Strategy-Based Program Synthesis with IOSS

Maritta Heisel* Thomas Santen Dominik Zimmermann

Abstract

This paper presents the program synthesis system IOSS (Integrated Open Synthesis System). It is based on the concept of strategy as a uniform representation of development knowledge making the integration of different synthesis methods possible. The implemented system is an instantiation of a generic system architecture designed to provide machine support for the application of formal methods in software development. The properties of the system are demonstrated by means of a sample development.

1 Introduction

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.

In order to guarantee semantic properties of a software product like correctness, formal methods have to be applied. These add considerable complexity not only to the artifacts software developers have to work with, but also to the development process itself. This means that tool support is even more indispensable than for conventional software development. On the other hand, this also means that the architecture of such systems has to be designed carefully, because satisfactory user support is essential for the acceptance of formal methods by software developers.

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.

*Technische Universität Berlin, FB Informatik – Softwaretechnik, Franklinstr. 28-29, Sekr. FR 5-6, D-10587 Berlin, email: {heisel,santen,dominik}@cs.tu-berlin.de, fax: (+49-30) 314-73488

- 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 over the progress of development. It should be easy to use, making program synthesis feasible for non deeply-involved experts and enabling the programmer to concentrate on the task at hand.

The system presented in this paper, IOSS (Integrated Open Synthesis System), supports program synthesis by application of strategies. It is a research prototype that possesses all of the above properties. The idea underlying IOSS is to support methods for program development as they are known for software development with formal methods. Examples for such methods are programming paradigms like divide-and-conquer or the method presented by Gries [Gri81]. They are formalized as so-called *strategies*. Strategies describe methods for problem solving by problem reduction. For the notion of strategy, it is not necessary to know exactly what problems and solutions look like, and when a solution is acceptable for a given problem. Instead, strategies are *generic* in these three parameters. In IOSS, problems are basically specifications, and solutions are basically imperative programs. The underlying architecture, however, is more general: it can also be used for alternative definitions of problems and solutions.

Our contribution is three-fold: First, we introduce the concept of *strategy* as a knowledge representation mechanism which makes development knowledge amenable to machine support (Section 2). Second, we provide a *generic system architecture* that is able to support software development by application of strategies (Section 3). Third, we present the *program synthesis system* IOSS as an instance of the system architecture (Section 4). An example serves to demonstrate how working with IOSS proceeds (Section 5) and illustrates the concepts that are only briefly sketched in Sections 2 and 3. A detailed discussion of strategies and the system architecture is given in [HSZ94]. We also report on experience with the implementation of the system (Section 6). In a low-budget project, its graphical user interface was designed and implemented within one person-month. It is an example of re-use and integration of heterogenous software packages that are freely available to the research community. In the concluding section we discuss related work and future improvements of the system.

2 Representing Software Development Knowledge by Strategies

There are two aspects to a method for software development: *strategies* and *heuristics*. Strategies describe possible steps during a development, e.g. how to implement a particular class of algorithms. They are the part of a method that is usually described in text books. In contrast, the ability to decide which strategy may successfully be applied in a particular situation requires human intuition and a deep understanding of the problem at hand. The

rules of thumb that experts develop when working with a technique, we call the heuristic part of their method.

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

Technically, the purpose of a strategy is to find a suitable solution to some software development problem. A strategy works by problem reduction. For a given problem, it determines a number of subproblems. From the solutions to these subproblems the strategy produces a solution to the initial problem. Finally, it tests if that solution is acceptable according to some notion of acceptability of a solution wrt. a problem. The solutions to subproblems are naturally obtained by strategy applications as well.

However, this description says nothing about interdependencies between the various subproblems and solutions. In general, the subproblems of a strategy are not independent of each other and of the solutions to other subproblems, see [Hei94]. These dependencies induce a partial ordering on the subproblems: it restricts the order in which the various subproblems can be set up and solved.

For a strategy to work, we need not only to know its dependency relation but also *how* exactly 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. All in all, 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. For IOSS, the explanation is a formal correctness proof.

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

Implementations of strategies should be independent of each other with a uniform interface between them. 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. We call an implementation of a strategy a *strategy module*. The signature shown in Figure 1 corresponds to the components of a strategy as described above.

