A Generic System Architecture for Strategy-Based Software Development

Maritta Heisel

Thomas Santen[1]

Dominik Zimmermann

Fachgebiet Softwaretechnik

Bericht 95-8

[a]GMD FIRST, Rudower Chaussee 5, D-12489 Berlin

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 *strategies*, and a generic architecture showing how to implement tools for strategy-based development. A prototype system for program synthesis called IOSS is described in some detail. It demonstrates the practicality of the approach.

Acknowledgment. We would like to thank Balachander Krishnamurthy for comments on this work.

Contents

Chapter 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 [CGR93]. 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.

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 report, we proceed as follows: In Chapter 2, strategies are introduced. Chapter 3 presents a general overview of the architecture, followed by a description of its components: implementation of strategies (Chapter 4), internal data structures (Chapter 5),

and data and control flow (Chapter 6). We describe an implemented program synthesis system called IOSS (Integrated Open Synthesis System) as an instance of the system architecture in Chapter 7, followed by an example development in Chapter 8. We look at related work in Chapter 9. In Chapter 10, we discuss how our approach meets the above requirements and mention implications to future research. An appendix presents a complete program development with IOSS.

Chapter 2

Representing Software Engineering Knowledge by Strategies

Considering what makes up a method or process used for software development, it becomes apparent that there are two aspects to it: a set of *strategies*, and *heuristics* when to apply them. 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. They provide schemas what has to be done in which order to achieve a certain goal. 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 formalism, we call the heuristic part of their method.

While heuristics are hardly mechanizable, strategies can be implemented in interactive tools. Our system architecture is therefore designed to support problem solving by application of strategies in an interactive environment that does not hinder experts to use their heuristic knowledge.

The purpose of a strategy is to find a suitable solution to some software development problem. Strategies work by problem reduction:

For a given problem, a strategy determines a number of subproblems that are ideally easier to solve. 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. This process terminates if a problem is simple enough to be solved without further reduction.

Of course, this description is too general to be of much use. Indeed, it is even inadequate in its simplicity because it says nothing about interdependencies between the various subproblems and solutions. A "strategy" where a subproblem could only be formulated after the solution to the initial problem was known would certainly contradict intuition. An example from program synthesis serves us to motivate the more detailed description of strategies given in Section 2.2.

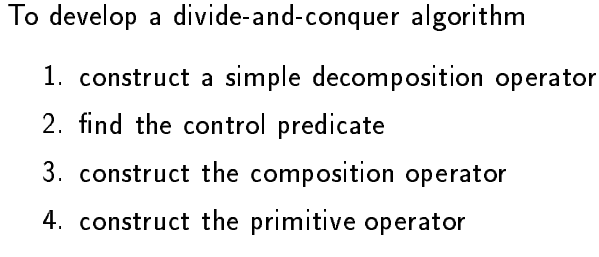


Figure 2.1: A Divide-And-Conquer Strategy

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

If we want to describe concrete strategies for a specific area of software development, we first have to fix the notions of problem, solution and acceptability. For 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 shall mean 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. For IOSS, the program synthesis system described in Chapter 7, the specification language is first-order predicate logic augmented with restrictions on the variables that may be changed by the program.

As an example, we consider an approach from literature to synthesize divide-and-conquer algorithms [Smi85]. Here, 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:

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

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 is left unchanged ($g = id$). Finally, the results yielded by f and g are composed to form the final result of the algorithm.

In [Smi85], several "strategic" ideas on how to develop divide-and-conquer algorithms by filling the gaps in the schematic algorithm are described. One is shown in Figure 2.1. The idea is to develop the decompose-recursion-compose part from front to back, and to find the algorithm for *decompose* by searching a library.

Consider the problem of sorting a list of integers.[1] The first thing to do is to find an algorithm in a library that reduces the length of the list. One possible solution is *listsplit*

[1] We are aware of the fact that to solve a problem like this our framework is not necessary because the

which splits the input list in two halves of approximately equal length. Once we have decided on the decomposition function, we can determine the test to stop the recursion: *listsplit* is applicable only if the input has at least length two. So $primitive(x)$ will become $length(x) < 2$.

Selecting *listsplit* also has consequences for the recursive case. Both halves of the input list have to be sorted, i.e. g becomes f in the schema. Hence, the composition operation has two sorted lists as input. It must merge them to produce the result of the sorting function. This problem again leads to a divide-and-conquer algorithm.

If *primitive* holds sorting is easy: a list with at most one element is always sorted, so *directly_solve* becomes the identity function. In the end, selecting *listsplit* has lead us to developing a mergesort algorithm.

$$mergesort(x) \equiv \textbf{if}\ length(x) < 2\ \textbf{then}\ x \quad \textbf{else}\ \ merge \circ (mergesort \times mergesort) \circ listsplit(x)\ \ \textbf{fi}$$

The procedure shown in Figure 2.1 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 in the example. 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 then 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, i.e. *decompose* returned the *car* and the *cdr* 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 (the sorted *cdr* of x) and a single integer (the *car* of x). We would end up with an insertion sort algorithm.

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 Figure 2.2. 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.[2]

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 problem. 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 for the divide-and-conquer strategy of Figure 2.1. 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 strategy must

algorithms for this purpose are well known. It should be noted, however, that with a similar approach [Smi90], a scheduling algorithm has been derived that is 2000 times faster than the ones known before.

[2]For program synthesis, solutions do not consist of just program code. They contain additional information about the behavior of the program. See Chapter 7 for more detail.

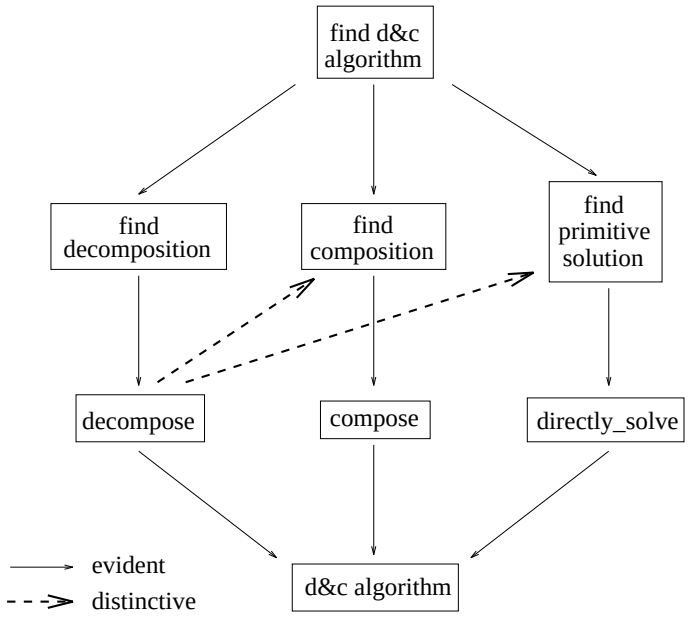


Figure 2.2: Example Dependency Graph

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,
- 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. In case of formal program synthesis, this may be a formal correctness proof. For specification acquisition this may be informal text; for planning problems, it may be some measure of goal distance.

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 architecture is described in the following chapters.

The notion of strategy sketched here can be precisely defined in terms of relational calculus and partial orders in [Hei94].

Chapter 3

The System Architecture

In this chapter, 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. The following chapters provide a more detailed description of the components.

3.1 Overview of the Architecture

Figure 3.1 gives a general view of the architecture. There are two global data structures

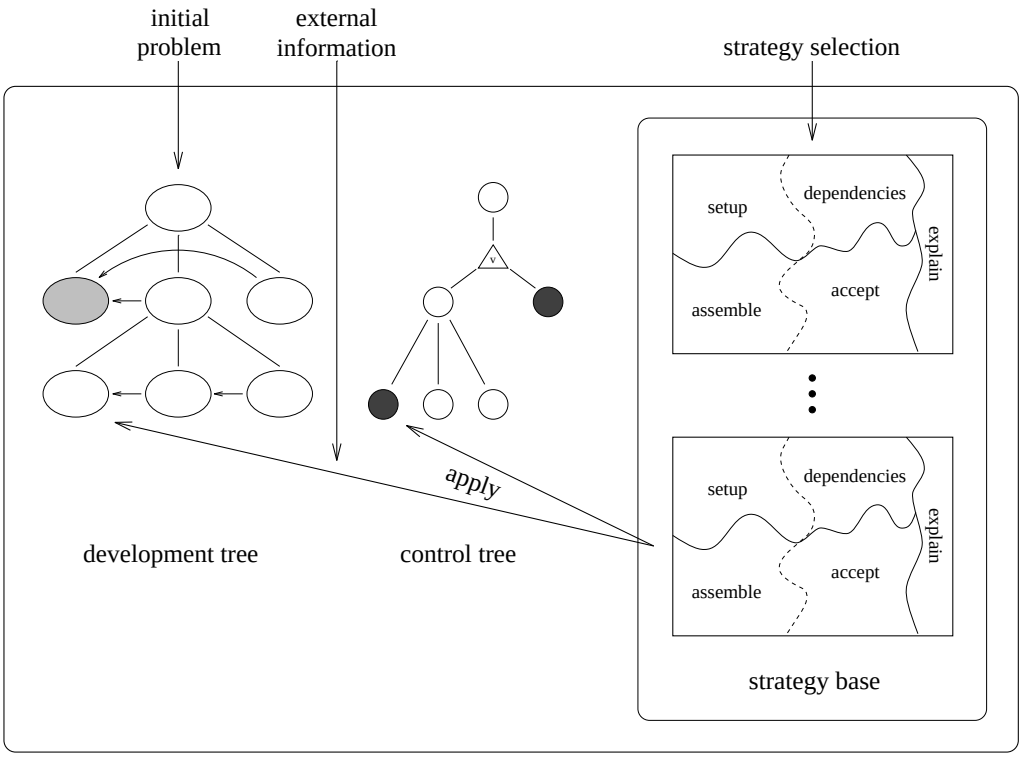


Figure 3.1: General view of the system architecture

that 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, i.e. they point to nodes in the development tree that do not yet contain a solution. Leaves in the control tree point to unreduced problems in the development tree. The degrees of freedom to chose the next problem to work on are also represented in the control tree.

The third major component of the architecture is the knowledge base that represents development knowledge for strategy based problem solving. It is a set of modules that implement strategies. Each module consists of a set of functions that implement the tasks comprising a single strategy. These are to set up subproblems, to assemble a solution, to check for correctness and acceptability of a solution, and to provide information about the strategy itself, in particular how many subproblems are generated and how they depend on other subproblems or solutions.

A development roughly proceeds as follows:

The initial problem is the input to the system. It becomes the root node of the development tree and 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 to the initial problem has been constructed.

To apply a strategy, first the problem to be reduced is selected from the leaves of the control tree. This may be done automatically by the system, it may propose a problem to reduce or to choose a problem may be left entirely to the user. Second, a strategy is selected from the strategy base. Strategy selection will usually be interactive but 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 produced 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 also need more information than is provided by the problem it is applied to. Like strategy selection, this external information may be supplied interactively.

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. As a consequence, the control tree shrinks, because it contains only references to unsolved problems.

The process terminates when the control tree vanishes, because then the solution to the initial problem has been found.

3.2 Example: Mergesort Revisited

To illustrate how the notion of strategy introduced in Chapter 2 is supported by the architecture of Figure 3.1, we reconsider the example of Section 2.1 and sketch how an instance of the architecture for program synthesis works when developing the mergesort algorithm. Figure 3.2 shows a snapshot of the development. The solution of our development is a mergesort

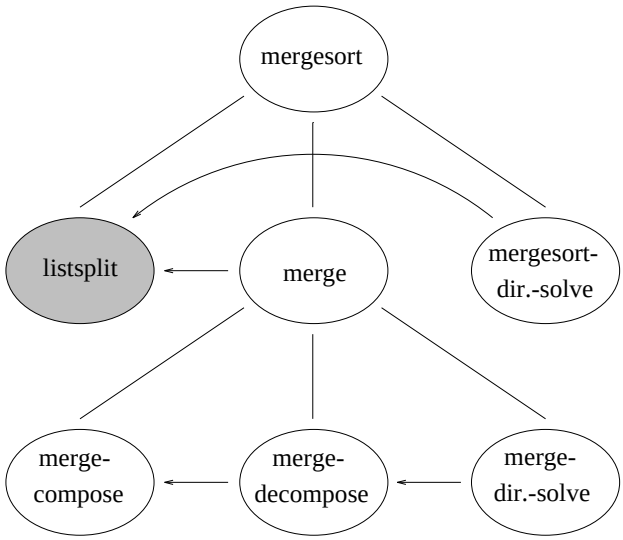


Figure 3.2: Development Tree for a Mergesort Algorithm

algorithm. To ease understanding the figure, the nodes in the development tree are labeled with names of the constructed procedures, i.e. the solutions, although these become known only as the development proceeds.

The initial problem is to sort a list. So the root *mergesort* of the development tree initially contains just this problem. We decide to apply the divide-and-conquer strategy to this problem. "Application" here first of all means to extend the development tree by nodes for the three subproblems that are produced by the strategy.

Upon invocation of the strategy, we already know the dependencies between the children of the initial problem: the exact form of both the composition and direct solution problems depend on the algorithm constructed to decompose the list. So we can record these dependencies in the development tree (pointed arrows).

The only problem that can be tackled now is the decomposition, because it is the only one that can exactly be set up. Assume we have found *listsplit* as a solution of the decomposition problem (shaded node) by application of some strategy. We are then free to choose which one of the two remaining problems to tackle, since they are independent of each other. In Figure 3.2, the composition problem is reduced first. How to set up this problem is described in the divide-and-conquer strategy. The information to which subproblem of which strategy a node in the development tree belongs has to be associated with that node of the development tree. Otherwise, the system would not know how to set up the problem in that node from the information in its sibling and parent nodes.

Since *listsplit* is known now, we can set up the problem for *merge*. Had we developed some other algorithm to decompose the input list, e.g. one that cuts off the first element of the list, then the problem for composition would look different.

An alternative divide-and-conquer strategy to the one described in Chapter 2 constructs the algorithm "backward": first the composition part, then the decomposition, and finally the primitive solution part [Smi85]. Applying this strategy to the problem for *merge* produces the development tree shown in Figure 3.2.

Suppose now, the *merge* and *merge-dir.-solve* algorithms are found. 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-dir.-solve*. This gives us the *mergesort* algorithm that is the solution to the sorting problem.

Chapter 4

Strategy Implementation

The information a strategy has to provide and the tasks it has to accomplish are described in Chapter 2. We now discuss how to implement a strategy within the generic system architecture of Figure 3.1.

The general view of the architecture has already made clear that implementations of strategies should be independent of each other with a uniform interface to the rest of the system. Thus, the implementation of a strategy is some kind of module with a clearly defined interface to other strategies and the rest of the system. As it is our aim to provide a design of a system for strategy-based problem solving, we will not be more specific with respect to the kind of modularization used in an actual implementation. Obviously, modularization mechanisms as Modula-2 modules, classes in object-oriented languages or ML functors are conceivable as grouping mechanisms for the elements of a strategy. In the following, we call an implementation of a strategy a *strategy module*.

The development of *mergesort* in Chapter 3.2 reveals two requirements on strategy modules within the system architecture. First, it shows that the tasks which make up the "application" of a strategy may be carried out with great distances of time between them: the time between deciding which strategy to reduce the initial problem with and assembling the solution to that problem encompasses the entire development. Therefore it is hardly possible to implement a strategy as one monolithic function. Second, the external structure of strategy modules depends on the implementation of the development tree and vice versa. Information about the strategy used to reduce a particular problem must be recorded in the corresponding node of the development tree. How this is accomplished is discussed in Section 5.1.

What is the external structure of a strategy module? The requirements on what makes up a strategy can be translated into the signature shown in Figure 4.1 where problems are denoted by $\mathcal{P}$ and solutions by $\mathcal{S}$. It provides one constant or function for each item of the list in Chapter 2. A node of the development tree will eventually contain both a problem and its solution. One branching in the development tree corresponds to one application of a strategy. To implement the dependency graph of a strategy amounts to represent the dependencies between the respective *nodes* of the development tree. Its structure is coarser than the one of the dependency graph in Figure 2.2, because the dependencies between nodes of the development tree do not distinguish between problems and solutions. We label the nodes of one branching by 0 through *subpr* which is the number of subproblems generated by the strategy. Node 0 is the root and contains the problem that is reduced by the strategy and the solution to that problem (which is produced by the strategy). Nodes 1 through *subpr*

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

Figure 4.1: External Structure of a Strategy Module

each contain a subproblem and its solution (and more which is described in Section 5.1).

