Applications

The present implementation of IOSS is certainly not powerful enough to approach problems of realistic size. We are working on the implementation of more powerful strategies. Concerning these, we see three implications to future research:

- Strategies are a means to describe methods in a way that makes them implementable. More experience has to be gained in expressing methods dealing with the different phases of the software life cycle as strategies. This will enhance understanding of the requirements of efficient and extensive tool support.
- Our approach concentrates on the problems that are specific to tool support for formal methods. Integration with CASE technology is highly desirable. We are currently working on integration of IOSS with tools for the specification language Z.
- Failure analysis and re-use are important to scale up the approach to realistic examples. The development tree is a basis to tackle both and can already be used for simple replay and re-use "as-is" techniques. More refined techniques remain to be developed.

Acknowledgment. We thank Sabine Dick, Bernd Krieg-Brückner, Balachander Krishnamurthy and Robert Raschke for comments on a draft of this paper.

References

1. W. Bibel and K. M. Hörnig. LOPS – a system based on a strategical approach to program synthesis. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 69–89. MacMillan, New York, 1984.
2. M. Broy and S. Jähnichen, editors. *KORSO: Methods, Languages, and Tools to Construct Correct Software.* LNCS. Springer Verlag, 1995. to appear.
3. CIP System Group. *The Munich Project CIP. Volume II: The Program Transformation System CIP-S.* LNCS 292. Springer-Verlag, 1987.
4. J. Conclin and M. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. *ACM Transactions on Office Informations Systems*, 6:303–331, October 1988.

5. D. Craigan, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1993.
6. N. Dershowitz. *The Evolution of Programs.* Birkhäuser, Boston, 1983.
7. M. Fröhlich and M. Werner. daVinci V1.3 User Manual. Technical report, Universität Bremen, 1994.
8. R. Goldblatt. *Axiomatising the Logic of Computer Programming.* LNCS 130. Springer-Verlag, 1982.
9. D. Gries. *The Science of Programming.* Springer-Verlag, 1981.
10. M. Heisel. A formal notion of strategy for software development. Technical Report 94–28, TU Berlin, 1994.
11. M. Heisel, W. Reif, and W. Stephan. Implementing verification strategies in the KIV system. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, LNCS 310, pages 131–140. Springer-Verlag, 1988.
12. M. Heisel, T. Santen, and D. Zimmermann. A generic system architecture of strategy-based software development. Technical Report 95-8, Technical University of Berlin, 1995.
13. B. Hoffmann and B. Krieg-Brückner, editors. *PROgram Development by SPECification and TRAnsformation, the PROSPECTRA Methodology, Language Family and System.* LNCS 680. Springer-Verlag, 1993.
14. B. Krieg-Brückner, W. Menzel, W. Reif, H. Ruess, T. Santen, D. Schwier, G. Schellhorn, K. Stenzel, and W. Stephan. *System Architecture Framework for KORSO.* In Broy and Jähnichen [2], 1995. to appear.
15. R. Milner. Logic for computable functions: description of a machine implementation. *SIGPLAN Notices*, 7:1–6, 1972.
16. L. Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.
17. J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.
18. L. C. Paulson. Isabelle: The next seven hundred theorem provers. In E. Losk and R. Overbeek, editors, *Ninth International Conference on Automated Deduction*, LNCS 310, pages 772–773. Springer Verlag, 1988.
19. C. Potts. A generic model for representing design methods. In *International Conference on Software Engineering*, pages 217–226. IEEE Computer Society Press, 1989.
20. C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, pages 10–25, November 1988.
21. T. Shepard, S. Sibbald, and C. Wortley. A visual software process language. *Communications of the ACM*, 35(4):37–44, April 1992.
22. D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27:43–96, 1985.
23. D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.
24. J. Souquières. *Aide au Développement de Specifications.* Thèse d'Etat, Université de Nancy I, 1993.
25. J. Souquières and N. Lévy. Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223, 1993.
26. D. S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983.

This article was processed using the LaTeX macro package with LLNCS style