The natural number *subpr* is the number of subproblems generated by a strategy. The Boolean matrix *dependency* represents dependencies between the children nodes 1 through

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

Figure 1: Interface of a Strategy Module

subpr. While we can describe *dependency* as an array, the remaining elements of the module are proper functions or procedures because they represent the algorithmic content of the strategy. For each subproblem, we need to know how to set it up. Thus *setup* is an array of functions. The function $setup[i]$ produces the ith subproblem from the initial problem and a list of problems and solutions. This list contains the sibling problems and their solutions on which problem i depends.

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

3 The System Architecture

Figure 2 gives a general view of the architecture. Two global data structures represent the

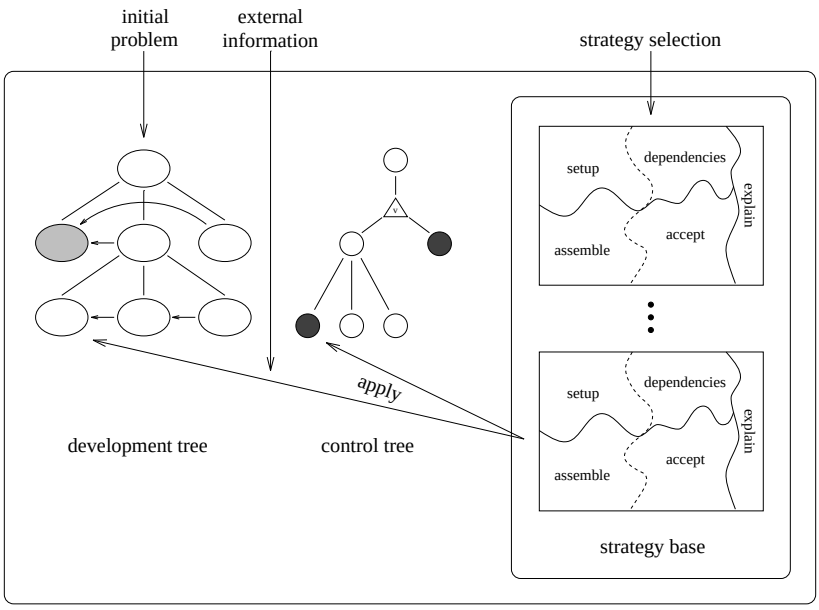

Figure 2: General view of the system architecture

state of development: the *development tree* and the *control tree*. The development tree represents the entire development that has taken place so far. A node contains a problem and its solution (once it has been found), and references to its 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 involved strategy modules. Two strategies are involved in processing one node of the development tree: a *creating* and a *reducing* strategy. Arrows on one level of the development tree indicate dependencies. The shaded node indicates that the corresponding problem has already been solved.

The data in the control tree is only concerned with the future development. Its nodes point to unsolved nodes of the development tree, thus representing open tasks and keeping track of unsolved problems and their dependencies. It provides a basis to choose the next problem to reduce. The shaded node of the development tree has no counterpart in the control tree.

There are two kinds of branchings in the control tree that stem from the dependencies between the development nodes. They tell if siblings have to be solved in a fixed left-to-right order or if they may be solved in an arbitrary order. The "normal" branching in the left subtree of the control tree in Figure 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 for the two children of the root. The leaves of the control tree point to unreduced problems. The shaded leaves may be tackled in the next step.

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

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 third major component of the architecture is the knowledge base. It represents the knowledge for strategy-based problem solving by strategy modules as described in Section 4.

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

To apply a strategy, first the problem to be reduced is selected from the leaves of the control tree. The set of reducible leaves can be determined by considering the control tree's two kinds of branchings. The reducible leaves of a tree with normal root branching are the reducible leaves of the leftmost subtree. For a triangle branching, they are the *union* of the sets of reducible leaves of all subtrees. Users may choose from the set of reducible leaves which is marked in black in Figure 2.

Second, a strategy is selected from the strategy base. Strategy selection will usually be interactive but implementations of heuristics to choose a strategy or to suggest a set of applicable ones are also conceivable. Applying the strategy to the problem means to extend the development tree with nodes for the new subproblems, install the functions

of the strategy in these nodes, and set up dependency links between them. The control tree is also extended according to the dependencies between the produced subproblems. Application of a strategy may need more information than is provided by the problem it is applied to, e.g. information needed to prove termination. Like strategy selection, providing external information usually encompasses user interaction or heuristics.