The Boolean matrix *dependency* represents dependencies between the children nodes 1 through *subpr*.[1] If $\mathit{dependency}[n, m]$ is true then subproblem n depends on subproblem m or its solution.

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 $\mathit{setup}[i]$ produces the i-th subproblem from the initial problem and a list of problems and solutions. That list contains the sibling problems and their solutions on which problem i depends.

The *assemble* function computes the solution to the initial problem from the solution to all subproblems. It should be able to produce a partial solution from partial or unknown solutions to the subproblems. A partial solution may provide information on the structure of the final solution while some detail is still unknown. Such information may still suffice to proceed on other branches of the development. It can be propagated through the development tree and may contain enough information to set up a problem in some other branch of the development. Thus maximal flexibility where to continue the development can be provided.

The *accept* and *explain* functions are concerned with the final solution that is provided by *assemble* when all subproblems have been solved. This solution is checked by *accept* for acceptability with respect to the initial problem. In order to check acceptability, it may be necessary to refer to subproblems and their solutions. Hence, these are parameters of *accept*, together with the original problem and its solution. Optionally, *explain* may provide an explanation of type $\mathcal{E}$ to further document why the solution is acceptable.

[1] Non-evident dependencies to node 0 are forbidden (cf. Section 2.2).

Chapter 5

The Structure of Development and Control Trees

We here describe the internal structure of development and control trees, 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 5.1 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 obviously contains a problem and its solution and references to its

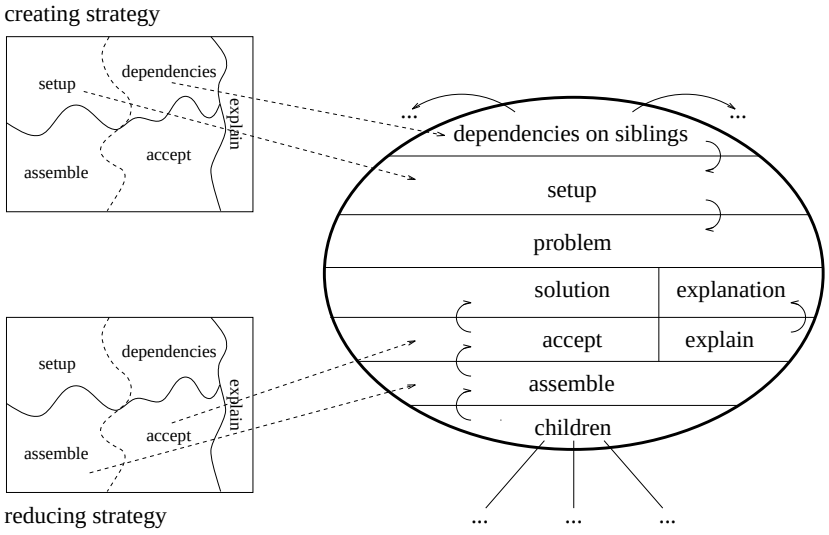


Figure 5.1: Structure of a Node in the Development Tree

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

involved (cf. Figure 4.1).

Let a particular node belong to the i-th subproblem of the creating strategy. Then this strategy provides the set-up function $setup[i]$. The dependency pointers are obtained from the projection $dependecy[i]$ of its dependency matrix.

The reducing strategy produces the children of a development node. It is therefore responsible to provide the functions that build the solution, check its acceptability and provide an explanation. The same strategy plays a dual role for the children nodes: for them it is the creating strategy.

The implementation of the development tree is obvious if the implementation language has higher-order features as functional languages do. Then it is possible to actually store the functions of the creating and reducing strategies in the nodes. For programming paradigms without higher-order features, the nodes of the development tree will probably contain references to the two involved strategies.

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 5.2 shows how the nodes of the control tree point to unsolved nodes in the development tree. There are two kinds

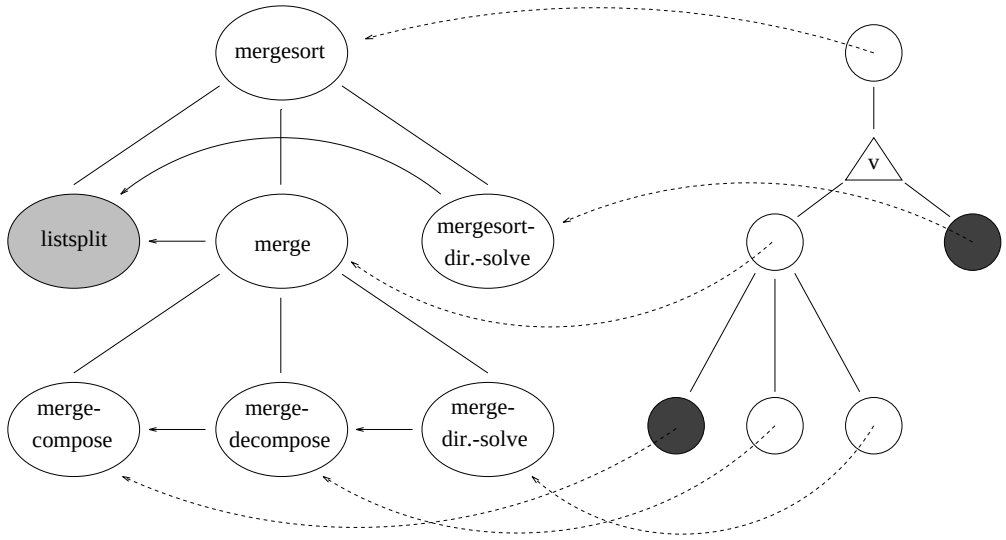


Figure 5.2: The Control Tree Tracks Unresolved Problems

of branchings in the control tree that stem from the dependencies between the development nodes. They tell if siblings have to be solved in fixed left-to-right order or if they may be solved in arbitrary order. The "normal" branching in the left subtree of the control tree in Figure 5.2 represents a fixed order in which the problems have to be solved. On the other hand, the triangle v in the upper branching represents a variable order of solution for the two children of the root. The leaves of the control tree point to unreduced problems. The shaded leaves can be set up completely and hence may be tackled in the next step. The control tree

thus is a representation of the global dependency relation on the nodes of the development tree that is induced by combination of the dependencies between siblings. Processing the control tree amounts to topologically sorting the problems with respect to the global dependency relation. This sorting process usually is not automated because the developers should be free to select the next problem to tackle.

Figure 5.2 once again depicts the situation in the development of *mergesort* after the composition problem *merge* has been reduced by application of the alternative divide-and-conquer strategy (cf. Section 3.2). This strategy requires a fixed order of solution for the subproblems: find a *compose* algorithm first, then a matching *decompose*, and finally a *directly_solve* algorithm. The strategy used to reduce the initial problem (cf. Section 2.1) does not prescribe an order in which *merge* and *mergesort-directly-solve* are developed: both problems only depend on *listsplit*. The two reducible candidates are shaded gray in the control tree.

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.

The control tree as a separate data structure is not strictly necessary. All information it represents is contained in the development tree. Still, for efficiency reasons, it is useful to maintain control information explicitly. The development tree grows monotonically with every strategy application while the control tree shrinks whenever a solution is found. The leaves of the control tree are exactly the unreduced problems and its branching types represent the global dependency relation. Without an explicit control tree, the set of reducible nodes would have to be re-computed for each strategy application.

Chapter 6

Data Flow

Chapters 4 and 5 have in detail described the global data structures for our system architecture. We now describe the engine that manipulates them with the data flow diagram shown in Figure 6.1. The global structure of flow is a loop of strategy applications. The initialization

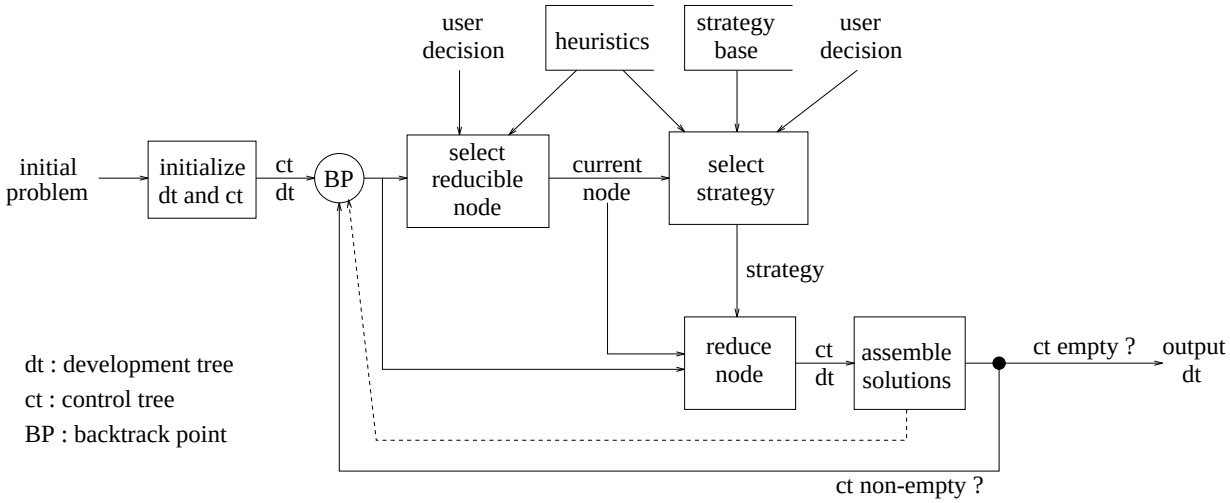


Figure 6.1: Data Flow in the Architecture

creates the root node of the development tree. The initial problem is the input to the system and entered into this node. The root node of the control tree points to that node.

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 leaves of the control tree point to unreduced development nodes. These nodes can be classified as *definitely reducible* or *possibly reducible*. Definitely reducible nodes are the ones that only depend on nodes with already known solutions. The set of definitely reducible nodes can be determined by considering the control tree's two kinds of branchings (cf. Section 5.2). The set of definitely reducible leaves of a tree with normal

root branching is the set of definitely reducible leaves of the leftmost subtree. For a triangle branching, it is the *union* of the sets of definitely reducible leaves of all subtrees.

A possibly reducible leaf of the development tree depends on other nodes that are not completely solved yet. Still, there is a chance that the problem of such a leaf can nevertheless be set up. Since, for the general architecture, no internal structure of problems and solutions is presupposed, the dependency relation between nodes is quite coarse. To set up a problem, it may suffice to know some structure of a solution it depends on without knowing the solution in full detail. To work on such a problem, developers may try and execute *setup* on a possibly reducible leaf. The called *setup* initiates calls to *assemble* – but not to *accept* – the solutions of the nodes the problem to set up depends on. If *setup* and all *assemble* functions successfully terminate the leaf is reducible and may be selected. If one of the functions fails[1] there is not enough information to construct the problem of the leaf, and it is no candidate for reduction.

Selection of a node is a user decision by nature. Users may choose from the set of definitely reducible leaves, or they may try a possibly reducible one and see if constructing the problem for this node succeeds. There may be heuristics to support this process. The chosen node becomes the *current node*.

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

Node Reduction. Node reduction extends the development tree and the control tree at the current node according to the selected strategy. The strategy module's *subtr* and *dependency* fields provide information how many children nodes must be created and which dependency pointers between them have to be established. The functions *setup*[*i*] are entered in the children nodes, and according to the role as reducing strategy for the current node, the *assemble*, *accept* and *explain* functions are entered in that node.

Solution Assembly. After node reduction, the extended development and control trees are searched for solutions to assemble. If the selected strategy creates no subproblems, i.e. $subpr = 0$, the solution to the current node can be determined immediately: *assemble* is called for the current node. Since the resulting solution must be definite for this node, 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 Figure 6.1.

If the solution is acceptable, *explain* fills in the *explanation* field of the current node (cf. Figure 5.1). The current node of the *control* tree is deleted, because the corresponding development node is now completely solved. 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 it is acceptable for the selected strategy (with $subpr = 0$), the solution it produces 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.

[1] We see the need for some *failure* mechanism in the implementation language.

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. Therefore, user driven backtracking is possible during both node and strategy selection.

The loop of strategy applications terminates when the control tree is empty. Then all nodes of the development tree have successfully been solved. Its root contains the solution to the initial problem which is the product of the development. The development tree as a whole documents the design process.

Chapter 7

IOSS – A Prototypical Implementation

Tool support for software development is an area of growing interest, as can be seen by the flourishing of computer-aided software engineering (CASE). CASE tools are fairly well understood, and there is a large number of them, see [Fug93]. If, however, one is interested in developing provably correct software, tool support is hardly available to date.

We consider the following properties as crucial for a system that supports the development of correct software. They are discussed in more detail in [HWW94].

- The system should be *interactive*, i.e. the user must be able to control the development process. This requirement holds at least as long as it is not possible to replace human creativity, e.g. in finding loop invariants or inductive arguments, by automatic processes.
- When the users control the development process, they will be confronted with system-generated intermediate states of a development. In this situation, they must be able to make sensible decisions. Therefore, all information that is important for the development process must be represented *explicitly* and not in encoded form.
- A system that imposes severe restrictions on the procedure to be followed in the development process will not be accepted. Hence, the system should be *flexible* and support different ways of developing a program.
- *Openness* is one more requirement to guarantee flexibility. It should be possible to add new development methods in a routine way, and the evolution of the system should take place gradually, without invalidation of former work.
- The system should visualize the development process in an appropriate way and provide an overview of the progress of development. It should be easy to use, making program synthesis feasible for users who did not invent the formalisms used, and enabling the programmers to concentrate on the task at hand.

The system presented in this report, IOSS (Integrated Open Synthesis System) is an instantiation of the described architecture that supports synthesis of provably correct imperative programs by application of strategies. It is a research prototype designed with the above requirements in mind. We first establish the notions of problem, solution, acceptability, and

explanation that are used for IOSS. We then give an overview of its strategy base. Finally, we describe the implementation of IOSS that integrates several existing software packages. Examples giving an impression of how it "feels" to work with the system are presented in Chapter 8 and Appendix A.

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*. The invariant follows from the precondition and needs only to be maintained. The distinction between goals and invariants helps focussing on the relevant parts of the task. In addition to pre- and postconditions, it has to be specified 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 latter 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 former is not equivalent to true, the developed program can only be guaranteed to work if not only the originally specified, but also the additional precondition holds. 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.

Explanations for solutions are provided as formal proofs in dynamic logic [Gol82, HRS89]. 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

The strategy base of IOSS contains formalized development knowledge in form of strategy modules. A number of interactive, semi-automatic and fully automatic strategies have been implemented. In the current version, they are oriented on programming language constructs. In the near future, higher level strategies, e.g. for the development of divide-and-conquer algorithms or re-usable procedures, will be built in.

Strategies Solving a Problem Directly. Sometimes the precondition of a problem is sufficient for its goal, e.g., if a conditional only needs one branch. In this case, the empty program **skip** is developed using the *skip* strategy.

The two *assignment* strategies are used more frequently since assignments are the basic building blocks of imperative programs. In the interactive version, the assignment solving the problem has to be given by the user; for the automatic version, the goal must contain equations in some of the result variables; these are used to set up an assignment.

Strategies Modifying a Problem. 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.

Sometimes it is necessary to introduce a new state variable for some result variable. This is accomplished using the *state variable* strategy.

Strategies for Developing Compound Statements. The *intermediate assertion* strategy corresponds to the rule for compound statements in the Hoare calculus. There, an intermediate assertion is introduced which forms the postcondition of the first part of the compound and the precondition of the second part.

Two other strategies are based on the assumption that a conjunctive goal can be achieved by a compound statement, each part of the compound establishing one conjunct (see [Der83]). The *disjoint goal* strategy can be applied if the goal can be divided into two independent subgoals. Two subgoals are independent if the result variables that must be changed to achieve the one subgoal are disjoint from the result variables that have to be changed to achieve the other one. The strategy can also be applied if the goal is not of conjunctive form but there is an invariant which is invalidated by the achievement of the goal. The additional postcondition of the first statement may be necessary to develop the second part of the compound. Hence, the first statement must be developed first.

The *protection* strategy can be applied when a conjunctive goal is to be achieved by a compound statement but 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. Again, the problem for the second part of the compound depends on the solution for the first part.

Strategies for Developing Conditionals. The *conditional* strategy reflects the rule for conditionals of the Hoare calculus. The *disjunctive conditional* strategy applies if the goal is of disjunctive form and each of the branches of the conditional will establish one disjunct of the goal.

Strategies for Developing Loops. The *loop* strategy develops a loop for a given problem. 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 contrast, the *while* strategy (which is defined [Hei94] but not yet built into the system) performs the development of the invariant, the initialization and the loop body in a single reduction step, according to the heuristics given in [Gri81].

A complete description of these strategies can be found in [Hei94].

7.3 The Interface

In this section we give a brief description of the IOSS interface with its most important features. We restrain from explaining the trivial ones such as loading or saving a development (even though they are nevertheless important). The File menu contains basically what you would expect it to.

Figure 7.1 shows the general interface of IOSS, featuring a snapshot of the program synthesis described in Chapter 8. The main window displays the development task, represented

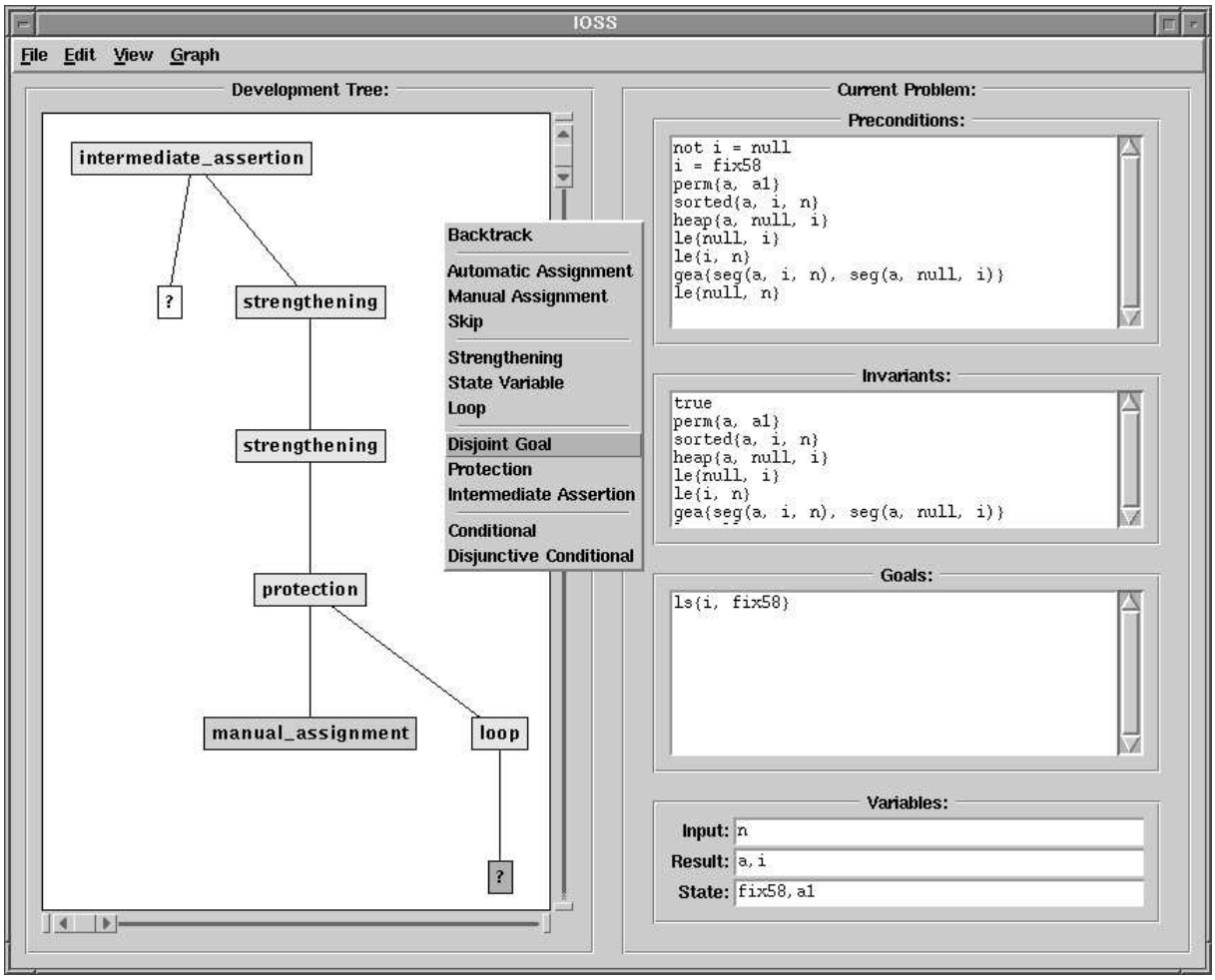


Figure 7.1: The IOSS interface

by the development tree – on the left-hand side of the window – and the specification of the current problem – on the right-hand side of the window. The tree visualizes the process and the state of development. Each node is labeled with the name of the strategy applied to it. The state of the node is color coded, showing at a glance whether it is reducible, or solved, etc.

A node is selected by simply clicking it with the mouse. If that node is reducible, it becomes the current node and its problem specification is shown on the right-hand side of the window. Any node can be selected for the purpose of inspecting it, but only reducible nodes can become the current node. A node can be inspected via the View menu. A separate window pops up for each node; several nodes can be inspected at the same time.

The explanation (i.e. proof) of the development process is accessible via the View menu, too. IOSS combines the explanations for each strategy application to form a coherent proof that – once the development process is completed – verifies the developed program (see Figure 8.3).

The strategy base of IOSS is accessible via the Edit menu. A strategy is applied to the current node by invoking the respective menu entry. In Figure 7.1 the menu is shown in the center of the window. It is possible for any menu to be kept on the screen at an arbitrary position. This allows developers to quickly access frequently used features such as the strategy base. Whenever a strategy requires user input, the user is prompted for it in a window. IOSS also provides features to manipulate the graph. The user can, for example, re-scale the tree or hide subgraphs.

7.4 Implementation of IOSS

Technically speaking, IOSS is not one single program. It makes use of a number of programs/software packages to realize what the user perceives as IOSS. This section describes how these packages are integrated.

We give a brief introduction to each package in Section 7.4.1 to aid understanding the role of each one in IOSS. Section 7.4.2 describes how these work together during the start-up of IOSS. We describe the interaction of and communication between the interface and the IOSS kernel in Section 7.4.3. Finally, Section 7.4.4 explains the interfacing with the graph visualization package used in IOSS.

7.4.1 Software Packages used in IOSS

Karlsruhe Interactive Verifier (KIV)

The *Karlsruhe Interactive Verifier* (KIV) is a shell for the implementation of proof methods for imperative programs [HRS88]. It provides a functional *Proof Programming Language* (PPL) with higher-order features and a backtrack mechanism. Assertions about programs can be formulated in Dynamic Logic [HRS89], an extension of Predicate Logic. The language for programs itself is a Pascal-like language with while-loops and recursive procedures. KIV serves for the implementation of the IOSS kernel, its data structures, and strategy modules.

daVinci

daVinci is a generic visualization system for directed graphs [FW94]. It performs the visualization of graphs only, i.e. it is not a graph editor or the like. The visualization of a graph is not static; daVinci offers a range of functions to manipulate the graph layout such as fine tuning, abstraction, and scaling. A graph is given to daVinci in a term representation composed of ASCII-characters. Attributes for nodes and edges can be specified that influence the visualization.

For use by other applications daVinci offers an application interface realized via standard terminal I/O (stdin and stdout). An application can connect to daVinci using, for example, pipe communication. The communication protocol consists of commands and answers. Commands are available for the transmission of a graph and for calling daVinci visualization and window operations, among others. daVinci reacts to the successful invocation of commands as well as to events like user selection (with the mouse) of nodes and edges. The system is used to visualize the development tree of IOSS and to allow users to manipulate it, e.g. to select nodes for strategy applications.

Tcl

Tcl (*Tool command language*) is a simple scripting language providing generic programming facilities such as variables, loops, and procedures [Ous94]. Its interpreter is a library of C procedures. An application can extend the Tcl *core* by additional commands. Although applications can be solely programmed in Tcl, this wasn't its original intention. Quite a number of extensions already exist, each adding specific commands to the Tcl core. Three of them (Tk, expect, TkSteal) are used for IOSS. Their existence was the actual reason for using Tcl at all, and we describe them in the subsequent paragraphs. No new Tcl commands have been implemented for IOSS. The specialized interpreter that is used in IOSS is illustrated in Figure 7.2.

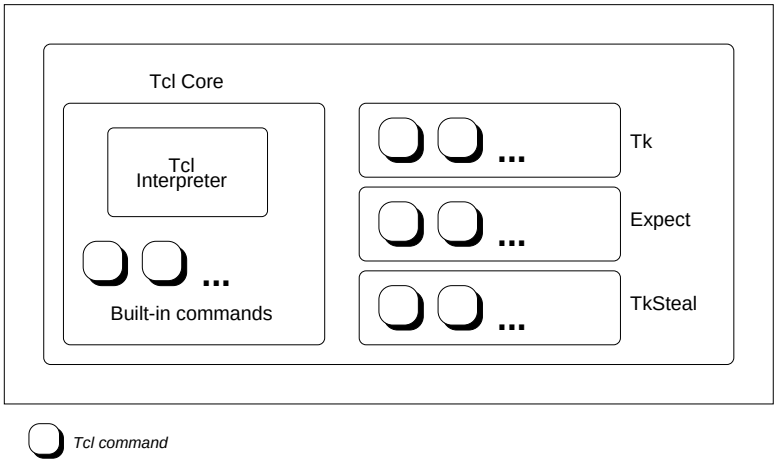


Figure 7.2: Specialized Tcl interpreter used in IOSS.

Tk

Tk is an extension to Tcl providing a toolkit for the X Window System [Ous94]. It extends the Tcl core by commands for building user interfaces. It hides much detail C programmers must address when constructing a user interface. The Tk commands are used for all the interface elements like menus, scrollbars, and text windows.

expect

expect is an extension to Tcl/Tk designed to control interactive programs using standard terminal I/O [Lib91]. For the controlled programs, expect takes over the part of users "typing" commands and interpreting output.

The program to be controlled is started with the command `spawn`. expect sets up a pseudo terminal (pty) connection with this program. With the command `expect` one can describe a list of patterns to watch for in the output of the spawned program. Each pattern is followed by an action that is executed if the pattern is found. The command `send_spawn` can be used to send input to the controlled program. expect is capable of controlling several programs at the same time. In IOSS it is used to control the command-line interface of KIV and communicate with daVinci's application interface.

TkSteal

TkSteal is an extension to Tk to integrate stand-alone X applications in a Tk-built interface [Del94]. Windows in the X Window System are arranged in a tree hierarchy, where top-level application windows are children of the root window and windows in a particular application are descendants of the applications top-level window. The visible extent of a window is normally limited to the extent of the parent window. A special function of the X Window Toolkit allows TkSteal to re-parent windows after their creation. TkSteal re-parents a specified top-level window to a Tk window causing the reparented window to appear inside the Tk window. Sizing the Tk window and placing the reparented window in the Tk window provide a means, for example, to hide the menu bar of the reparented window from view (and thus access). TkSteal integrates the daVinci window with the Tk-built user interface.

Figure 7.3 illustrates the integration of the different packages and their roles in IOSS: the Tcl-Interpreter and daVinci constitute the IOSS interface, KIV constitutes the kernel of IOSS.

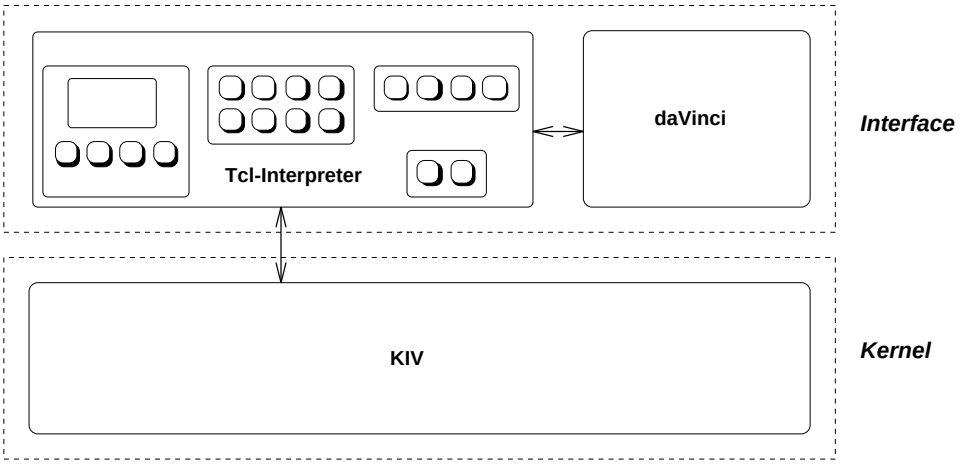


Figure 7.3: Integrated packages and their roles in IOSS.

7.4.2 IOSS Start-Up Procedure

During IOSS start-up the three binaries are started, communication between them is set up, and the daVinci interface is integrated with the Tk-built interface (Figure 7.4). Invoking IOSS from the command line starts the Tcl-Interpreter that interprets the Tcl-code for IOSS. First, the IOSS main window is created. Then KIV and daVinci are started from within the Tcl-Interpreter by the `spawn` command. The daVinci application window is incorporated into the Tk-built interface with the `tksteal` command (see Figure 7.5). Before the daVinci window can be "stolen", it must have been created and displayed. To ensure this, the Tcl-Interpreter waits for an `ok` answer from daVinci's application interface that is sent when daVinci is ready for communication. Waiting for the window to be displayed would mean that it appears on the screen *before* it is incorporated into the Tk-built interface. We avoid this by specifying a geometry that will cause the daVinci window to be displayed outside the visible area.[1] The

[1] First, this is only possible because daVinci supports a `-geometry` command-line option. Second, this may or may not work, depending on the window manager. It worked so far with these: fvwm, twm, olvwm.

Tcl-Interpreter is synchronized with KIV by watching its output for the prompt of the KIV shell. The prompt indicates that KIV is ready to receive input. At this point the initialization is complete and IOSS enters its base state where the user can initiate new developments or load previously saved ones.

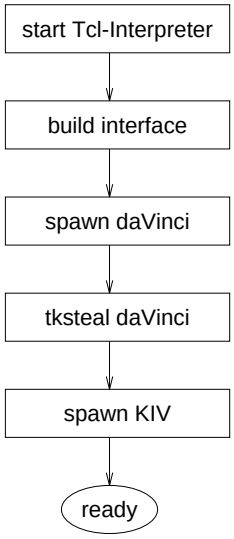


Figure 7.4: IOSS initialization.

Data flow and interaction between the packages during the development process are described in the two following sections and illustrated in Figure 7.6.

7.4.3 Interaction between Interface and Kernel

In interacting with the kernel, the interface has to follow the shell behavior of it. A prompt in the output of the kernel signals that input is requested. The input requested can be a command, special input requested by a called command, or general user input requested by a called strategy. The specific input requested is indicated by a pattern preceding the prompt and the prompt itself (displaying the PPL machine level). If a command is requested, the interface usually waits for a user action (e.g., menu selection) and sends the corresponding string to the kernel. A user action may also result in a sequence of kernel commands. For example, loading a previously saved development amounts to cancelling the current one, if there is any, then loading the development and entering the IOSS main loop. In this case the interface communicates the sequence of commands to the kernel, synchronized with the shell behavior of the kernel. If specific input is requested, it is either taken from the internal state of the interface (e.g., currently selected node) or the user is prompted for it in a special dialog (e.g., file name for saving or loading). For general input the user is given a window with a simulation of the shell of the kernel. During general user input, the interface displays all output received from the kernel in this window and in turn sends everything typed by the user to the kernel. The interface assumes a general input request whenever it reads the prompt in the kernel output without a recognized preceding pattern.