If a strategy immediately produces a solution and does not generate any subproblems, or if solutions to all subproblems of a node in the development tree have been found, the functions to assemble and accept a solution are called, and, if successful, the solution is recorded in the respective node of the development tree. Also the explanation field of the current node is filled in. The current node of the control tree is deleted. If the parent node of the deleted one has no other children, the process of solution assembly is recursively applied to that node.

In case the accept 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.

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

The loop of strategy applications terminates when the control tree is empty. 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.

4 IOSS

IOSS is an instantiation of the described architecture. It supports synthesis of provably correct imperative programs. In this section, we first establish the notions of problem, solution, acceptability, and explanation that are used for IOSS. We then give an overview over its strategy base. Finally we present the IOSS interface.

4.1 Problems, Solutions, and Explanations

In IOSS, *problems* are specifications of programs, expressed as pre- and postconditions that are formulas of first-order predicate logic. To aid focusing on the relevant parts of the task, the postcondition is divided into two parts, *invariant* and *goal*. In addition to these 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 <u>true</u>, 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]. 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.

4.2 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 needs only 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 need to be changed to achieve the one subgoal are disjoint from the result variables that need to be changed to achieve the other subgoal. 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].

4.3 Interface

Figure 3 shows a snapshot the program synthesis described in Section 5. On the left-hand