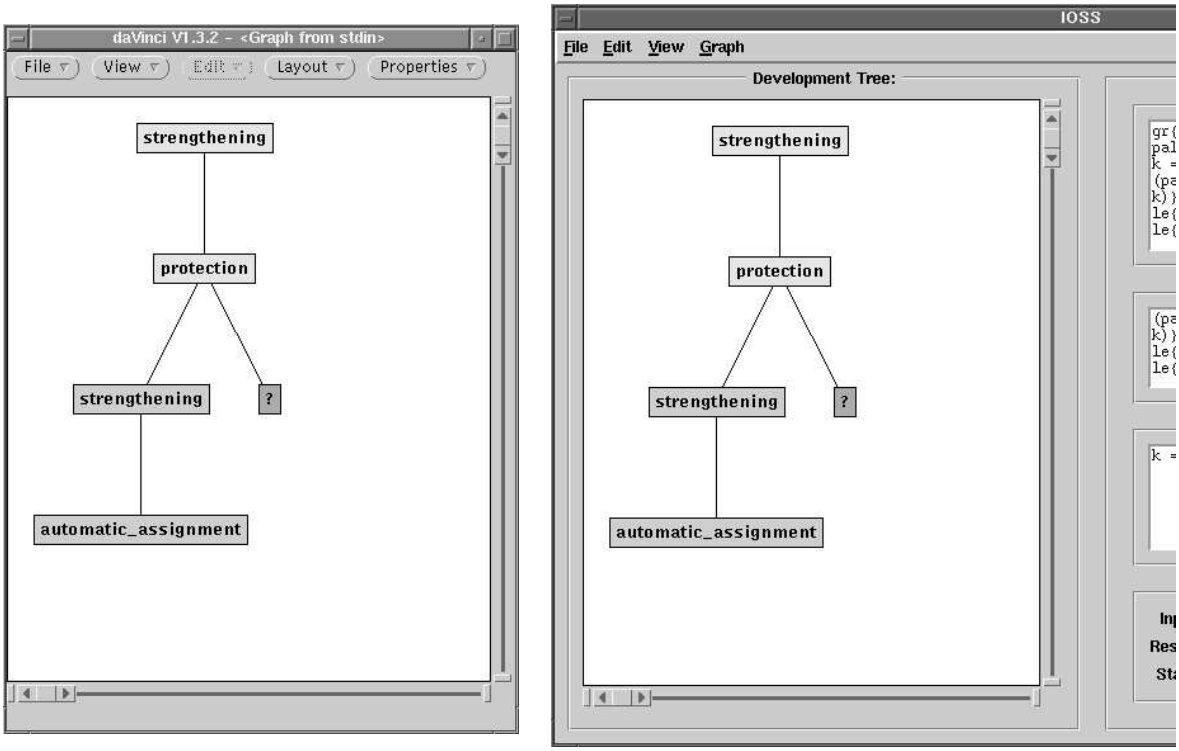


Figure 7.5: The daVinci application window stand-alone (left), and incorporated into the IOSS interface (right).

The kernel outputs information about its state and data to present to the user to the interface. Upon each entrance to the main loop, the interface receives

- the term representation of the development tree,
- the current problem, if there is any,
- the references to the reducible nodes, and
- the set of applicable commands.

Furthermore the kernel outputs

- problem and solution of a node when requested,
- state patterns, and
- patterns to specify input requests (as described above).

The data is enclosed in specific patterns for the interface to parse the output and extract the data. The term representation of the development tree is passed on to daVinci (see Section 7.4.4). Other data (e.g., the current problem) is passed to the respective Tk windows for display. State patterns give further information to the interface about the internal state of IOSS. The interface then takes appropriate action to visualize this state. For example, after a strategy application there is no current problem until one is selected. In this state, the previous

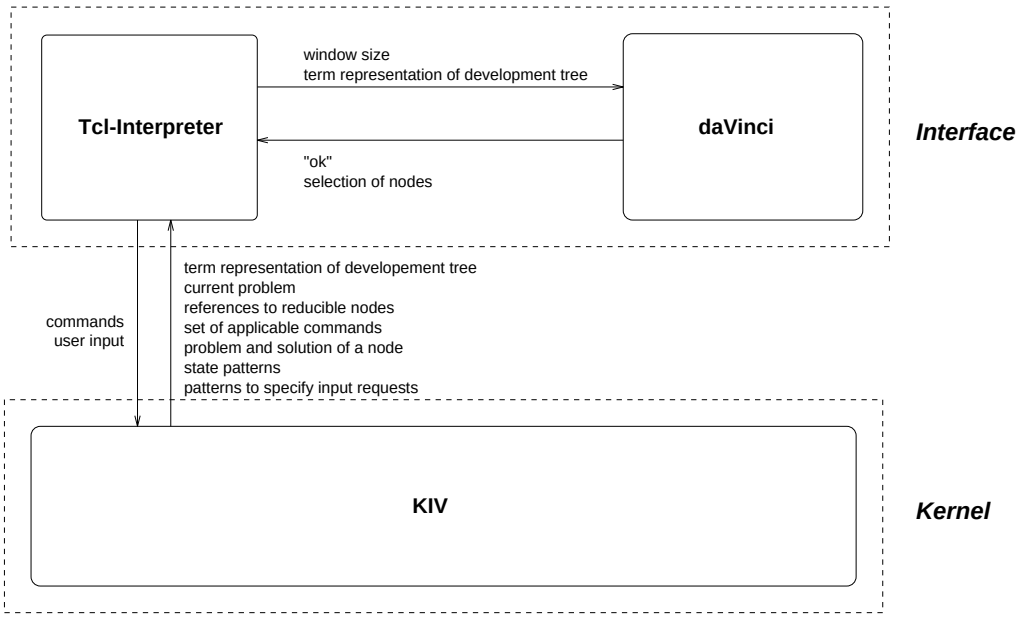


Figure 7.6: Data flow between the packages in IOSS.

problem is still displayed for reference to the user. This requires the respective label of the graphical user interface to be changed from "`Current Problem:`" to "`Previous Problem:`". The set of applicable commands provides reference for the interface to disable/enable menu entries according to the state of the kernel.

7.4.4 Interaction with daVinci

The commands available for communicating with daVinci fall within four groups:

1. sending graphs,
2. application specific menus,
3. user dialogues, and
4. triggering daVinci operations.

In IOSS only commands out of the first and the last group are used.

Whenever the kernel outputs a new term representation of the development tree, it is sent to daVinci with the `new_term` command. Since in the currently used version of daVinci (1.3) the visualization of nodes cannot be manipulated via the application interface, a new term is sent even if only the state of a node and thus its color in the visualization changed. New releases of daVinci provide commands for this without sending a completely new term, and IOSS will make use of this in the future.

When the user initiates a new development or loads a previously saved one, daVinci's window has to be cleared of any displayed graph. This is done by sending an empty term to daVinci.

Resizing of the IOSS window also requires communication with daVinci. Its size must be set to fit the size of its parenting Tk window. The required size is computed from the new size of the Tk window and communicated to daVinci with the `set_window_size` command.

Communication from daVinci to the Tcl-Interpreter regards the selection of nodes and `ok` messages after the successful invocation of a command via the application interface. daVinci reports the selection of nodes with the answer `node_selections_labels(`*strings*`)` (multiple nodes can be selected). The labels of the selected nodes are stored internally. They are needed in communication with the kernel for the set-up or display of nodes.

We think it is remarkable how little effort it took to build the interface. Only one person-month was required to build it in its current shape. Only 800 lines of code needed to be written in Tcl. The PPL code had to be extended by 116 lines of code.

Chapter 8

Heapsort: An Example Development

In this chapter we present a few selected steps from a sample development to show how development with IOSS proceeds. The task is to sort an array `a` of integers. To do this we want to develop a heapsort algorithm. The initial problem is shown in Figure 8.1(a)[1], where the goal `perm{a,a1}` requires the sorted array to be a permutation of the original one. The concept of the heapsort algorithm is to first build a heap[2], and then level down the heap putting the top (maximum) element at the end of the array and restoring the heap for the remaining unsorted segment of the array.

Since the program section that builds the heap will be almost identical in both parts of the algorithm, the idea is to develop the second part first and re-use the developed program section in the first part.

To start with, we apply the *intermediate assertion* strategy to the initial problem. This strategy allows us to choose which part of the compound we want to develop first. The strategy prompts us for an intermediate assertion. For the second part of the compound, we must ensure that the array `a` is a heap and that it is a permutation of the original array, denoted by the state variable `a1`:

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

The goals for the second subproblem yielded by the *intermediate assertion* strategy are the goals of the initial problem. With two applications of the *strengthening* strategy we use the fact that the empty array always is a heap and other domain-specific knowledge to replace these goals by stronger ones, resulting in the problem shown in Figure 8.1(b), where the last goal means that all elements in the second segment of the array are greater than or equal to all elements in the first segment.

Our approach now is to establish all goals but `i = null` in a first step, and then establish `i = null` with a loop, the formerly achieved goals comprising the invariant of the loop. We select the *protection* strategy, where the first statement will establish the loop invariant and the second will be the loop itself. The assignment `i := n` establishes the goals for the first

[1] 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.

[2] 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 $2i+2$.

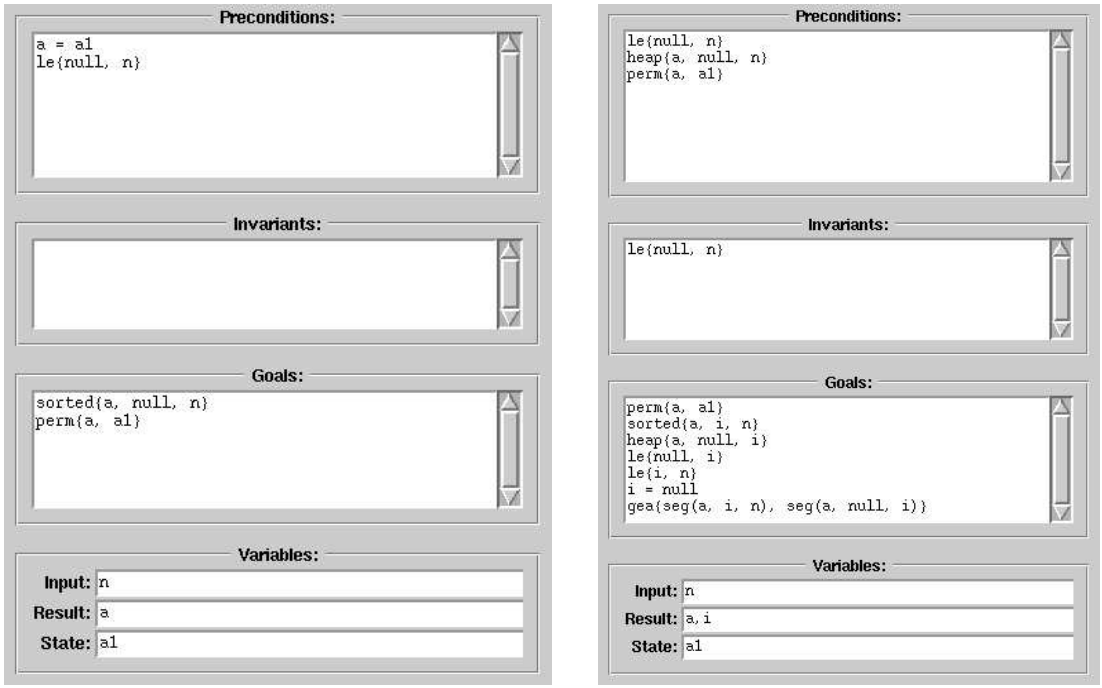


Figure 8.1: The problems (a) initially and (b) before the initialization of the loop

statement. We enter it using the *manual assignment* strategy. Since it does not generate any new problems, IOSS automatically calls the corresponding *assemble* and *accept* functions to check the solution for its acceptability. In particular, this may involve the invocation of the built-in theorem prover, proving the correctness of the assignment with respect to its specification.

After application of the *loop* strategy to the second subproblem yielded by the *protection* strategy, the system automatically selects the negation of the goal as the test for the loop: `not i = null`. To ensure termination of the loop, we have to interactively enter a bound function (`i`), a predicate for a well founded order (`ls`), and a least element with respect to the order (`null`). We may also supply additional invariants. The goal for the loop body is constructed from the bound function and the less predicate: it is to reduce the value of the bound function (while maintaining the invariant). The problem for the loop body is shown in Figure 8.2(a).

Reducing the value of the bound function will invalidate the invariant; it has to be re-established afterwards. We first apply the *disjoint goal* strategy. It automatically determines the only goal `ls{i,fix12}` to be the goal for the first part of the compound. The invariants of the problem automatically become the goals for the second part. Reducing the value of the bound function is trivial: `i := p(i)`. Re-establishing the invariant is done in two steps: swapping the first and last element of the heap,$a[0]$ and $a[i]$, and re-establishing the heap condition for the unsorted segment $a[0 \ldots i-1]$ of the array.

Since the unsorted segment is a heap except for the first element (resulting from the swapping), we let this element "descend down" in the tree. This is again achieved by a

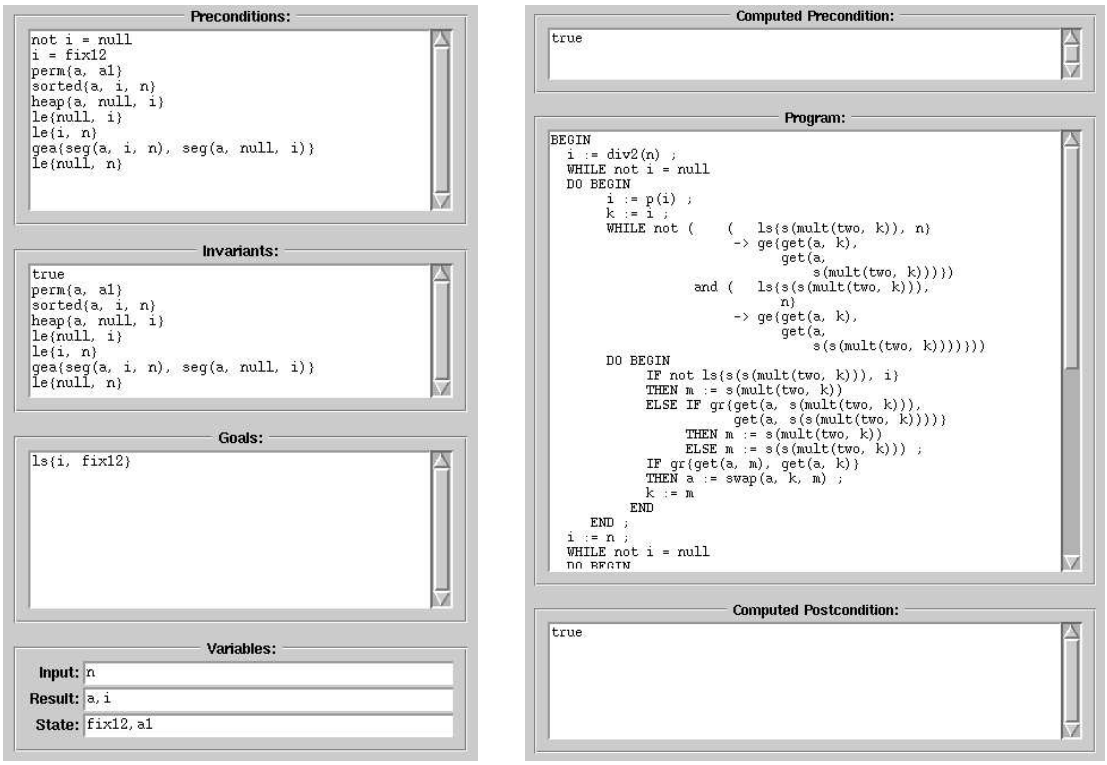


Figure 8.2: Problem for the loop body (a) and solution to the initial problem (b)

loop. It is synthesized with a similar approach as the first one. We first apply the *protection* strategy. Its first subproblem specifies the initialization of the loop, solved by `k := null`. Before we can apply the *loop* strategy to the second subproblem, we need to establish a goal appropriate for the termination of the loop with the *strengthening* strategy. Casually expressed, we're done when the element that descends down is at its proper place in the heap. The formula expressing this is:

$$(2k+1 < \ i \rightarrow \ a[k] \geq \ a[2k+1]) \wedge \ (2k+2 < \ i \rightarrow \ a[k] \geq \ a[2k+2])$$

Its prefix-ASCII notation as used in IOSS is:

```
      (ls{s(mult(two,k)),i} -> ge{get(a,k),get(a,s(mult(two,k)))})
and (ls{s(s(mult(two,k))),i} -> ge{get(a,k),get(a,s(s(mult(two,k))))})
```

Now we apply the *loop* strategy that selects the negation of the above formula as the test for the loop. For the development of the loop body we first need state variables for `a` and `k`. We get them by applications of the *state variable* strategy. In the loop body we need to determine the successor with which the descending element has to be exchanged, swap these two, and reduce the bound function to work towards the termination of the loop. We refrain from elaborating the development of the loop body in this place. It involves nested applications of the *disjunctive conditional* strategy, as well as application of the *conditional* strategy, the

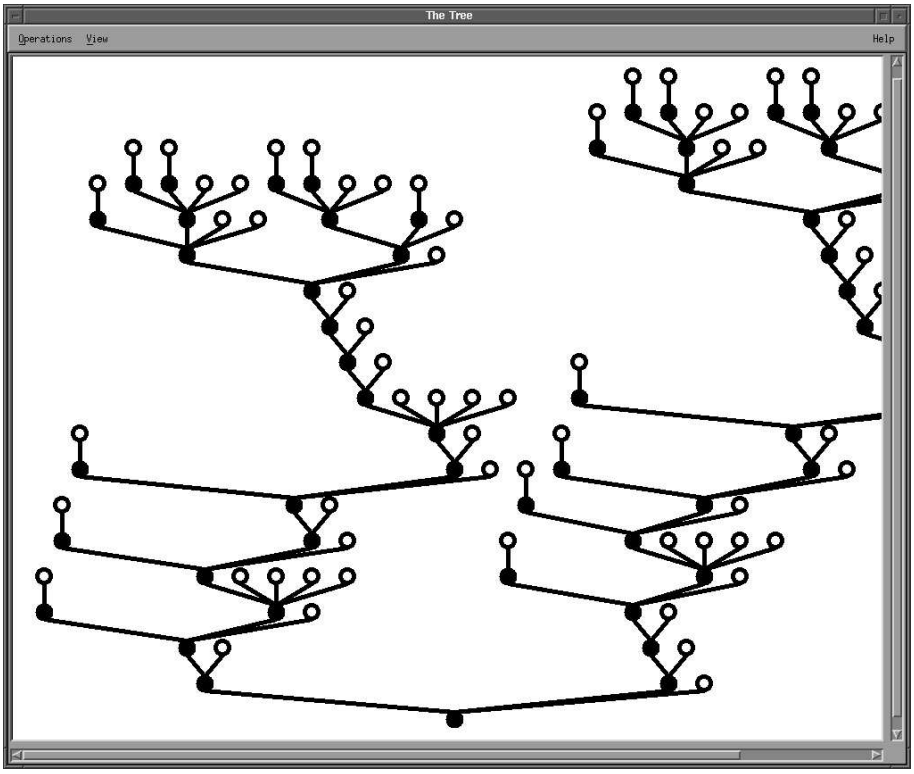


Figure 8.3: Proof tree for the heapsort algorithm.

skip strategy, and several applications of assignment and compound strategies. With the development of the loop, the second part of the heapsort algorithm has been completed. Developing the first part proceeds much in the same way as developing the second. Once the development is completed[3], we can inspect the root node of the development tree and have a look at the solution to the initial problem, shown in Figure 8.2(b). Figure 8.3 shows the proof tree for the developed program that was built by the system hand in hand with the development of the program.

[3]For the curious reader: The development of the heapsort algorithm involves a total of 54 strategy applications.

Chapter 9

Related Work

The work presented in this report relates to several areas of research:

CASE. As stated in the beginning, our general interest is to provide machine support for the software development process, thus improving the quality of the resulting products. This is exactly the aim of CASE. The support offered by our architecture is tailored for the application of formal methods. Hence, it can be seen as a special case of CASE technology. It complements conventional CASE tools. But not only the general objective, also the more concrete assessment of goals, problems, and enabling technology as they are stated in [FN92] coincide to a surprisingly high extent.

According to Forte and Norman, "among the greatest challenges ahead is the need for tighter *integration* among tools in a manner that supports *openness* to a variety of methods, notations, processes, tools, and platforms." (emphasis ours). As can be seen (not only) from the name of our synthesis system, we also consider these as key requirements. We agree with Forte and Norman that generic processes should be defined, and that the lack of a single method adequately addressing all application domains makes highly tailorable CASE tools necessary.

The difference between existing CASE technology and our approach lies in the means which are applied to reach these goals. Forte and Norman name simulators, dynamic code analyzers and rapid prototyping environments as second-generation CASE tools. Such "dynamic tools" alone can certainly not guarantee semantic properties of software products. In this respect, our architecture provides an alternative way of achieving the same goals as CASE for the special case of formal methods application.

Knowledge-Based Software Engineering (KBSE). This discipline seeks to support software engineering by artificial intelligence techniques. It comprises a variety of approaches to specification acquisition and program synthesis, see [LD89, LM91]. Our approach could also be subsumed under this field since a knowledge representation mechanism is the heart of the architecture.

A prominent example of KBSE which is close to our aims is the Programmer's Apprentice project [RW88]. There, programming knowledge is represented by *clichés*. These are prototypical examples of the artifacts in question, e.g. programs, requirements documents or designs. They can contain schematic parts. The programming task is then performed by "inspection", i.e. choice of an appropriate cliché and its customization by combination with other clichés, instantiation of schematic parts, and structural changes. This is achieved by high-level editing commands. The assumption underlying the Apprentice approach is that

a library of prototypical examples provides better user support than the representation of general-purpose knowledge. Our position is to prefer general-purpose knowledge because clichés to a large extent depend on the application domain. This makes 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 [Wil83] presents the development language Paddle. Paddle is similar to conventional programming languages. Its control structures are called *goal structures*. Paddle programs are a means to express developments, i.e. procedures to transform a specification into a program. Since performing the thus specified process consists of executing the corresponding program, a disadvantage of this procedural representation of process knowledge is that it enforces a strict depth-first left-to-right processing of the goal structure. This restriction also applies to more recent approaches to represent software development processes by process programming languages [Ost87, SSW92].

Potts [Pot89] uses *Issue-based Information Systems* (IBIS) [CB88] to represent design methods. Not only is represented what to do in which order when a design step is performed. *Reasons* for design decisions are also recorded: each design step raises certain *issues*. Different possibilities to resolve an issue are called *positions*. Finally, an IBIS contains *arguments* in favor or against positions. This rich structure causes representations of even small examples to become very complicated and hard to comprehend. Moreover, it is hard to see how to represent methods of sufficient generality in this framework without reference to the problem that is to be solved by them: arguments in favor or against a position usually depend on the application domain that is considered: in the examples of [Pot89], for instance, commands for elevator motors have to be taken into account. Although we acknowledge the desirability to record the reasons for design steps, as will be discussed in Section 10.1, we prefer a non-formal representation in this case because such knowledge lacks uniformity.

Souquières [Sou93] 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. 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.

Program Synthesis. We have presented a program synthesis system as an instance of our system architecture. As such, IOSS is a specialization of the general concepts presented in Chapters 2 to 6. Compared to other synthesis systems, however, it is more general because it is not specialized to support a particular method but serves to integrate a variety of methods which can be expressed in its basic formalism.

The synthesis systems CIP [CIP85, CIP87], PROSPECTRA [HKB93] and LOPS [BH84] are all designed to support a specific underlying method. These are not intended to integrate with other methods, nor are these systems customizable. Moreover, the support of other activities than program synthesis was not a design goal for these systems.

The approach underlying KIDS [Smi85, Smi90] 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.

Recent work at the Kestrel Institute aims at curing these deficiencies: The *classification approach to design*[Smi93a] consists of capturing software design knowledge in a hierarchy of *design theories*. A design task is solved by successively classifying the task along the hierarchy as instances of increasingly specialized design theories. Design theories are basically parameterized algebraic specifications. Associated with each is a schematic solution. Solutions lower down in the hierarchy are intended to be "better", more specialized than solutions for theories further up. At the end of the classification process, a solution optimal with respect to the design knowledge represented in the theory hierarchy is obtained by instantiating the schematic solution associated to the most specialized design theory found for the task.

The aim of this effort is to come to a purely logical, descriptive representation of design knowledge. This would admit a uniform procedure to constructively show that a task is an instance of a particular design theory [Smi93b].

Tactical Theorem Proving. Tactical theorem proving has first been employed in Edinburgh LCF [Mil72]. 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 [Pau88] and in KIV.

KIV [HRS91], in the version underlying IOSS, is a shell for the implementation of proof methods for imperative programming. Recently, KIV has been specialized to support the verification of program modules according to a fixed strategy [Rei92]. The degree of automation achieved with this strategy is impressive. Since in program verification both the specification and the program are known, automation is much easier to achieve than in program synthesis.

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.

Chapter 10

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 Section 9).

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.

Processes and formalisms are orthogonal in that the ideas of what is done during a development are often similar, even in different formalisms. Differences usually appear when carrying out a development step in detail. This fact led us (i) to a uniform notion of strategy which stresses similarities in methods independently of the underlying formalism, and (ii) to a generic architecture that allows one to exploit these similarities and customize the system for the formalism suited best in a given situation.

Let us review the requirements stated in Chapter 1.

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. Consequently, 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. The development tree contains explanations for all strategy applications. This improves comprehensibility of the product and may be used as a basis to conduct inspections by certification authorities.

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.

10.1 Future Improvements

Guaranteeing semantic properties imposes limitations that cannot be overcome. The number of subproblems is fixed for each strategy. Bottom-up steps are limited to single re-use steps, e.g. searching a library. Still, there are a number of improvements to widen the architecture's range of application.

With the strategies of the current version, program synthesis with IOSS is a time-consuming and highly interactive task. This is due to the fact that the current strategies are quite low-level. Higher-level strategies with a better potential for automation are already designed [Hei94, Hei92] and can be incorporated in the near future.

Another weakness of the current implementation concerns the proof of predicate logic formulas. The theorem prover of KIV is not very sophisticated and knows nothing, e.g., about ordering relations. It is worthwhile to improve the prover by parameterizing it with theories and incorporating rewriting techniques. Work on this problem is in progress.

The possibility to work with incomplete solutions will be introduced in the near future. This will add further flexibility to the development process.

Until now, there is only one version of the development tree. In order to explore other ways of proceeding, one has to store the different versions explicitly. To switch between these, one has to load the other version anew. To support a more explorative style of development, it is desirable to allow several alternative development (and control) trees in a single development.

In the current version of IOSS, no heuristics are implemented that could provide guidance in the selection of strategies, because only very general such heuristics are known to date. A first step towards the elicitation of heuristics is an empirical approach: whenever a strategy is selected, the reasons for the selection should be recorded. This is also very important if we

want to make the development comprehensible for persons who did not perform it, but want to re-use it.

Re-use is also an important feature that is only realized in form of a strategy for selecting items of a library. It is our intention to provide better support for re-use in the future. Here, the problems are of a technical nature: for example, different development trees have to be merged without identifying different information that by chance is referred to by the same name.

Despite its customizability for different formalisms, an instance of the architecture supports only one formalism. It requires some deeper research to come up with a concept that allows for the integration of different instantiations of the architecture. With such a system, a wider range of the software life cycle could be supported in an integrated way, not only different methods to achieve the same purpose.

Last not least, we consider our approach not to be confined to the application of formal methods. It should be possible to also support informal methods by strategies. A promising candidate in this respect is specification acquisition. Here, the transition from informal requirements to formal specifications is made. This means that the problems cannot have a formal semantics, whereas the solutions have. Acceptable specifications could be required to possess certain properties.

Appendix A

Palindrome Test: A Complete Development

We now present the complete development of a simple program. The development is illustrated by snapshots of the screen after each major step.

The task is to test if a given string is a palindrome. Strings are represented as arrays. A string is a palindrome if it reads backwards the same as forwards. Formally, this is expressed as follows:

$$\forall i.0 \leq i < n \Rightarrow str[i] = str[n - i - 1]$$

We abbreviate this condition by $palindrome(str, 0, n)$, i.e. the upper bound is *not* included. We also need to express that a substring of a string is a palindrome:

$$palindrome(str, i, j) \Leftrightarrow \forall l.i \leq l < j \Rightarrow str[l] = str[j - l - 1]$$

A substring of a string consisting of at most one element is always a palindrome. Hence,

$$palindrome(str, div2(n), n - div2(n)) \tag{A.1}$$

holds for any $n > 0$. The function $div2$ denotes the integer division by 2. To determine if the given string is a palindrome, we develop a loop, working from the middle of the string outwards to both ends (see Figure A.1). For this purpose, we use the index variable k that is initially set to $div2(n)$. The variable pal records if str can still be a palindrome ($pal = 1$) or a non-matching pair of elements has been found ($pal = 0$). In each execution of the loop body, k is decreased by one. If $pal = 1$ the elements $str[k]$ and $str[n - k - 1]$ are compared (the old value of k is denoted by k'). If these are equal, pal remains unchanged, otherwise it is set to 0. The loop terminates when $k = 0$.

We now realize this development idea in IOSS. The first thing to do is to load a theory file containing declarations for all variables, constants, functions, and predicates. It also contains axioms describing properties of the declared constants, functions, and predicates.

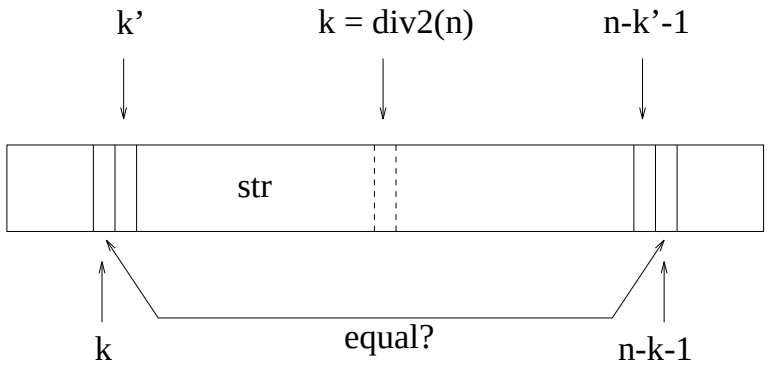
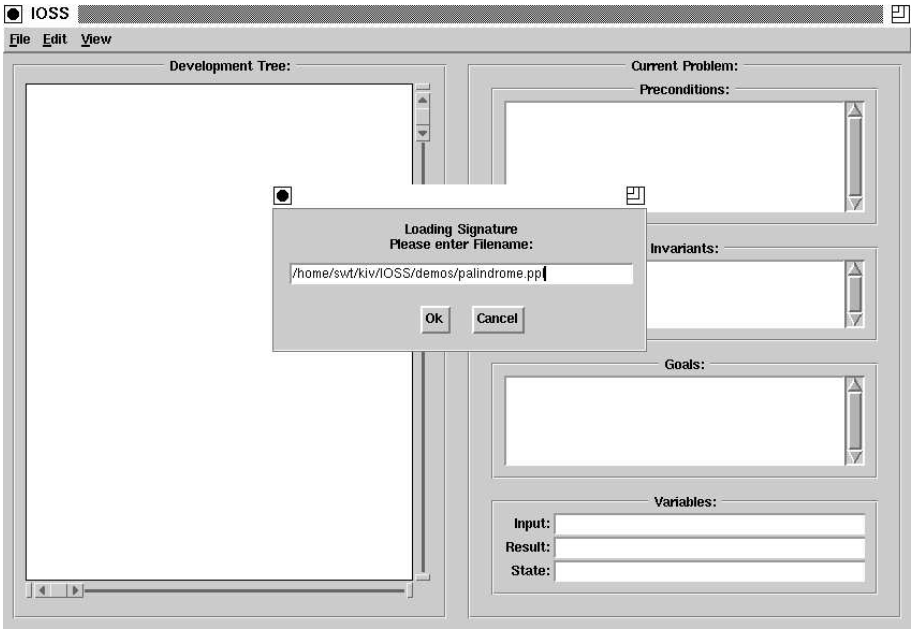
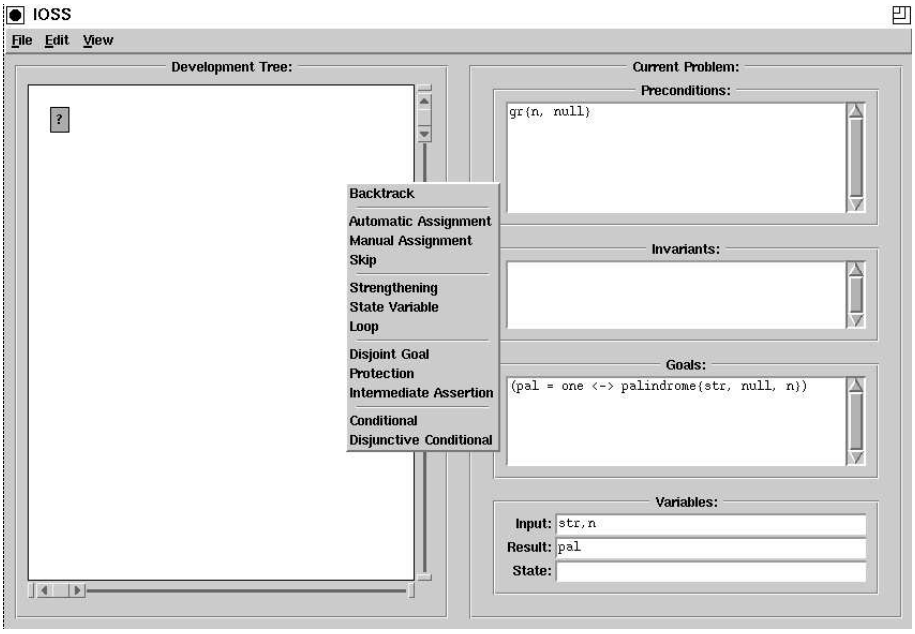