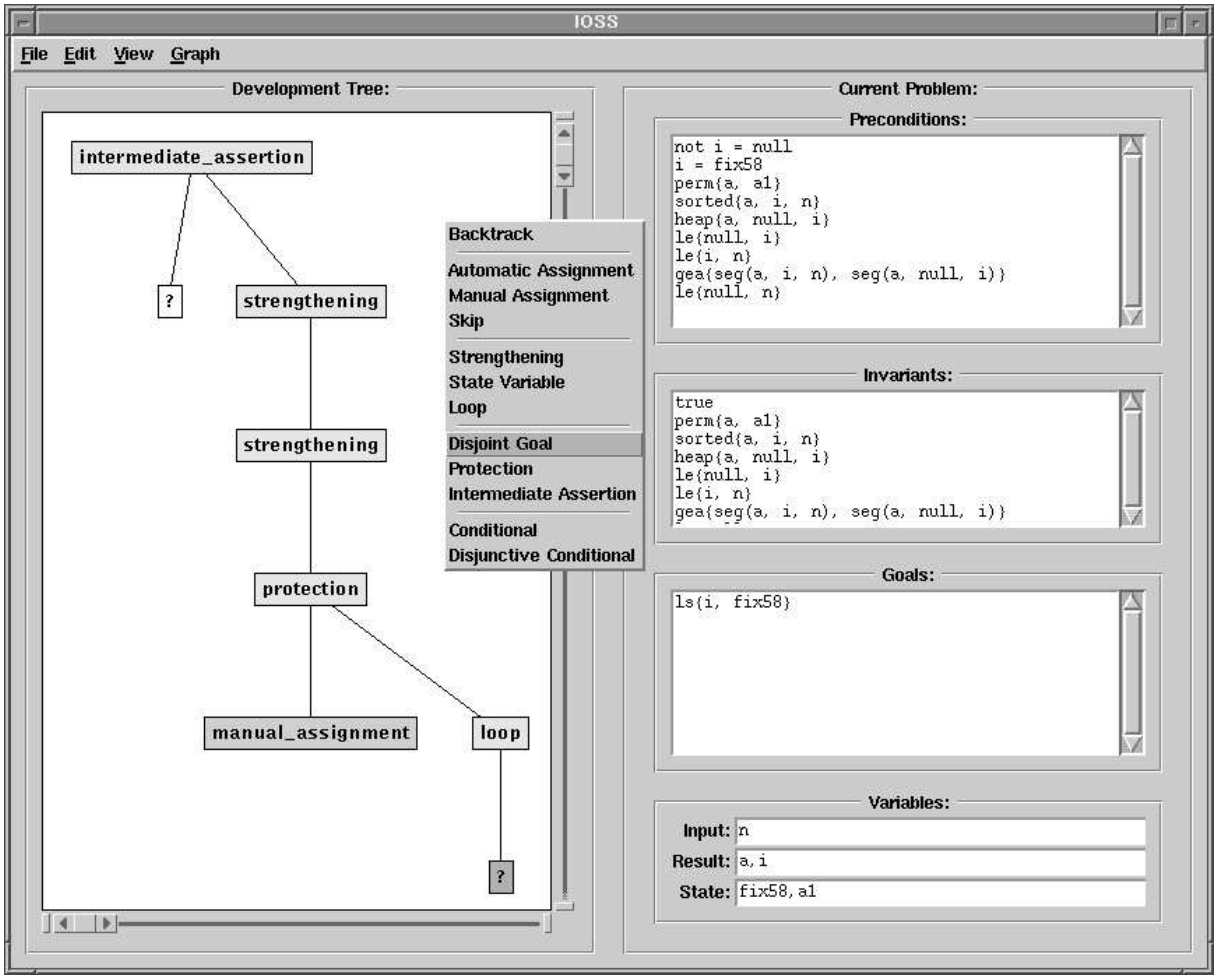

Figure 3: The IOSS interface

side of the window the development tree is displayed. Different colors and shades of the nodes visualize different states. The user can re-scale the tree, hide subgraphs, or view nodes. A separate window pops up for each node; several nodes can be inspected at the

same time. On the right-hand side the user sees the current problem. To apply a strategy one chooses one from the menu of strategies shown in the center. Whenever a strategy requires user input, the user is prompted for it in a window.

5 An Example

In this section we present a few selected steps from a sample development with IOSS 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 4(a)[1]. 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.

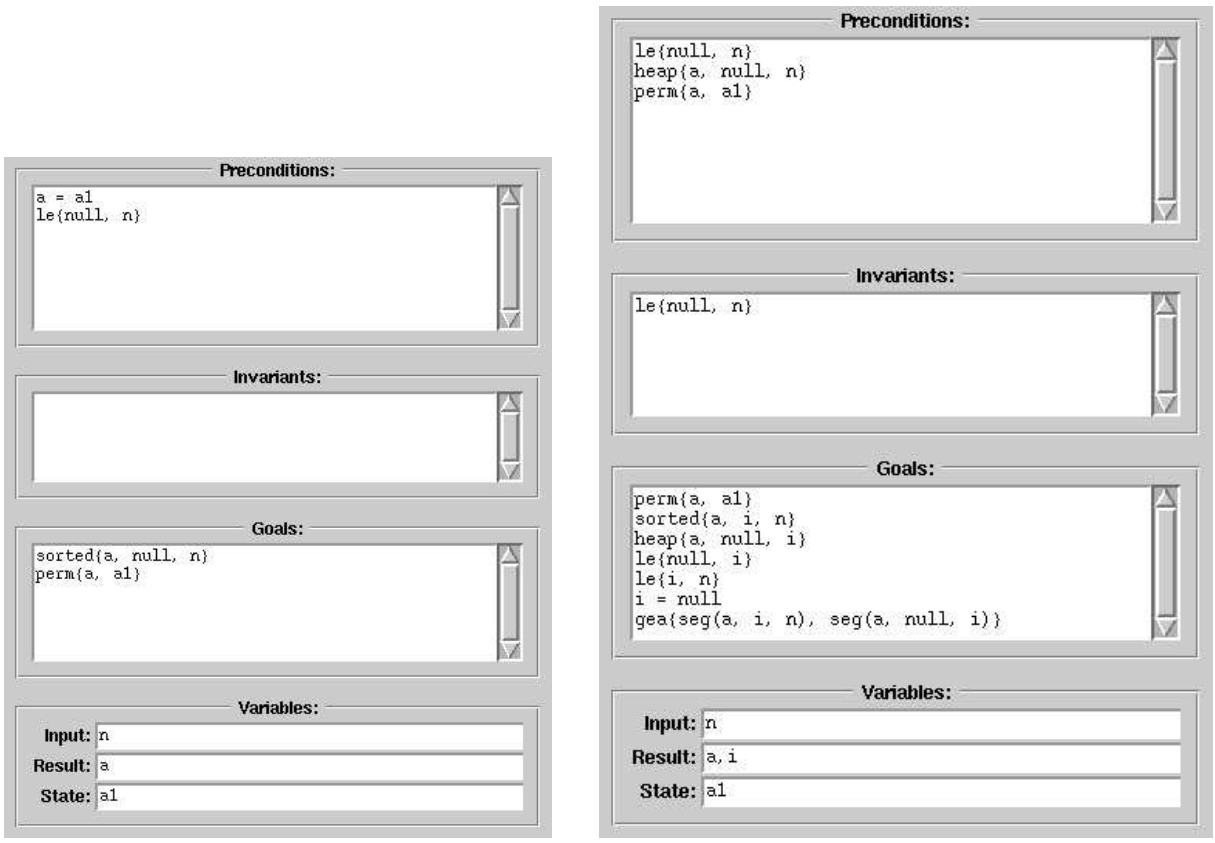

Figure 4: The problems (a) initially and (b) before the initialization of the loop

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.