Figure A.1: Development Idea for Palindrome Program

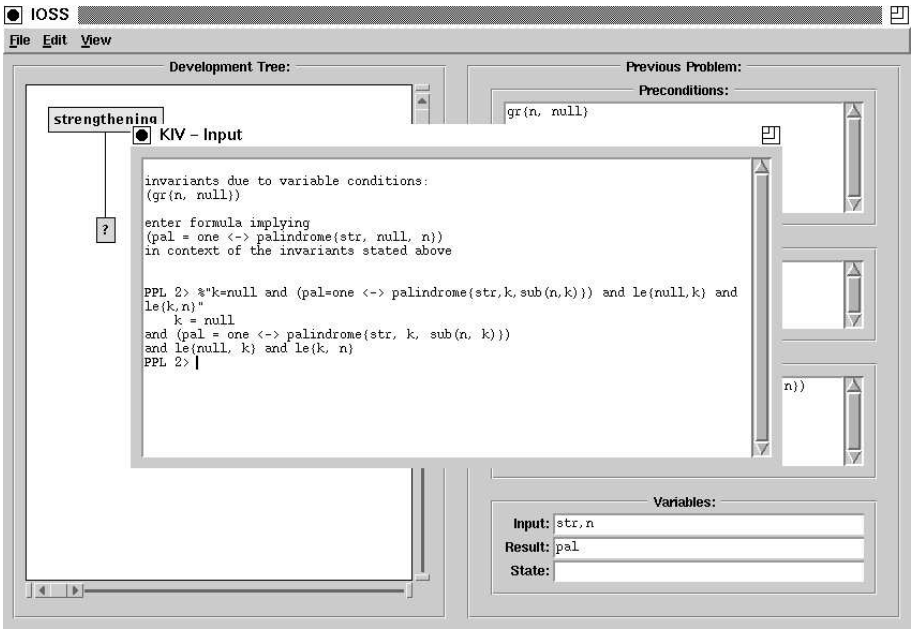


After loading the signature, IOSS is ready to accept our programming problem. After we have typed in its components, the initial problem is displayed on the right hand side of the IOSS window. So far, the development tree on the left hand side contains only the node for the initial problem.

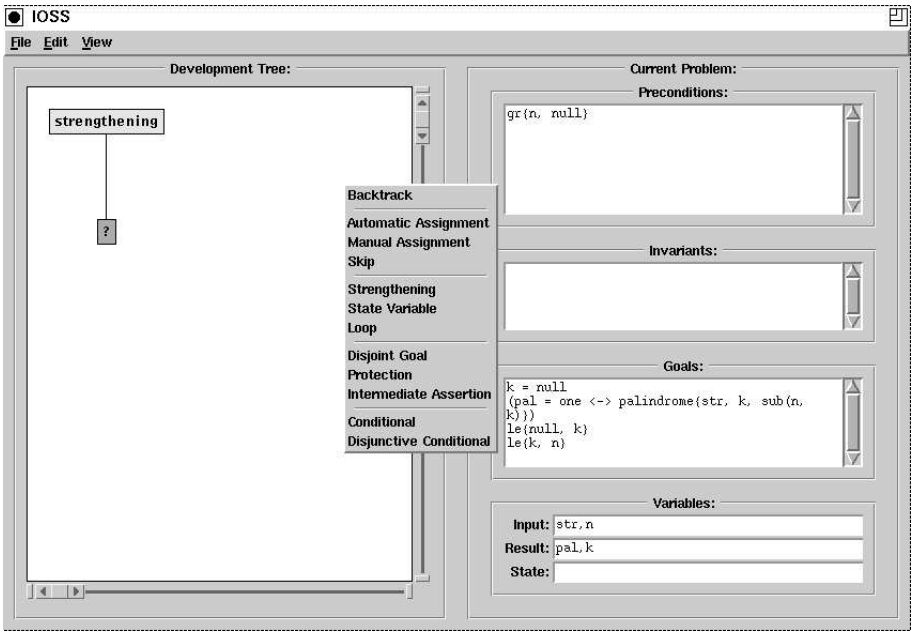


As often is the case, we start program development with an application of the *strengthening* strategy. Here, we follow the heuristics to replace a constant (n) by a variable (k) with suitable bounds. We also use the fact that $n - 0 = n$.

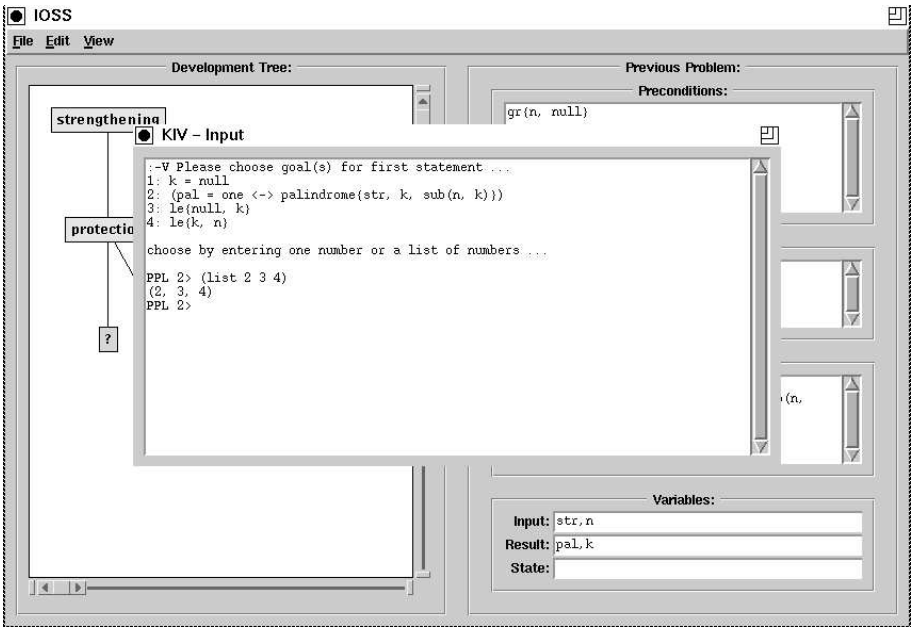
The input window shows the goal to strengthen and invariants from the precondition that may be used to show that the formula we type in indeed implies the original goal.



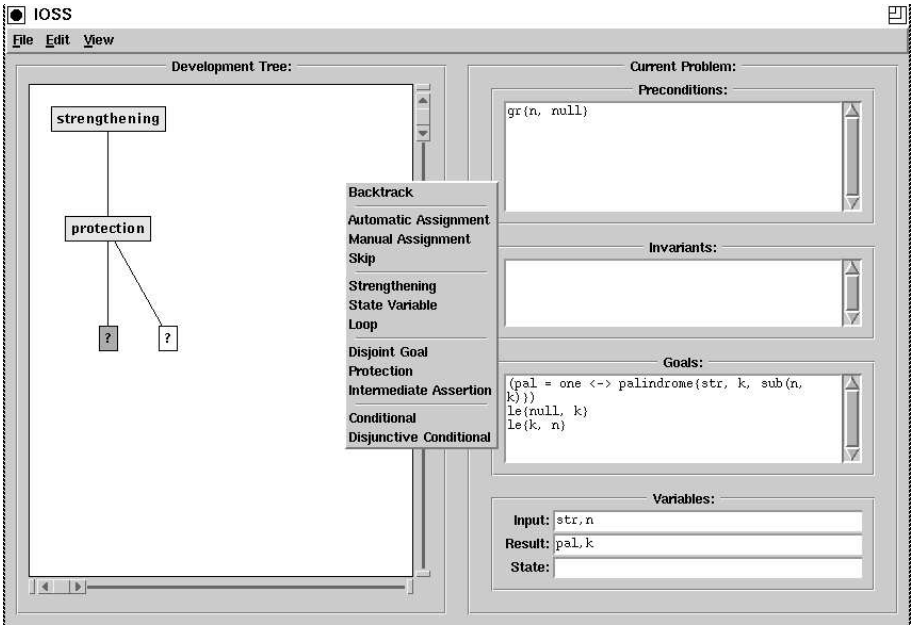
The original goal has been replaced by the stronger one and the development tree has been extended by a node containing the stronger goal. The new variable k that has been introduced by the strengthening has been classified as a result variable.



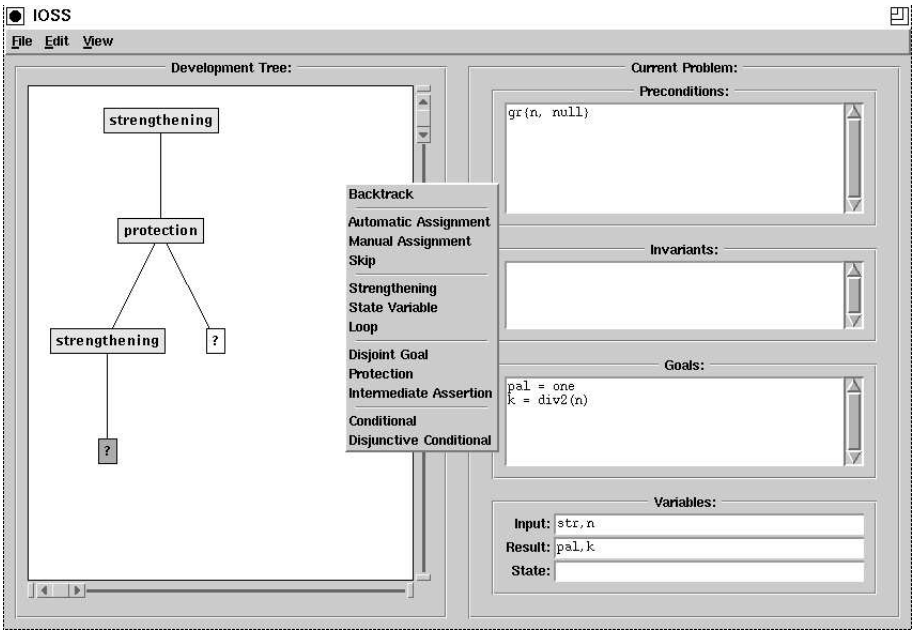
In order to develop a loop together with its initialization, we apply the *protection* strategy. The first part of the compound statement becomes the initialization for the loop. Setting up the problem for this statement means to select those parts of the goal that have to be established by the initialization, i.e. that will form the loop invariant.



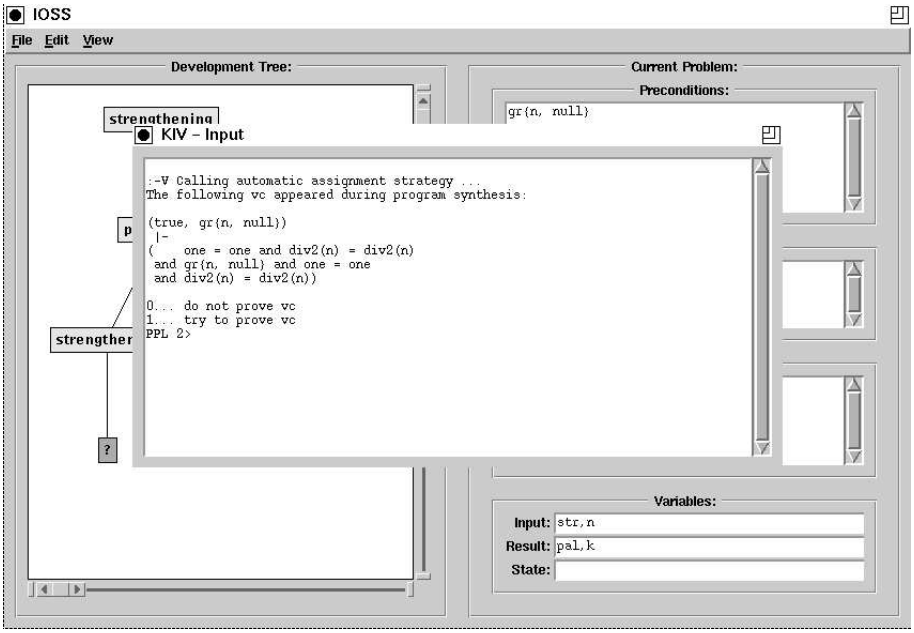
The new problem is to establish the loop invariant.



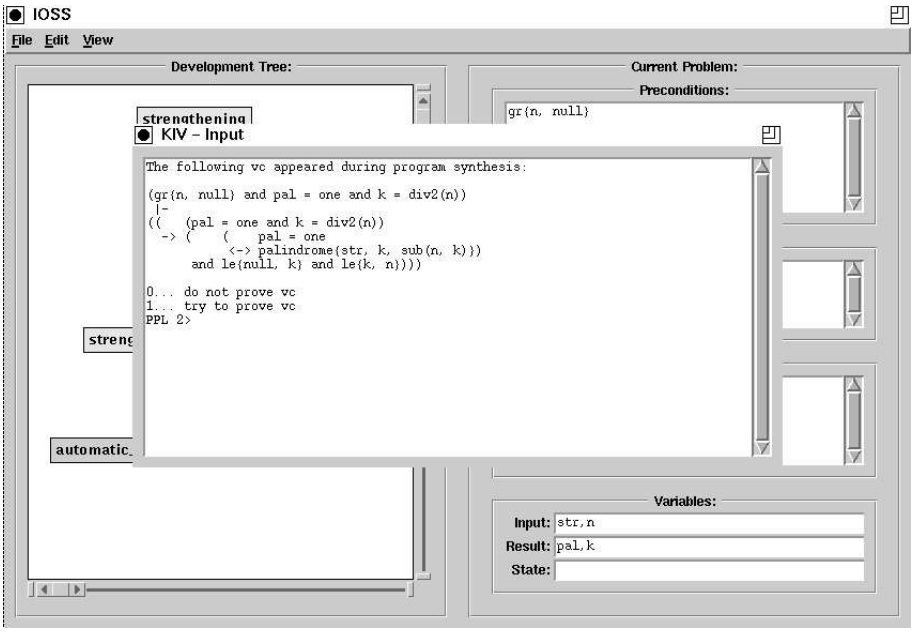
If $pal = 1$ and $k = div2(n)$, our goal is achieved by (A.1). To make use of this fact, we apply the *strengthening* strategy.



Our goals now contain equations for the result variables that can automatically be transformed into assignments by the *automatic assignment* strategy. The generated assignments are correct if their weakest precondition with respect to the postcondition follows from the precondition. This weakest precondition is computed and presented to us by the system because the built-in theorem prover is in interactive mode. This means that we can decide for each verification condition whether to attempt a proof or not. In this case, the verification condition is easy and we decide to prove it.