[1] IOSS uses a prefix-ascii notation for functions and predicates. Variables, constants, predicates, and functions are defined in a signature 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 $2i+2$.

To start with, we apply the *intermediate assertion* strategy to the initial problem, because it allows us to choose which part of the compound we want to develop first. The strategy prompts us for an intermediate assertion. The one we need as precondition for the second part is 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 is always a heap and other domain-specific knowledge to replace these goals by stronger ones, resulting in the problem shown in Figure 4 (b).

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 being the invariants 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 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 wrt. its specification.

After application of the *loop* strategy to the second subproblem yielded by the *protection*

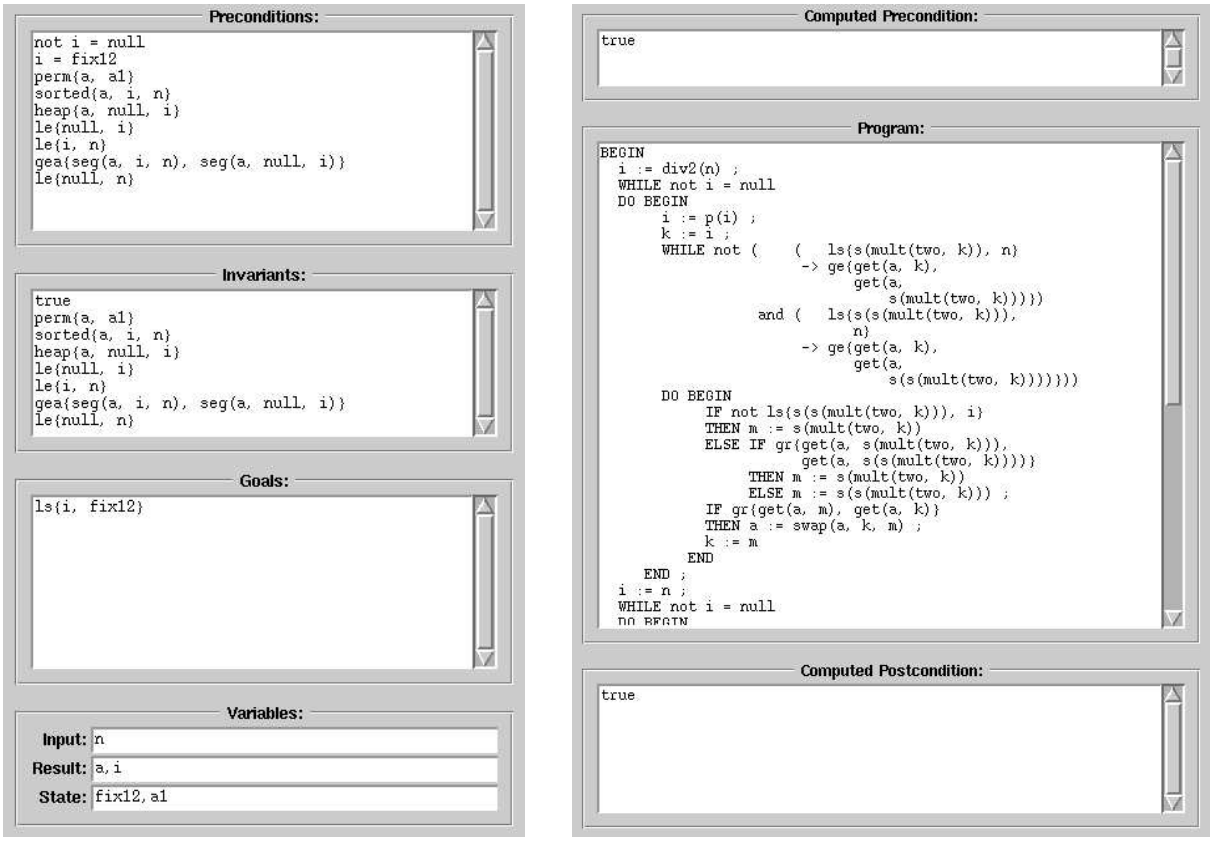

Figure 5: Problem for the loop body (a) and solution to the initial problem (b)

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 wrt. 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 5(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 of the problem (`ls{i,fix12}`) to be the goal for the first part of the compound. The invariants of the problem are automatically determined to be 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 ($a[0]$) and last ($a[i]$) element of the heap, and restoring the heap for the unsorted segment of the array ($a[0 \ldots i-1]$).

Since the unsorted segment is a heap except for the first element (resulting from the swapping), we restore the heap by letting this element "descend down" in the tree. This again is achieved by a loop. It is synthesized with a similar approach as the first loop. We first apply the *protection* strategy. The first subproblem of it 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 swapped, 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 *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 total development is completed[3], we can inspect the root node and have a look at the solution to the initial problem, shown in Figure 5(b). Figure 6 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 in total 54 strategy applications.

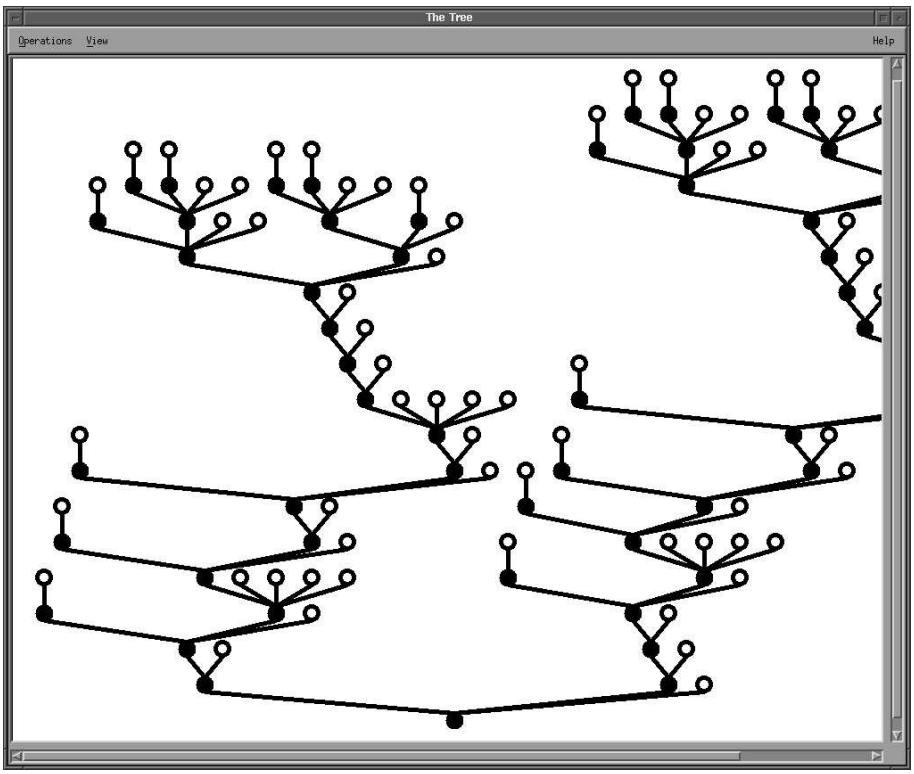

Figure 6: Proof tree for the heapsort algorithm.

6 Experience with Re-Use and Integration

The implementation of IOSS has been carried out in two steps. First, the kernel system has been implemented as an instance of the proposed architecture. Second, a graphical user interface has been designed and implemented on top of the kernel system.

The Kernel System. The basis for the implementation of IOSS is the *Karlsruhe Interactive Verifier* (KIV), 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. Strategies are implemented as collections of `PPL`-functions in separate modules. New strategies can be incorporated in a routine way. Currently a template file for new strategies supports this process; for the future, we envision tool support relieving the implementor of anything but the peculiarities of the newly implemented strategy.

A severe restriction of KIV is its command-line interface. There is no reasonable way to bring into effect the potential to inspect the state of development and to take advantage of the freedom of choice provided by the architecture. The need for a more sophisticated interface became apparent, if IOSS was ever to be used by anyone else but its creators.

The Graphical Interface. Since we had very limited resources in terms of person-power to realize the interface, we decided to rely as much as possible on existing software packages and toolkits. The interface was to be built with minimal changes to the kernel system. We needed a means to transfer data from KIV to whichever interface system we would use. For

the visualization of the state of development, we needed a graph layout system. Moreover, we wanted to avoid programming on a level such as the X Window Toolkit, since this is a tedious, time-consuming task.