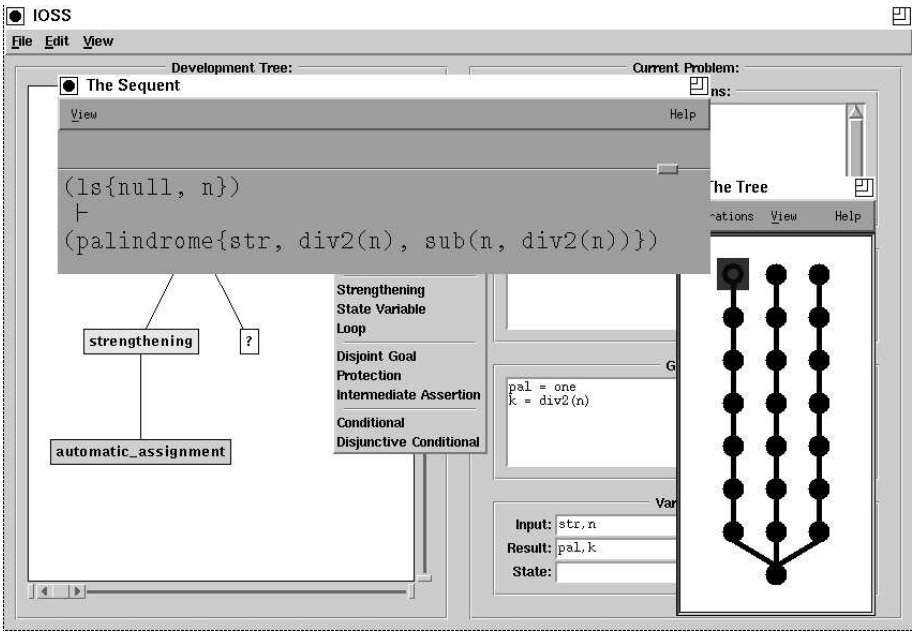


The next proof obligation stems from the strengthening applied prior to generating the assignments. Again, we decide to prove it by machine.

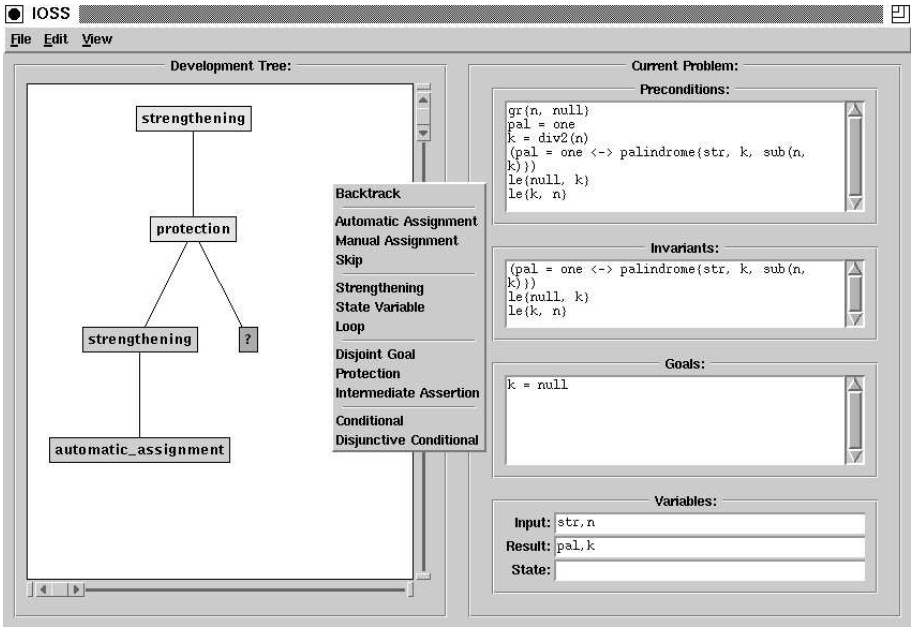


The prover does not have enough knowledge about palindromes. Thus it cannot fully prove the verification condition but reduce it to a number of sufficient conditions that we are presented together with the generated proof tree[1]. In this case, the remaining premise of the proof is just (A.1).

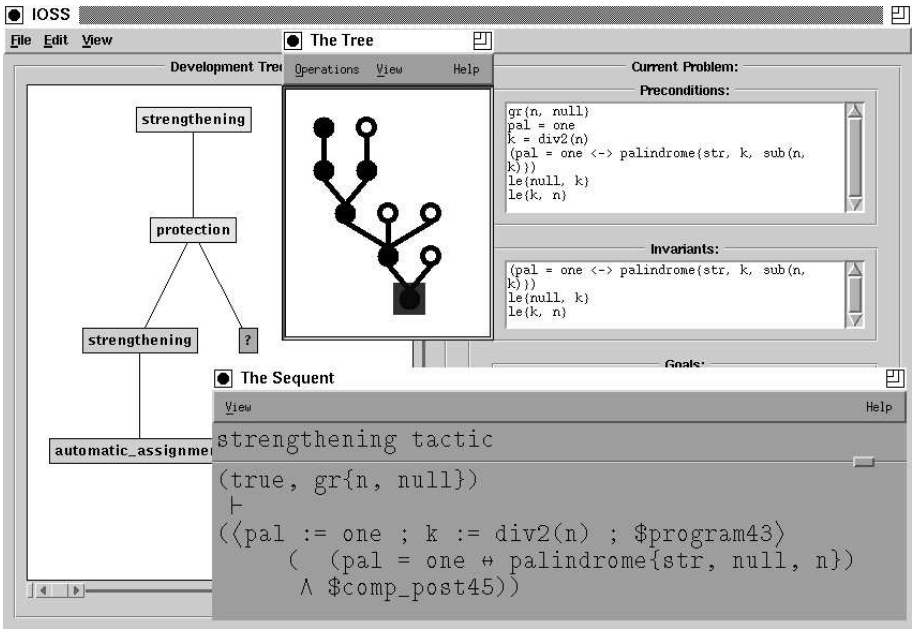
[1] We skip the verification conditions for the rest of the development.



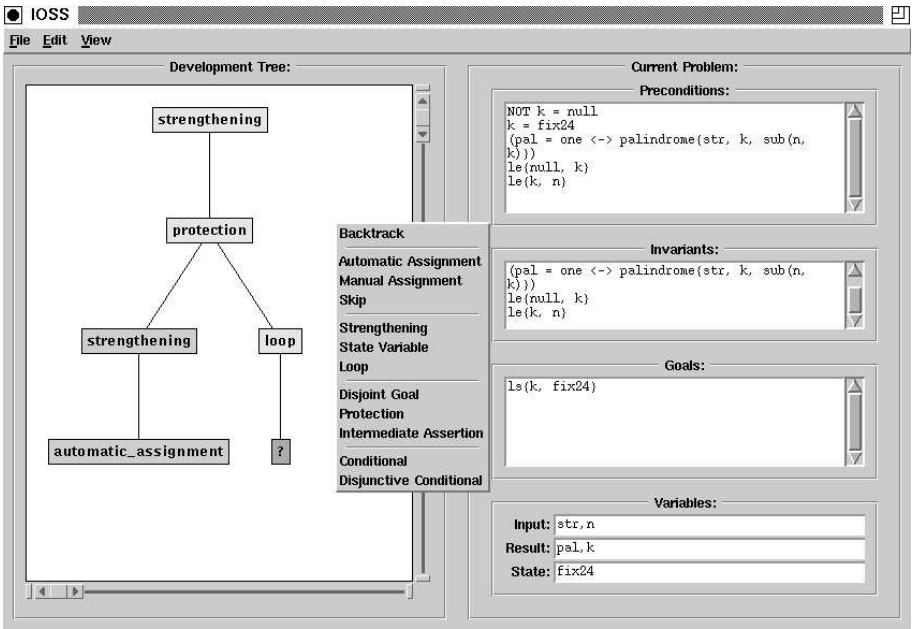
For the second part of the compound, we have to establish $k = 0$ while maintaining the invariant.



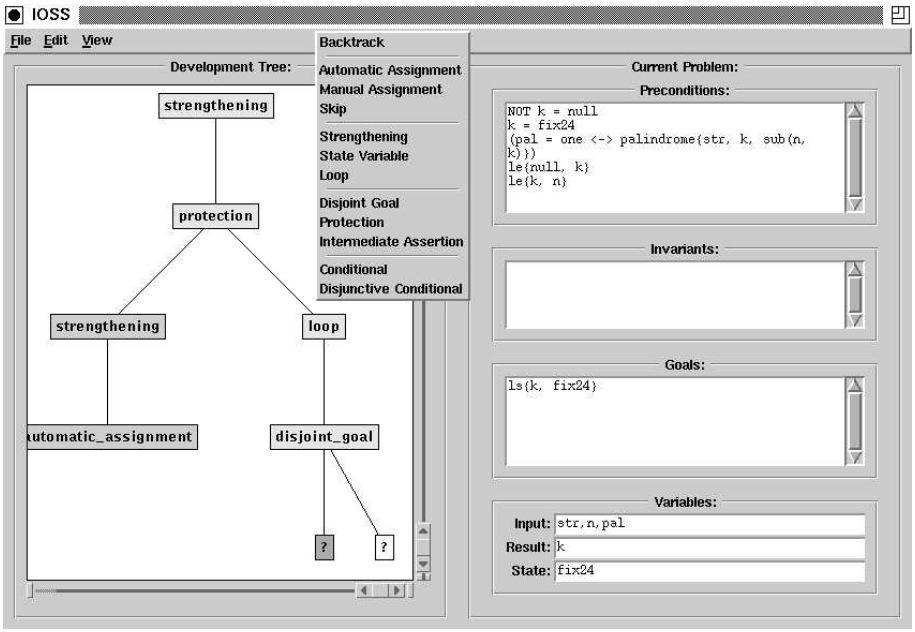
Before we start working on this problem, we decide to have a look at the current proof tree. Each of its nodes can be inspected. The root node shows the program developed so far. For the statement and the additional postcondition not yet known, the sequent contains the metavariables $\$program43$ and $\$comp\_post45$.



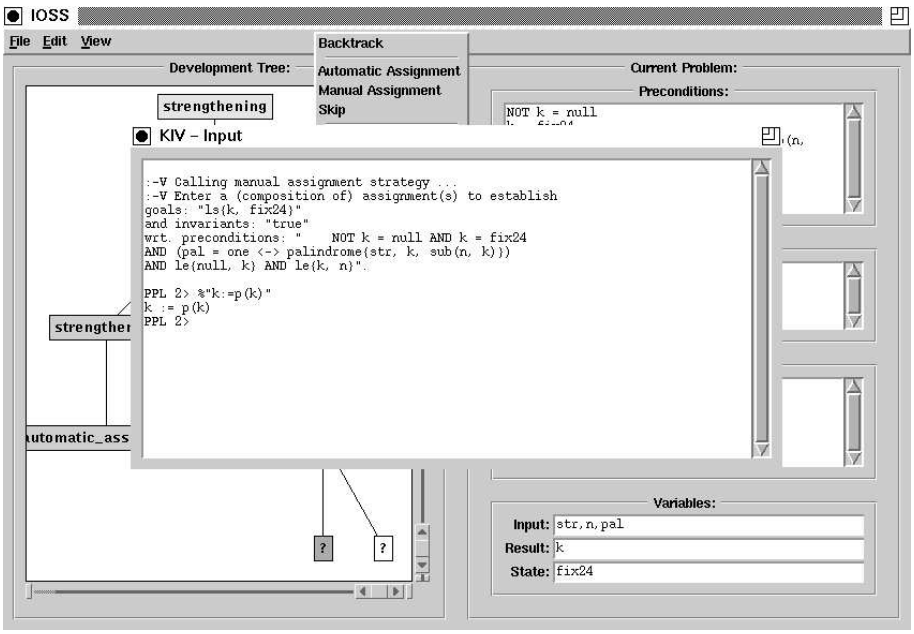
Now for the development of the loop: The *loop* strategy asks us to supply a bound function to ensure termination of the loop. Since k is to be decreased – the goal being $k = 0$ – we can use just this variable. The goal for the loop body is to decrease the bound function while maintaining the invariant. Therefore, a new state variable $fix24$ referring to the value of the bound function before entering the loop body is automatically introduced. In the precondition we get $k = fix24$ and the new goal is $k < fix24$.



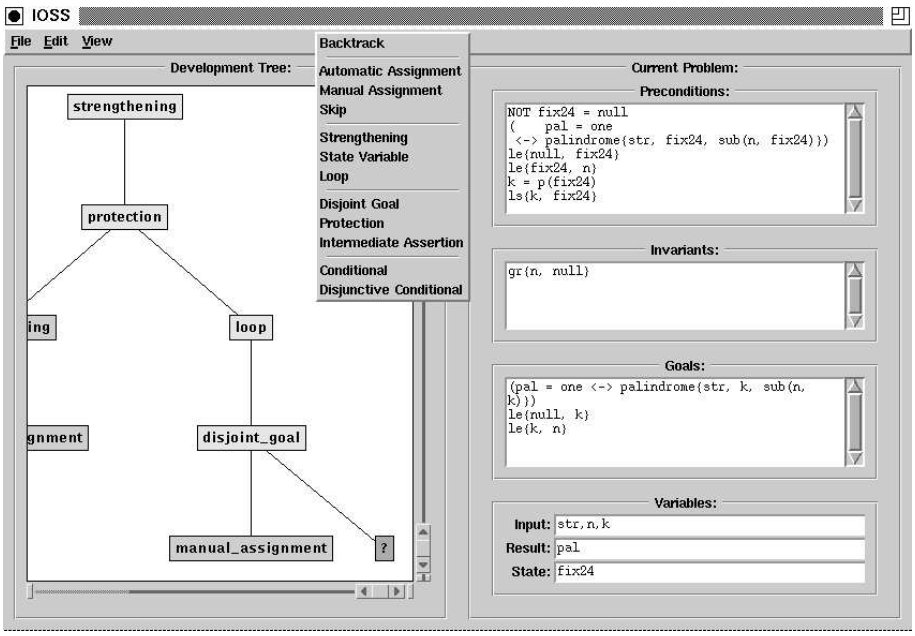
Decreasing k by 1 achieves this goal but also destroys the invariant. It has to be re-established in a second step by updating *pal* appropriately. Since the variables to be changed are different for the two steps we may use the *disjoint goal* strategy.



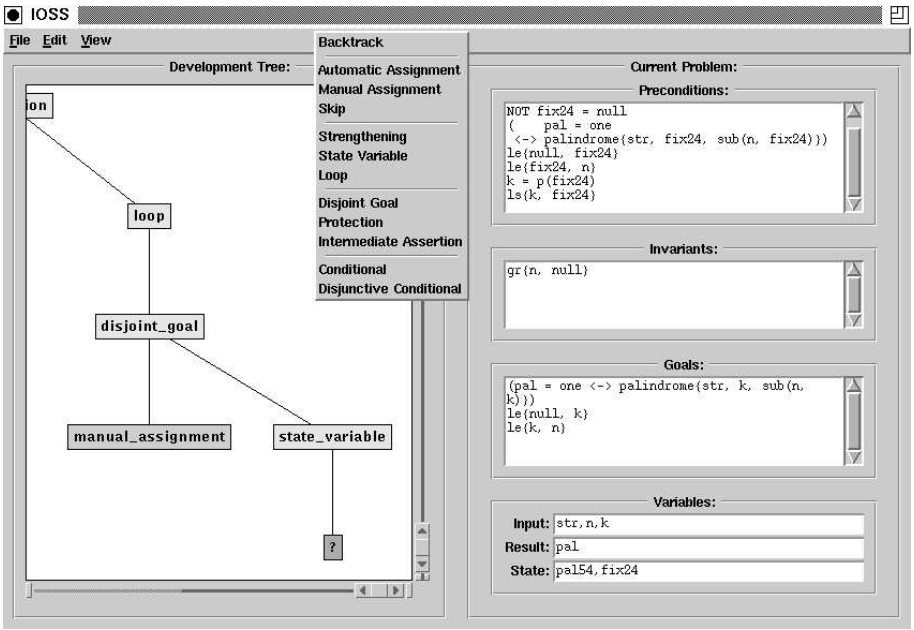
Since we know exactly what to do, we do not bother to massage the goal any further. We apply the *manual assignment* strategy and give – in IOSS syntax – the assignment $k := k-1$.



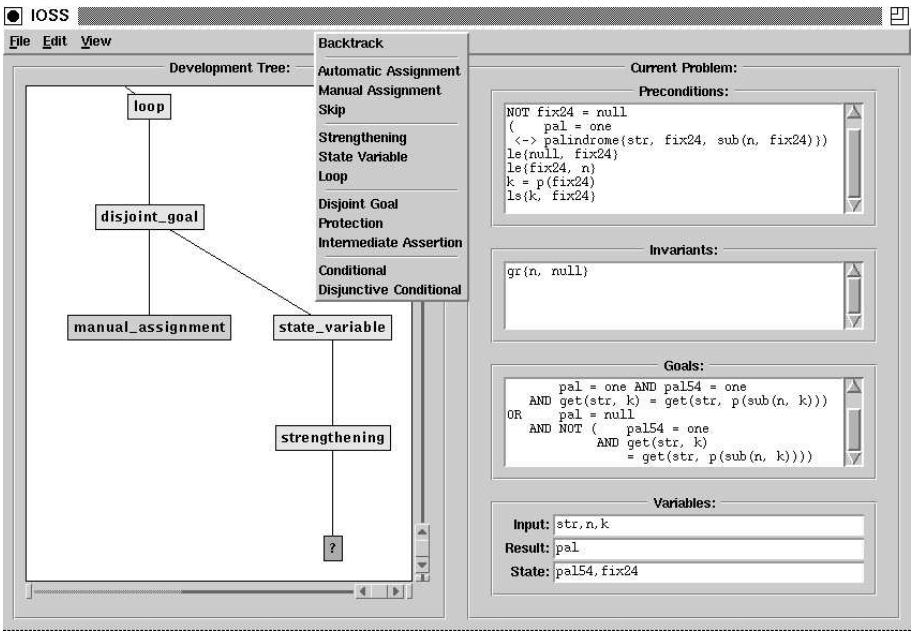
We are left with the task to re-establish the invariant by changing only *pal*.



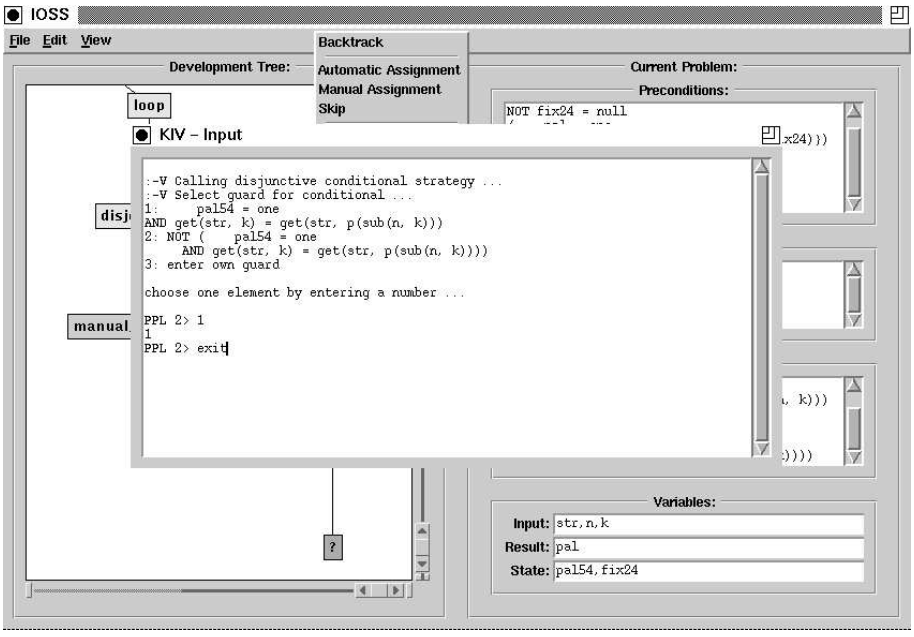
In order to express how *pal* must be updated, we have to refer to its old value. Once *pal* has been set to 0 it must not be set to 1. Therefore, we need a new state variable for *pal*. We get it by applying the *state variable* strategy.



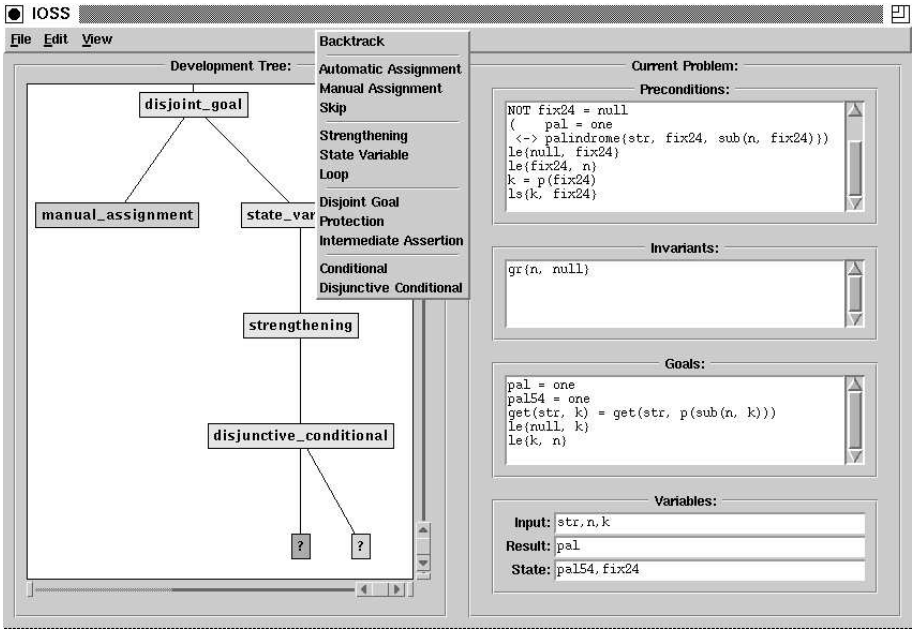
Now we are ready to strengthen the goal: if the old value of *pal* is 1 and the next elements to be compared are equal, *pal* remains unchanged. Otherwise, *pal* must be set to 0. We express this case distinction by a disjunction.



The disjunctive goal allows us to apply the *disjunctive conditional* strategy. The system proposes several candidates for the test of the conditional. We choose the first one.



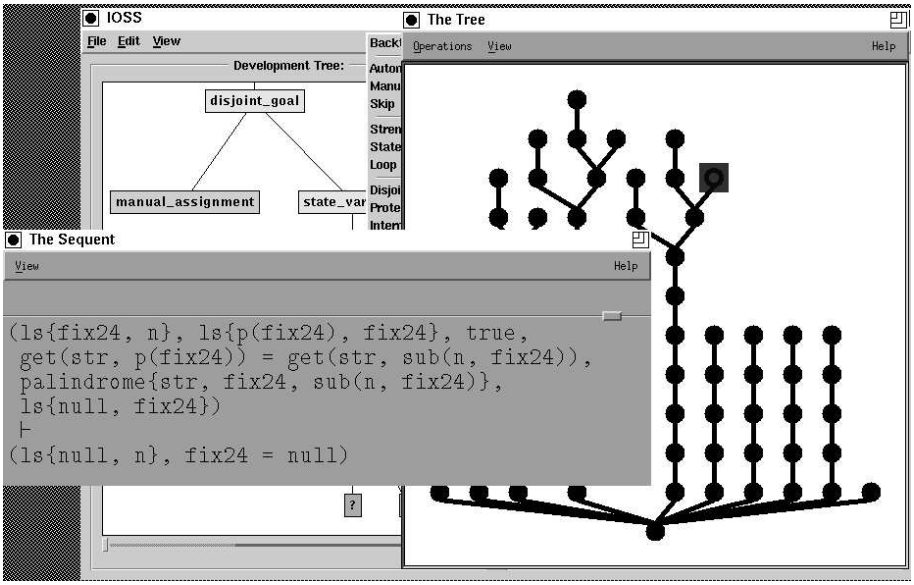
The goals of both branches of the conditional contain an equation for *pal* and can be solved by the *automatic assignment* strategy.



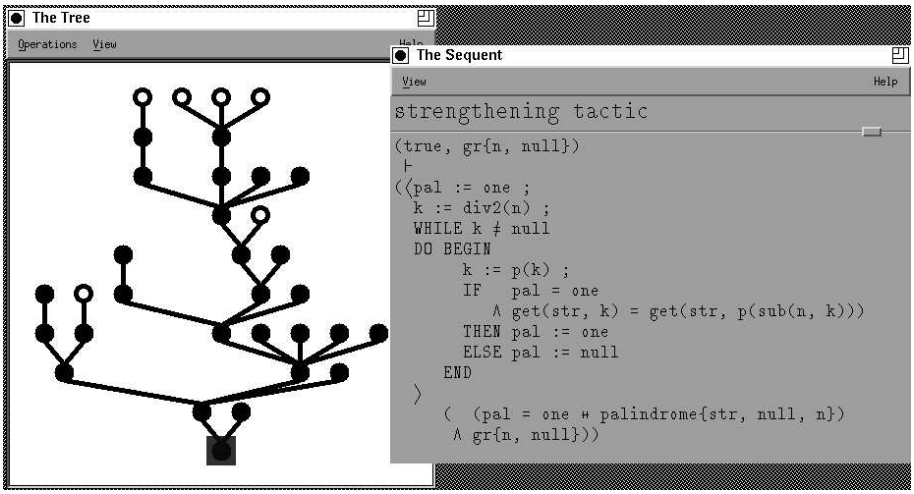
The program now is fully developed and the solutions are propagated upwards in the development tree while testing the composed solutions for acceptability. This amounts to generating and proving verification conditions. One condition that cannot be reduced further is shown on the next screen dump. It basically states

$$0 < fix24 \wedge fix24 < n \Rightarrow 0 < n$$

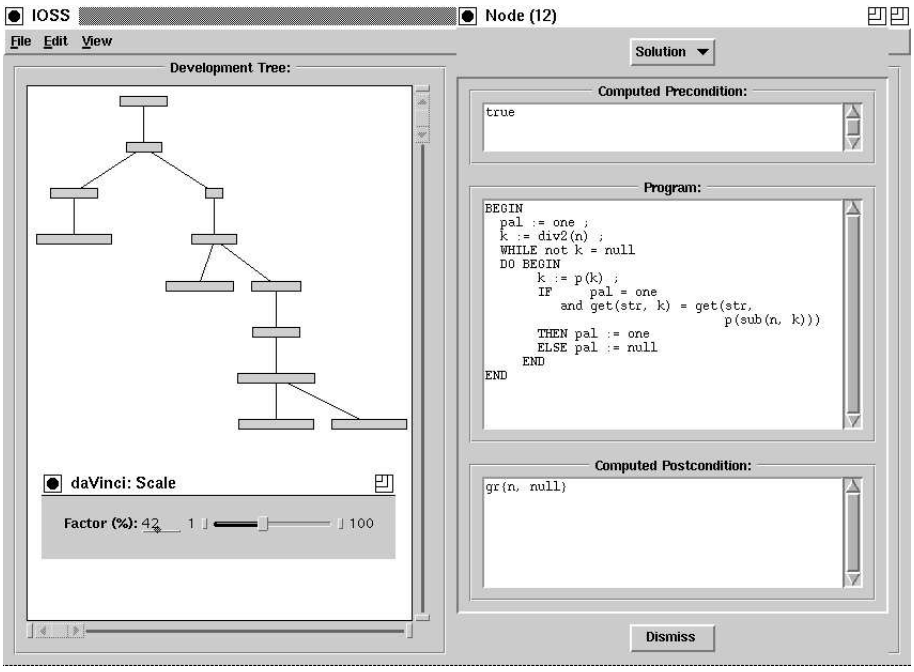
but the built-in prover cannot yet deal with transitivity of $<$.



The final proof tree together with the sequent in its root node – that contains the program with its pre- and postconditions expressed in dynamic logic – are shown next. The premises drawn as circles are the verification conditions the built-in prover could not prove. The more complex proof trees for verification conditions shown above are collapsed to single branchings in the tree below.



This is the final development tree scaled down to fit in the window. The window next to it shows the solution contained in the root node, i.e. the program we have developed.



The development tree – containing all strategy applications and all proofs – can be saved in a file for further use.

Bibliography

[BCF89] Avron Barr, Paul R. Cohen, and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume 4. Addison-Wesley, Reading, MA, 1989.

[BH84] 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.

[CB88] J. Conclin and M. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. *ACM Transactions on Office Informations Systems*, 6:303–331, October 1988.

[CGR93] Dan Craigan, Susan Gerhart, and Ted 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.

[CIP85] CIP Language Group. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L.* Number 183 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[CIP87] CIP System Group. *The Munich Project CIP. Volume II: The Program Transformation System CIP-S.* Number 292 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[Del94] Sven Delmas. Kidnapping X Applications. Unpublished Paper, TU Berlin, 1994.

[Der83] Nachum Dershowitz. *The Evolution of Programs.* Birkhäuser, Boston, 1983.

[FN92] Gene Forte and Ronald G. Norman. A self-assessment by the software engineering community. *Communications of the ACM*, 35(4):28–32, April 1992.

[Fug93] Alfonso Fuggtta. A classification of case technology. *Computer*, 26(12):25–38, December 1993.

[FW94] Michael Fröhlich and Mattias Werner. daVinci V1.3 User Manual. Technical report, Universität Bremen, 1994.

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* LNCS 130. Springer-Verlag, 1982.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Hei92] Maritta Heisel. *Formale Programmentwicklung mit dynamischer Logik.* Deutscher Universitätsverlag, Wiesbaden, 1992.

[Hei94] Maritta Heisel. A formal notion of strategy for software development. Technical Report 94–28, TU Berlin, 1994.

[HKB93] 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.

[HRS88] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Implementing verification strategies in the KIV system. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 131–140. Springer-Verlag, 1988.

[HRS89] Maritta Heisel, Wolfgang Reif, and Werner Stephan. A dynamic logic for program verification. In A. R. Meyer and M. A. Taitslin, editors, *Proceedings Logic at Botik*, number 363 in Lecture Notes in Computer Science, pages 134–145. Springer Verlag, 1989.

[HRS91] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Formal software development with the KIV system. In Michael R. Lowry and Robert D. Mc Cartney, editors, *Automating Software Design*, chapter 21, pages 547–574. AAAI Press, 1991.

[HWW94] Maritta Heisel and Debora Weber-Wulff. Korrekte Software: Nur eine Illusion? *Informatik – Forschung und Entwicklung*, 9(4):192–200, October 1994.

[LD89] Michael Lowry and Raul Duran. Knowledge-based software engineering. In *[BCF89]*, chapter 20, pages 241–322. Addison-Wesley, Reading, MA, 1989.

[Lib91] Don Libes. expect: Scripts for controlling interactive processes. *Computing Systems*, 4(2), November 1991.

[LM91] Michael R. Lowry and Robert D. McCartney, editors. *Automating Software Design.* AAAI Press, Menlo Park, 1991.

[Mil72] Robin Milner. Logic for computable functions: description of a machine implementation. *SIGPLAN Notices*, 7:1–6, 1972.

[Ost87] Leon Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In E. Losk and R. Overbeek, editors, *Ninth International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 772–773. Springer Verlag, 1988.

[Pot89] Colin Potts. A generic model for representing design methods. In *International Conference on Software Engineering*, pages 217–226. IEEE Computer Society Press, 1989.

[Rei92] Wolfgang Reif. Verification of Large Software Systems. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science. 12th Conference. New Delhi, India, December 1992. Proceedings*, LNCS 652, pages 241–252. Springer Verlag, 1992.

[RW88] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, pages 10–25, November 1988.

[Smi85] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27:43–96, 1985.

[Smi90] Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Smi93a] Douglas R. Smith. Classification approach to design. Technical Report KES.U.93.4, Kestrel Institute, November 1993.

[Smi93b] Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15(5–6):571–606, May-June 1993. special issue: Automatic Programming.

[Sou93] Jeanine Souquières. *Aide au Développement de Specifications.* Thèse d'Etat, Université de Nancy I, 1993.

[SSW92] Terry Shepard, Steve Sibbald, and Colin Wortley. A visual software process language. *Communications of the ACM*, 35(4):37–44, April 1992.

[Wil83] David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983.