With the following packages we found just what we needed:

`Tcl` – A simple, extensible scripting language providing generic programming facilities. Each application can implement new features as `Tcl` commands [Ous94].

`Tk` – An extension to `Tcl` providing a toolkit for the X Window System. `Tk` extends the core `Tcl` facilities by commands for building user interfaces. It hides much detail C programmers must address when constructing a user interface [Ous94].

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

`TkSteal` – An extension to `Tk` to integrate stand-alone X applications in a `Tk`-built interface [Del94].

`daVinci` – A generic visualization system for directed graphs [FW94].

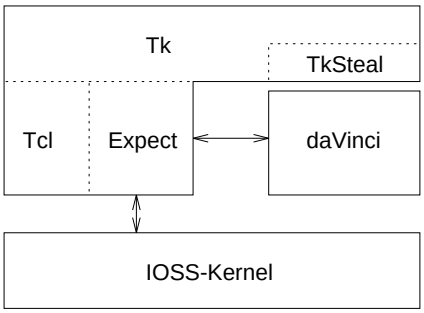

Figure 7: System integration for the IOSS interface

Figure 7 illustrates the integration of these packages to construct the graphical interface for IOSS. `expect` controls the command-line interface of KIV and the application interface of `daVinci`. `TkSteal` provides "interface sugar" for the graphical interface. It integrates `daVinci` with the other parts of the IOSS interface.

We think it is remarkable how little effort was required to build the interface. It took only one person-month to build it in its current shape. Only 800 LOC needed to be written in `Tcl`, 116 LOC of additional code had to be written in `PPL`.

7 Discussion

The example of Section 5 has given an impression of how it "feels" to work with the system. The user controls the development process by selecting strategies and providing additional information necessary to apply the selected strategies. The intermediate development states essentially consist of the development tree and the current problem. The current problem completely and unambiguously states the task that is to be solved. For software developers, however, it is often important to understand the context in which this task is to be solved. This context information is provided by the development tree. Browsing the tree gives an overview of the whole development.

Since a broad variety of strategies is at the user's disposal, the system is able to support different approaches to program development. Different methods can be applied in combination. This is achieved by a common interface: all strategies produce and consume the same kind of information. The uniform design and local implementation of strategy modules makes it possible to incorporate new strategies in a routine way.

7.1 Related Work

As already stated, IOSS is built on top of a former version of KIV. In its current version [Rei92], KIV supports the verification of program modules according to a fixed strategy. The degree of automation is impressive. Since in program verification, not only the specification but also the program is known, automation is much easier achieved than in program synthesis.

The paradigm motivating the development of KIV as well as IOSS is *tactical theorem proving*. The idea is to use a metalanguage to write programs that construct proofs in a logical formalism. The basic steps such programs can perform are called *tactics* and are derived rules of the basic formalism. The control structures of the metalanguage are called *tacticals*. This approach was introduced in Edinburgh LCF [GMW79] and is used today in the Nuprl system [Con86] and others.

The approach underlying KIDS (Kestrel Interactive Development System) [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. In KIDS, however, there is no general concept of design tactics and how to incorporate a new one into the system.

LOPS [BH84] is a system for deductive program synthesis following a fixed procedure. The systems CIP-S [CIP87], see also this volume, and PROSPECTRA [HKB93] support transformational program synthesis. All of these systems are designed to support a specific synthesis method. It was not the intention of their creators to integrate these with other ones.

7.2 Future Improvements

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.

Until now, there is only one version of the development tree. 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 to guide the selection of strategies. 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 also provides further documentation of the development.

Finally, we want to explore other application domains besides program synthesis. A promising candidate for this project is specification acquisition. In this phase of the life cycle, the transition from informal requirements to formal specifications is made. The problems do not have a formal semantics but the solutions do. Our long-term goal is to provide machine support for various phases of the software life cycle, where a common system architecture provides a strong potential for integration.

References

[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.

[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.

[Con86] R. L. Constable, et al. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, Englewood Cliffs, NJ, 1986.

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

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

[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.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF.* Number 78 in Lecture Notes in Computer Science. Springer Verlag, New York, 1979.

[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.

[HSZ94] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. A system architecture for strategy-based software development. Submitted for publication, 1994.

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

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

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

[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.

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